Bug Fixing Practices within Free/ Libre Open Source Software Development Teams[1]

Kevin Crowston, Syracuse University, USA

Barbara Scozzi, Politecnico di Bari, Italy

ABSTRACT

Free/libre open source software (FLOSS, e.g., Linux or Apache) is primarily developed by distributed teams. Developers contribute from around the world and coordinate their activity almost exclusively by means of email and bulletin boards, yet some how profit from the advantages and evade the challenges of distributed software development. In this article we investigate the structure and the coordination practices adopted by development teams during the bug-fixing process, which is considered one of main areas of FLOSS project success. In particular, based on a codification of the messages recorded in the bug tracking system of four projects, we identify the accomplished tasks, the adopted coordination mechanisms, and the role undertaken by both the FLOSS development team and the FLOSS community. We conclude with suggestions for further research.

Keywords: *bug-fixing processes; coordination practices; free/libre open source software; software development*

INTRODUCTION

In this article, we investigate the coordination practices for software bug fixing in Free/Libre open source software (FLOSS) development teams. Key to our interest is that most FLOSS software is developed by distributed teams, i.e., geographically dispersed groups of individuals working together over time towards a common goal (Ahuja *et al.*, 1997, p. 165; Weisband, 2002). FLOSS developers contribute from around the world, meet face to face infrequently, if at all, and coordinate their activity primarily by means of computer mediated communications (Raymond, 1998; Wayner, 2000). As a result, distributed teams employ processes that span traditional boundaries of place and ownership. Since such teams are increasingly commonly used in a diversity of settings, it is important to understand how team members can effectively coordinate their work.

The research literature on distributed work and on software development specifically emphasizes the difficulties of distributed software development, but the case of FLOSS development presents an intriguing counter-example, at least in part: a number of projects have been outstandingly successful. What is perhaps most surprising is that FLOSS development teams seem not to use many traditional coordination mechanisms such as formal planning, system level design, schedules and defined development processes (Mockus *et al.*, 2002, p. 310). As well, many (though by no means all) programmers contribute to projects as volunteers, without working for a common organization and/or being paid.

The contribution of this article is to document the process of coordination in effective FLOSS teams for a particularly important process, namely bug fixing. These practices are analyzed by adopting a process theory, i.e., we investigate which tasks are accomplished, how and by whom they are assigned, coordinated, and performed. In particular, we selected four FLOSS projects, inductively coded the steps involved in fixing various bugs as recorded in the projects' bug tracking systems and applied coordination theory to identify tasks and coordination mechanisms carried out within the bug-fixing process.

Studying coordination of FLOSS processes is important for several reasons. First, FLOSS development is an important phenomenon deserving of study for itself. FLOSS is an increasingly important commercial issue involving all kind of software firms. Million of users depend on systems such as Linux and the Internet (heavily dependent on FLOSS software tools) but as Scacchi notes "little is known about how people in these communities coordinate software development across different settings, or about what software processes, work practices, and organizational contexts are necessary to their success" (Scacchi, 2002, p. 1; Scacchi, 2005). Understanding the reasons that some projects are effective while others are not is a further motivation for studying the FLOSS development processes. Second, studying how distributed software developers coordinate their efforts to ensure, at least in some cases, high-performance outcomes has both theoretical and managerial implications. It can help understanding coordination practices adopted in social collectives that are not governed, at least apparently, by a formal organizational structure and are characterized by many other discontinuities that is, lack of coherence in some aspects of the work setting: organization, function, membership, language, culture, etc. (Watson-Manheim *et al.*, 2002). As to the managerial implications, distributed teams of all sorts are increasingly used in many organizations. The study could be useful to managers that are considering the adoption of this organizational form not only in the field of software development.

The remainder of the article is organized as follows. In Section 2 we discuss the theoretical background of the study. In Section 3 we stress the relevance of process theory so explaining why we adopted such a theoretical approach. We then describe coordination theory and use it to describe the bug-fixing process as carried out in traditional organizations. The research methodology adopted to study the bug-fixing process is described in Section 4. In Section 5 and 6 we describe and discuss the study's results. Finally, in Section 7 we draw some conclusions and propose future research directions.

BACKGROUND

In this section we provide an overview of the literature on software development in distributed environment and the FLOSS phenomenon.

Distributed Software Development

Distributed teams offer numerous potential benefits, such as the possibility to perform different projects all over the world without paying the costs associated with travel or relocation, or ease of reconfiguring teams to quickly respond to changing business needs (DeSanctis & Jackson, 1994; Drucker, 1988) or to exploit available competences and distributed expertise (Grinter *et al.*, 1999; Orlikowski, 2002). Distributed teams seem particularly attractive for software development, because software, as an information product, can be easily transferred via the same systems used to support the teams (Nejmeh, 1994; Scacchi, 1991). Furthermore, while many developed countries face a shortage of talented software developers, some developing countries have a pool of skilled professionals available, at lower cost (Metiu & Kogut, 2001, p. 4; Taylor, 1998). As well, the need to have local developers in each country for marketing and localization have made distributed teams a business need for many global software corporations (Herbsleb & Grinter, 1999b, p. 85).

While distributed teams have many potential benefits, distributed workers face many real challenges. The specific challenges vary from team to team, as there is a great diversity in their composition and in the setting of distributed work. As mentioned, distributed work is characterized by numerous discontinuities that generate difficulties for members in making sense of the task and of communications from others, or produce unintended information filtering (de Souza, 1993). These interpretative difficulties make it hard for team members to develop a shared mental model of the developing project (Curtis *et al.*, 1990, p. 52). A lack of common knowledge about the status, authority and competencies of participants brought together for the first time can be an obstacle to the creation of a social structure and the development of team norms (Bandow, 1997, p. 88) and conventions (Weisband, 2002), thus frustrating the potential benefits of increased flexibility.

Numerous studies have investigated social aspects of software development teams (e.g., Curtis *et al.*, 1988; Humphrey, 2000; Sawyer & Guinan, 1998; Walz *et al.*, 1993). These studies conclude that large system development requires knowledge from many domains, which is thinly spread among different developers (Curtis et al., 1988). As a result, large projects require a high degree of knowledge integration and the coordinated efforts of multiple developers (Brooks, 1975). However, coordination is difficult to achieve as software projects are non-routine, hard to decompose perfectly and face requirements that are often changing and conflicting, making development activities uncertain.

Unfortunately, the problems of software development seem to be exacerbated when development teams work in a distributed environment with a reduced possibility for informal communication (Bélanger, 1998; Carmel & Agarwal, 2001; Herbsleb & Grinter, 1999a)..

In response to the problems created by discontinuities, studies of distributed teams stress the need for a significant amount of time spent in "community building" (Butler *et al.*, 2002). In particular, members of distributed teams need to learn how to communicate, interact and socialize using CMC. Successful distributed cross-functional teams share knowledge and information and create new practices to meet the task-oriented and social needs of the members (Robey et al., 2000). Research has shown the importance of formal and informal adopted coordination mechanisms, information sharing for coordination and communications, and conflict management for project's performance and quality (Walz et al., 1993). However, the processes of coordination suitable for distributed teams are still open topics for research (e.g., Orlikowski, 2002).

The FLOSS Phenomenon: A Literature Overview

The growing literature on FLOSS has addressed a variety of questions. Some researchers have examined the implications of free software from economic and policy perspectives (e.g., Di Bona *et al.*, 1999; Kogut & Metiu, 2001; Lerner & Tirole, 2001) as well as social perspective (e.g., Bessen, 2002; Franck & Jungwirth, 2002; Hann *et al.*, 2002; Hertel *et al.*, 2003; Markus *et al.*, 2000). Other studies examine factors for the success of FLOSS projects (Hallen *et al.*, 1999; Leibovitch, 1999; Pfaff, 1998; Prasad, n.d.; Valloppillil, 1998; Valloppillil & Cohen, 1998, Crowston and Scozzi, 2003). Among them, an open research question deals with the analysis of how the contributions of multiple developers can be brought into a single working product (Herbsleb & Grinter, 1999b). To answer such a question, a few authors have investigated the processes of FLOSS development (e.g., Jensen & Scacchi, 2005; Stewart & Ammeter, 2002). The most well-known model developed to describe FLOSS organization structure is the bazaar metaphor proposed by Raymond (1998). As in a bazaar, FLOSS developers autonomously decide the schedule and contribution modes for software development, making a central coordination action superfluous. While still popular, the bazaar metaphor has been broadly criticized (e.g., Cubranic, 1999). According to its detractors, the bazaar metaphor disregards some aspects of the FLOSS development process, such as the importance of the project leader control, the existence of de-facto hierarchies, the danger of information overloads and burnout, the possibility of conflicts that cause a loss of interest in a project or forking, and the only apparent openness of these communities (Bezroukov, 1999a, 1999b).

Nevertheless, many features of the bazaar model do seem to apply. First, many teams are largely self-organizing, often without formally appointed leaders or formal indications of rank or role. Individual developers may play different roles in different projects or move from role to role as their involvement with a project changes. For example, a common route is for an active user to become a co-developer by contributing a bug fix or code for a new feature, and for active and able co-developers to be invited to become members of the core. Second, coordination of project development happens largely (though not exclusively) in a distributed mode. Members of a few of the largest and most well-established projects do have the opportunity to meet face-to-face at conferences (e.g., Apache developers at *ApacheCon*), but such an opportunity is rare for most project members. Third, non-member involvement plays an important role in the success of the teams. Non-core developers contribute bug fixes, new features or documentation, provide support for new users and fill a variety of other roles in the teams. Furthermore, even though the core group provides a form of leadership for a project, they do not exercise hierarchical control. A recent study documented that self-assignment is a typical coordination mechanism in FLOSS projects and direct assignment are nearly non-existent (Crowston *et al.*, 2005). In comparison to traditional organizations then, more people can share power and be involved in FLOSS project activities. However, how these diverse contributions can be harnessed to create a coherent product is still an important question for research. Our article addresses this question by examining in detail a particular case, namely, coordination of bug-fixing processes.

CONCEPTUAL DEVELOPMENT

In this section, we describe the theoretical perspectives we adopted to examine the coordination of bug fixing, namely, a process-oriented perspective and the coordination theory. We also introduce the topic of coordination and discuss the literature on coordination in software development and the (small) literature on coordination in FLOSS teams.

Processes as Theories

Most theories in organizational and information system research are variance theories, comprising constructs or variables and propositions or hypotheses linking them. By adopting a statistical approach, such theories predict the levels of dependent or outcome variables from the levels of independent or predictor variables, where the predictors are seen as necessary and sufficient for the outcomes. In other words, the logical structure of such theories is that if concept *a* implies concept *b*, then more of *a* means more (or less) of *b*. For example, the hypothesis that the adoption of ICT makes organization more centralized, examined as a variance theory, is that the level of organization centralization increases with the number of new ICTs adopted.

An alternative to a variance theory is a process theory (Markus & Robey, 1988). Rather than relating levels of variables, process theories explain how outcomes of interest develop through a sequence of events. In that case, antecedents are considered as necessary but not sufficient for the outcomes (Mohr, 1982). For example, a process model of ICT and centralization might posit several steps each of which must occur for the organization to become centralized, such as development and implementation of an ICT system and use of the system to control decision premises and program jobs, resulting in centralization of decision making as an outcome (Pfeffer, 1978). However, if any of the intervening steps does not happen, a different outcome may occur. For example, if the system is used to provide information directly to lower-level workers, decision making may become decentralized rather centralized (Zuboff, 1988). Of course, theories may contain some aspects of both variance and process theories (e.g., a variance theory with a set of contingencies), but for this discussion, we describe the pure case. Typically, process theories are of some transient process leading to exceptional outcomes, e.g., events leading up to an organizational change or to acceptance of a system. However, we will focus instead on what might be called "everyday" processes: those performed regularly to create an organization's products or services. For example, Sabherwal and Robey (1995) described and compared the processes of information systems development for 50 projects to develop five clusters of similar processes.

Kaplan (1991, p. 593) states that process theories can be "valuable aids in understanding issues pertaining to designing and implementing information systems, assessing their impacts, and anticipating and managing the processes of change associated with them". The main advantage of process theories is that they can deal with more complex causal relationships than variance theories. Also they embody a fuller description of the steps by which inputs and outputs are related, rather than noting the relationship between the levels of input and output variables. Specifically, representing a process as a sequence of activities provides insight into the linkage between individual work and processes, since individuals perform the various activities that comprise the process. As individuals change what they do, they change how they perform these activities and thus their participation in the process. Conversely, process changes demand different performances from individuals. ICT use might simply make individuals more efficient or effective at the activities they have always performed. However, an interesting class of impacts involves changing which individuals perform which activities and how activities are coordinated. Such an analysis is the aim of this article.

Coordination of Processes

In this subsection, we introduce the topic of coordination and present the fundamentals of coordination theory. Studying coordination means analyzing how dependences that emerge among the components of a system are managed. That stands for any kind of system, e.g., social, economics, organic, information system. Hence, the coordination of the components of a system is a phenomenon with a universal relevance (Boulding, 1956). The above definition of coordination is consistent with the large body of literature developed in the field of organization theory (e.g., Galbraith, 1973; Lawrence & Lorsch, 1967; Mintzberg, 1979; Pfeffer & Salancik, 1978; Thompson, 1967) that emphasizes the importance of interdependence.

For example, according to Thompson (1967), organizational action consists of the coordination of the interdependences and the reduction of the costs associated to their management. Two components/systems are said to be interdependent if the action carried out by one of them affect the other one's output or performance (McCann & Ferry, 1979; Mohr, 1971; Victor & Blackburn, 1987). For space reason, it is not possible to present all the contributions on coordination in the literature, but because of its relevance, we here briefly report on Thompson's seminal work. Thompson (1967) identified three main kinds of interdependence, namely *pooled*, *sequential* and *reciprocal interdependence*. Pooled interdependence occurs among organization units that have the same goal but do not directly collaborate to achieve it. Sequential dependence emerges among serial systems. A reciprocal dependence occurs when the output of a system is the input for a second system and vice versa. The three kinds of interdependence require coordination mechanisms whose cost increases going from the first to the last one. The coordination by standardization, i.e., routine and rules, is sufficient to manage pooled-dependant systems. Coordination by plan implies the definition of operational schemes and plans. It can be used to manage pooled and sequential dependences. Finally, coordination *by mutual adjustment* is suitable for the management of reciprocal dependences.

The interest devoted by scholars and practitioners to the study of coordination problems has recently increased due to the augmented complexity of products, production processes and to the rapid advancement in science and technology. To address these issues scholars have developed coordination theory, a systemic approach to the study of coordination (Malone & Crowston, 1994). Coordination theory synthesizes the contributions proposed in different disciplines to develop a systemic approach to the study of coordination. Studies on coordination have been developed based on two level of analysis, a micro and a macro level. In particular, most organization studies adopt a macro perspective, so considering dependencies emerging among organizational units. Other studies adopt a micro perspective, so considering dependencies emerging among single activities/actors. Coordination theory adopts the latter perspective and, in particular, focuses on the analysis of dependencies among activities (rather that actors). Hence, it is particularly useful to the description and analysis of organizational processes, which can be defined as a set of interdependent activities aimed to the achievement of a goal (Crowston, 1997; Crowston & Osborn, 2003). In particular, this approach has the advantage of making it easier to model the effects of reassignments of activities to different actors, which is common in process redesign efforts. We adopted this perspective because the study focuses on analyzing coordination mechanisms within processes.

Consistent with the definition proposed above, Malone and Crowston (1994) analyzed group action in terms of *actors* performing *interdependent tasks*. These tasks might require or create *resources* of various types. For example, in the case of software development, actors include the customers and various employees of the software company. Tasks include translating aspects of a customer's problem into system requirements and code, or bug reports into bug fixes. Finally, resources include information about the customer's problem and analysts' time and effort. In this view, actors in organizations face *coordination problems* arising from dependencies that constrain how tasks can be performed.

It should be noted that in developing this framework, Malone and Crowston (1994) describe coordination mechanisms as relying on other necessary group functions, such as decision making, communications, and development of shared understandings and collective sensemaking (Britton *et al.*, 2000; Crowston & Kammerer, 1998). To develop a complete model of a process would involve modeling all of these aspects: coordination, decision making, and communications. In this article though, we will focus on the coordination aspects, bracketing the other phenomenon.

Coordination theory classifies dependencies as occurring between a task and a resource, among multiple tasks and a resource, and among a task and multiple resources. Dependencies between a task and a resource are due to the fact that a task uses or creates a resource. Shared use of resources can in turn lead to dependencies between the tasks that use or create the resource. These dependencies come in three kinds. First, the flow dependence resembles the Thompson's sequential dependency. Second, the fit dependence occurs when two activities collaborate in the creation of an output (though in the case where the output is identical, this might better be called synergy, since the benefit is that duplicate work can be avoided). Finally, the share dependency emerges among activities that share the use of a resource. Dependencies between a task and multiple resources are due to the fact that a task uses, creates or produces multiple resources or a task uses a resource and create another resource. For example, in the case of software development, a design document might be created by a design task and used by programming tasks, creating a fit dependency, while two development tasks might both require a programmer (a share dependency) and create outputs that must work together (a fit dependency).

The key point in this analysis is that dependencies can create problems that require additional work to manage (or provide the opportunity to avoid duplicate work). To overcome the coordination problems created by dependences, actors must perform additional work, which Malone and Crowston (1994) called *coordination mechanisms*. For example, if particular expertise is necessary to perform a particular task (a task-actor dependency), then an actor with that expertise must be identified and the task assigned to him or her. There are often several coordination mechanisms that can be used to manage a dependency. For example, mechanisms to manage the dependency between an activity and an actor include (among others): (1) having a manager pick a subordinate to perform the task; (2) assigning the task to the first available actor; and (3) having a labour market in which actors bid on jobs. To manage a usability subdependency, the resource might be tailored to the needs of the consumer (meaning that the consumer has to provide that information to the producer) or a producer might follow a standard so the consumer knows what to expect. Mechanisms may be useful in a wide variety of organizational settings. Conversely, organizations with similar goals achieved using more or less the same set of activities will have to manage the same dependencies, but may choose different coordination mechanisms, thus resulting in different processes. Of course, the mechanisms are themselves activities that must be performed by some actors, and so adding coordination mechanisms to a process may create additional dependences that must themselves be managed.

Coordination in Software Development

Coordination has long been a key issue in software development (e.g., Brooks, 1975; Conway, 1968; Curtis et al., 1988; Faraj & Sproull, 2000; Kraut & Streeter, 1995; Parnas, 1972). For example, Conway (1968) observed that the structure of a software system mirrors the structure of the organization that develops it. Both Conway (1968) and Parnas (1972) studied coordination as a crucial part of software development. Curtis et al. (1988) found that in large-scale software project, coordination and communication are among the most crucial and hard-to-manage problems. To address such problems, software development researchers have proposed different coordination mechanisms such a planning, defining and following a process, managing requirements and design specifications, measuring process characteristics, organizing regular meetings to track progress, implementing workflow systems, among the others.

Herbsleb and Grinter (1999b), in a study of geographically-distributed software development within a large firm, showed that some of the previously mentioned coordination mechanisms—namely integration plans, component-interface specifications, software processes and documentation—failed to support coordination if not properly managed. The mechanisms needed to be modified or augmented (allowing for the filling in of details, handling exceptions, coping with unforeseen events and recovering from errors) to allow the work to proceed. They also showed that the primary barriers to coordination breakdowns were the lack of unplanned contact, knowing whom to contact about what, cost of initiating a contact, ability to communicate effectively and lack of trust or willingness to communicate openly.

Kraut and Streeter (1995), in studying the coordination practices that influence the sharing of information and success of software development, identified the following coordination techniques: formal-impersonal procedures (projects documents and memos, project milestones and delivery schedules, modification request and error-tracking procedures, data dictionaries), formal-interpersonal procedures (status-review meetings, design-review meetings, code inspections), informal-interpersonal (group meetings and co-location of requirements and development staff, electronic communication such as e-mail and electronics bulletin boards, and interpersonal network). Their results showed the value of both informal and formal interpersonal communication for sharing information and achieving coordination in software development. Note though that this analysis focuses more the media for exchanging information rather than particular dependencies or coordination mechanisms that might be executed via these media. That is, once you have called a group meeting, what should you talk about?

Coordination in FLOSS Development

A few studies have examined the work practices and coordination modes adopted by FLOSS teams in more detail, which is the focus of this article (Iannacci, 2005; Scacchi, 2002; Weber, 2004). Cubranic (1999) observed that the main media used for coordination in FLOSS development teams were mailing lists. Such a low-tech approach is adopted to facilitate the participation of would-be contributors, who may not have access to or experience with more sophisticated technology. The geographical distribution of contributors and the variability in time of contributors precluded the use of other systems (e.g., systems that support synchronous communication or prescriptive coordination technology, such as workflow systems). Mailing lists supported low-level coordination needs. Also, Cubranic (1999) found no evidence of the use of higher-level coordination, such as group decision making, knowledge management, task scheduling and progress tracking. As they are the main coordination mechanisms, the volume of information within mailing lists can be huge. Mailing lists are therefore often unique repositories of source information on design choices and evolution of the system. However, dealing with this volume of information in large open source software projects can require a large amount of manual and mental effort from developers, who have to rely on their memory to compensate for the lack of adequate tools and automation.

In a well-known case study of two important FLOSS projects, namely Apache and Mozilla, Mockus et al. (2002) distinguished explicit (e.g., interface specification processes, plans, etc.) and implicit coordination mechanisms adopted for software development. They argued that, because of its software structure, the Apache development team had primarily adopted implicit coordination mechanisms. The basic server was kept small. Core developers worked on what interested them and their opinion was fundamental when adding new functionality. The functionality beyond the basic server was added by means of various ancillary projects, developed by a larger community that interacted with Apache only through defined interfaces. Such interfaces coordinate the effort of the Apache developers: as they had to be designed based on what Apache provided, the effort of the Apache core group was limited. As a result, coordination relied on the knowledge of who had expertise in a given area and general communication on who is doing what and when. On the other hand, in the Mozilla project, because of the interdependence among modules, considerable effort is spent in coordination. In this case, more formal and explicit coordination mechanisms were adopted (e.g., module owners were appointed who had to approve all changes in their module).

Jensen & Scacchi (2005) modelled the software-release process in three projects, namely Mozilla, Apache and NetBeans. They identified tasks, their dependencies and the actors performing them. However, they did not analyze the coordination issues in depth and did not focus specifically on the bug-fixing process, which is the aim of this article. Rather, their final goal was to study the relationships among the three communities that form a Web Information Infrastructure.

Iannacci (2005) adopted an organizational perspective to study coordination processes within a single large-scale and well-known

FLOSS development project, Linux. He identified three main (traditional) coordination mechanisms, namely standardization, loose coupling and partisan mutual adjustment. Standardization is a coordination mechanism to manage pooled dependencies emerging among different contributors. It implies the definition of well-defined procedures, such as in the case of patch submission or bug-fixing procedures. Loose coupling is used to manage sequential dependencies among the different subgroups of contributors. It is the coordination mechanisms used to, for example, incorporating new patches. Finally, partisan mutual adjustment is a mechanism used to manage what Iannacci (2005) called networked interdependencies, an extension of the reciprocal dependencies as proposed by Thompson (1967). Networked interdependencies are those emerging among contributors to specific part of the software. Partisan mutual adjustment produces a sort of structuring process so creating an informal (sub-)organization. However, these findings are based on a single exceptional case, the Linux project, making it unclear how much can be generalized to smaller projects. Indeed, most of the existing studies are of large and well-known projects and focused on the development process. To our knowledge, no studies have analyzed the bug-fixing process in depth within small FLOSS development teams.

A Coordination Theory Application: The Bug-Fixing Process

To ground our discussion of coordination theory, we will briefly introduce the bug-fixing process, which consists of the tasks needed to correct software bugs. We decided to focus on the bug-fixing process for three reasons. First, bug fixing provides "a microcosm of coordination problems" (Crowston, 1997). Second, a quick response to bugs has been mentioned as a particular strength of the FLOSS process: as Raymond (1998) puts it, "given enough eyeballs, all bugs are shallow". Finally, it is a process that involves the entire developer community and thus poses particular coordination problems. While there have been several studies of FLOSS bug fixing, few have analyzed coordination issues within bug-fixing process by adopting a process view. For example, Sandusky et al. (2004) analyzed the bug-fixing process. They focus their attention on the identification of the relationships existing among bug reports, but they do not examine in details the process itself. In contrast to the prior work, our article provides empirical evidence about coordination practices within FLOSS teams. Specifically, we describe the way the work of bug fixing is coordinated in these teams, how these practices differ from those of conventional software development and thus suggest what might be learned from FLOSS and applied in other settings.

We base our description on the work of Crowston (1997), who described the bug-fixing process observed at a commercial software company. Such a process is below defined as traditional because 1) it is carried out within a traditional kind of organization (i.e., the boundary are well defined, the environment is not distributed, the organization structure is defined) and 2) refers to the production of commercial rather than FLOSS software. The process is started by a customer who finds a problem when using a software system. The problem is reported (sometimes automatically or by the customer) to the company's response center. In the attempt to solve the problem, personnel in the center look in a database of known bugs. If a match is found, the fix is returned to the customer; otherwise, after identifying the affected product, the bug report is forwarded to an engineer in the marketing center. The assigned engineer tries to reproduce the problem and identify the cause (possibly requesting additional information from the reporter to do so). If the bug is real, the bug report is forwarded to the manager responsible for the module affected by the bug. The manager then assigns the bug to the software engineer responsible for that module. The software engineering diagnoses the problem (if she finds that the problem is in a different module, the report is forwarded to the right engineer) and designs a fix. The proposed fix is shared with other engineers responsible for modules that might be affected. When the feedback from those engineers is positive, the proposed design is transformed into lines of code. If changes in other module are needed, the software engineer also asks the responsible engineers for changes. The proposed fix is then tested, the eventual changed modules are sent to the integration manager. After approving, the integration manager recompiles the system, tests the entire system and releases the new software in the form of a patch. To summarize then, in the traditional bug-fixing process, the following tasks have been identified (Crowston, 1997):

Report, Try to solve the problem, Search database for solution, Forward to the marketing manager, Try to solve the problem/Diagnose the problem, Forward to the Software Engineering Group, Assign the bug, Diagnose the problem, Design the fix, Verify affected modules and ask for approval, Write the code for the fix, Test it, Integrate changes, Recompile the module and link it to the system.

After describing the above process, Crowston (1997) went on to analyze the coordination mechanisms employed. A number of the tasks listed can be seen as coordination mechanisms. For example, the search for duplicate bugs as well as the numerous forward and verify tasks manage some dependency. Searching for duplicate outputs is the coordination mechanism to manage a dependency between two tasks that might have the same output. In this case, the tasks are to respond to bug reports from customers. These tasks can be performed by diagnosing and repairing the bug, but if the solution to the bug report can be found in the database, then the effort taken to solve it a second time can be avoided. Thus, searching the database for a solution is a way to manage a potential dependency between the two bug-fixing tasks. Forwarding and verifying tasks are coordination mechanisms used to manage dependency between a task and the actor appropriate to perform that task. These steps are needed because many actors are involved in the process and each of them carry out a very specialized task, requiring additional work to find an appropriate person to perform each task.

RESEARCH METHODOLOGY

To address our research question, how are bug fixes coordinated in FLOSS projects, we carried out a multiple case study of different FLOSS projects, using the theoretical approach developed in the previous section. In this section, we discuss sample selection and data sources, data collection and data analysis, deferring a discussion of our findings to the following section.

Sample Section

In this sub-section we describe the basis for selecting projects for analysis. Projects to be studied were selected from those hosted on SourceForge, (http://sourceforge.net/), a Web-based system that currently supports the development of more than 100,000 FLOSS projects (although only a small proportion of these are actually active). We chose to examine projects from a single source to control for differences in available tools and project visibility. Because the process of manually reading, rereading, coding and recoding messages is extremely labor-intensive, we had to focus our attention on a small number of projects. We selected projects to study in-depth by employing a theoretical sampling strategy based on several practical and theoretical dimensions.

First, we chose projects for which data we need for our analysis are publicly available, meaning a large number of bug reports. (Not all projects use or allow public access to the bug-tracking system.) Second, we chose teams with more than 8 developers (i.e., those with write access to the source code control system), since smaller projects seemed less likely to experience significant coordination problems. The threshold of eight members was chosen based on our expectation that coordinating tasks within a team would become more complicated as the number of members increases. We assumed that each member of the team could manage 4 or 5 relationship, but with eight members, we expected some difficulty in coordination to arise.

Only 140 projects of SourceForge met the first two requirements in 2002 when we drew our sample. Third, projects were chosen so as to provide some comparison in the target audience and addressed topic, as discussed below. Finally, because we wanted to link coordination practices to project effectiveness, we tried to select more and less effective development teams. To this aim we used the definitions of effectiveness proposed by Crowston et al. (2006a), who suggest that a project is effective if it is active, the resulting software is downloaded and used and the team continues in operation. We selected 4 FLOSS projects to satisfy the mentioned criteria. Specifically, from the 140 large active projects, we selected two desktop chat clients that are aimed at end users (KICQ and Gaim) and two projects aimed primarily at developers (DynAPI, an HTML library and phpMyAdmin, a web-based database administration tool). A brief description of the projects is reported in Table 1, including the project goal, age at the time of the study, volume of communication and team membership. A consequence of the requirement of a significant number of bug reports is that all four projects are relatively advanced, making them representative of mature FLOSS projects. Based on the definition proposed by Crowston et al. (2006a), Kicq, Gaim and phpMyAdmin were chosen as examples of effective projects because they were active, the resulting software was being downloaded and the group had been active for a while. DynAPI was chosen as an example of a less effective project because the number of downloads and programming activity had rapidly decreased in the months leading up to the study.

Data Collection

In this sub-section we describe how data were selected and collected. As mentioned above, all of these projects are hosted on SourceForge, making certain kinds of data about them easily accessible for analysis. However, analysis of these data poses some ethical concerns that we had to address in gaining human subjects approval for our study. On the one hand, the interactions recorded are all public and developers have no expectations of privacy for their statements (indeed, the expectation is the opposite, that their comments will be widely broadcast). Consent is generally not required for studies of public behaviour. On the other hand, the data were not made available for research purposes but rather to support the work of the teams. We have gone ahead with our research after concluding that our analysis does not pose any likelihood of additional harm to the poster above the availability of the post to the group and in the archive available on the Internet.

We collected several kinds of data about each of the cases. First, we obtained data indicative of the effectiveness of each project, such as its level of activity, number of downloads and development status. Unfortunately, no documentation on the organization structure, task assignment procedures and coordination practices adopted was available on the projects' web sites (further supporting the position that these teams do not employ formal coordination methods). To get at the bug-fixing process, we considered alternative sources of data. Interviewing the developers might have provided information about their perceptions of the process, but would have required finding their identities, which was considered problematic given privacy concerns. Furthermore, reliance on self-reported data raises concerns about reliability of the data, the response rate and the likelihood that different developers would have different perceptions. While these issues are quite interesting to study (e.g., to understand how a team develops shared mental models of a project, e.g., Crowston & Kammerer, 1998), they seemed like distractions from our main research question. Because of these concerns, we elected to use objective data about the bug-fixing process. Hence, the main source of data about the bug-fixing process was obtained from the archives of the bug tracking system, which is the tool used to support the bug-fixing process (Herbsleb et al., 2001, p. 13). These data are particularly useful because they are unobtrusive measures of the team's behaviors (Webb & Weick, 1979) and thus provide an objective de-

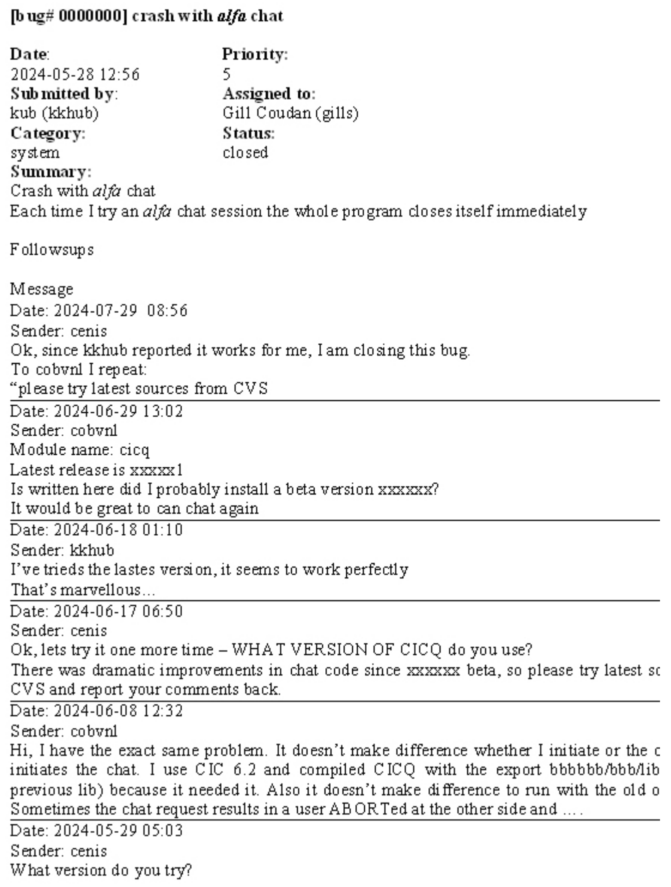
Table 1. Four examined projects

	KICQ	DynAPI	Gaim	phpMyAdmin
Goal	ICQ client for the KDE project (a chat client)	Dynamic HTML library	Multi-platform AIM client (a chat client)	Web-based database administration
Registration date	1999-11-19	2000-05-15	1999-11-13	2001-03-18
Development Status	4 Beta, 5 Production Stable	5 Production Stable	5 Production Stable	5 Production Stable
License	GPL	LGPL, GPL	GPL	GPL
Intended Audience	Developers, End Users/Desktop	Developers	Advanced End Users, Developers, End Users/ Desktop	Developers, End Users/Desktop, System Administrators
Topic	ICQ, K Desktop Environment (KDE)	Dynamic Content	AOL Instant Messenger, ICQ, Internet Relay Chat, MSN Messenger	Front-Ends, Dynamic Content, Systems Administration
Open bugs/ Total # of bugs	26 /88	45/220	269 /1499	29 /639
Open Support Requests/ Total # of requests	12/18		20/107	3/125
Open Patches/ Total # of Patches	1/8	14/144	75/556	7/131
Open Features requests/ Total # of requests	9/9	5/12	214/447	214/447
Mailing lists	813 messages in 3 mailing lists	9595 in 5 mailing lists	304 in 1 mailing list (developers)	5456 in 5 mailing lists
# of team members	9	11	9	9
Team member roles (# in role)	Admin/project manager (2); packager (1); developers (3); advisor/ mentor/ consultant(1); not specified (2)	Admin/project manager (1); developers (4); admin (3); not specified (3)	Project manager (1); admin/ developer (1); support manager (1); web designer (1); developers (3) not specified (2)	Project manager/ admin (1); admin/ developer (2); developers (6)

scription of the work that is actually undertaken, rather than perceptions of the work.

In the bug tracking system, each bug has a request ID, a summary (what the bug is about), a category (the kind of bug, e.g., system, interface), the name of the team member (or user) who submitted it, and the name of the team member it was assigned to. An example bug report in shown in Figure 1 (the example is fictitious). As well, individuals can post messages

Figure 1. Example bug report and followup messages



[bug# 0000000] crash with *alfa* chat

Date:	**Priority:**
2024-05-28 12:56	5
Submitted by:	**Assigned to:**
kub (kkhub)	Gill Coudan (gills)
Category:	**Status:**
system	closed

Summary:
Crash with *alfa* chat
Each time I try an *alfa* chat session the whole program closes itself immediately

Followsups

Message
Date: 2024-07-29 08:56
Sender: cenis
Ok, since kkhub reported it works for me, I am closing this bug.
To cobvnl I repeat:
"please try latest sources from CVS

Date: 2024-06-29 13:02
Sender: cobvnl
Module name: cicq
Latest release is xxxxx1
Is written here did I probably install a beta version xxxxxx?
It would be great to can chat again

Date: 2024-06-18 01:10
Sender: kkhub
I've trieds the lastes version, it seems to work perfectly
That's marvellous...

Date: 2024-06-17 06:50
Sender: cenis
Ok, lets try it one more time – WHAT VERSION OF CICQ do you use?
There was dramatic improvements in chat code since xxxxxx beta, so please try latest sc
CVS and report your comments back.

Date: 2024-06-08 12:32
Sender: cobvnl
Hi, I have the exact same problem. It doesn't make difference whether I initiate or the c
initiates the chat. I use CIC 6.2 and compiled CICQ with the export bbbbbb/bbb/lib
previous lib) because it needed it. Also it doesn't make difference to run with the old o
Sometimes the chat request results in a user ABORTed at the other side and ....

Date: 2024-05-29 05:03
Sender: cenis
What version do you try?

regarding the bug, such as further symptoms, requests for more information, etc. From this system, we extracted data about who submitted the bugs, who fixed them and the sequence of messages involved in the fix. By examining the name of the message senders, we can identify the project and community members who are involved in the bug-fixing process. Demographic information for the projects and developers and data from the bug tracking system were collected in the period 17–24 November 2002. We examined 31 closed bugs for Kicq, 95 closed bugs for DynAPI, 51 bugs for Gaim and 51 for PhPMyAdmin. The detailed text of the bug reports is not reported because of space restriction but is available on request.

Data Analysis

In this section we present our data analysis approach. For each of the bug reports, we carefully examined the text of the exchanged messages to identify the task carried out by each sender. We first applied the framework developed by Checkland & Scholes (1990), who suggested identifying the owners, customers and environment of the process, the actors who perform it, the transformation of inputs into outputs, the environment and the worldview that makes the process meaningful. We then followed the method described by Crowston & Osborn (2003), who suggested expanding the analysis of the transformation by identifying in more detail the activities carried out in the transformation. We identified the activities by inductively coding the text of the messages in the bug tracking systems of the four projects. We started by developing a coding scheme based on prior work on bug fixing (Crowston, 1997), which provided a template of expected activities needed for task assignment (those listed above). The coding system was then evolved through examination of the applicability of codes to particular examples. For example the message:

I've been getting this same error every FIRST time I load the dynapi in NS (win32). After reloading, it will work… loading/init problem?

represents a report submitted by another user (someone other than the person who initially identified and submitted the bug). This message was coded as "report similar problems". Table 2 shows the list of task types that were developed for the coding. The lowest level elementary task types were successively grouped into 6 main types of tasks, namely *Submit, Assign, Analyze, Fix, Test & Post,* and *Close*. A complete example of the coded version of a bug report (the one from Figure 1) is shown in Figure 2.

Once we had identified the process tasks, we studied in depth the bug-fixing process as carried out in the four cases. Specifically, we compared the sequence of tasks across different

Table 2. Coded tasks in the bug-fixing process

1.0.0 **Submit (S)**
1.1.0 Submit bug (code errors)
1.1.1 Submit symptoms
1.1.2 Provide code back trace (BT)
1.2.0 Submit problems
1.2.1 Submit incompatibility problems (NC)
2.0.0. **Assign (As)**
2.1.0 Bug self-assignment (A*)
2.2.0 Bug assignment (A)
3.0.0 **Analyze (An)**
3.1.0 Contribute to bug identification
3.1.1Report similar problems (R)
3.1.2 Share opinions about the bug (T)
3.2.0 Verify impossibility to fix the bug
3.2.1 Verify bug already fixed (AF)
3.2.2.Verify bug irreproducibility (NR)
3.2.3 Verify need for a not yet supported function (NS)
3.2.4 Verify identified bug as intentionally introduced (NCP)
3.3.0 Ask for more details
3.3.1 Ask for Code version/command line (V)
3.3.2 Ask for code back trace/examples (RBT/E)
3.4.0 Identify bug causes (G)

continued on following page

Table 2. continued

3.4.1 Identify and explain error (EE)
3.4.2 Identify and explain bug causes different from code (PNC)
4.0.0 Fix (F)
4.1.0 Propose temporary solutions (AC)
4.2.0 Provide problem solution (SP)
4.3.0 Provide debugging code (F)
5.0.0 **Test & Post (TP)**
5.1.0 Test/approve bug solution
5.1.1 Verify application correctness (W)
5.2.0 Post patches (PP)
5.3.0 Identify further problems with proposed patch (FNW)
6.0.0 **Close**
6.1.0 Close fixed bug/problem
6.2.0 Closed not fixed bug/problems
6.2.1 Close irreproducible bug (CNR) and close it
6.2.2 Close bug that asks for not yet supported function (CNS)
6.2.3 Close bug identified as intentionally introduced (CNCP)

Figure 2. Coded version of bug report in Figure 1

Bug ID	**Summary**	**Assigned to**	**Submitter**
0000000	crash with *alfa* chat	gills	kkhub

Task	**Person**	**Comments**
(S)	kkhub	
(V)	cenis	asks what version kkhub is running
(R)	cobvnl	reports the same problem as kkhub. submits information about the operating systems and the libraries
(V)	cenis	asks again what version both users are running
(W)	kkhub	reports the most recent version of cicq works
(TP&C)	cobvnl	reports version information and close the bug
(C)		bug closed

bugs to assess which sequences were most common and the role of coordination mechanisms in these sequences. We also examined which actors performed which tasks as well as looked for ways to more succinctly present the pattern of tasks, e.g., by presenting them as Markov processes. Because of the shortness and relative simplicity of our task sequences, we could exactly match task sequences, rather than having to statistically assess the closeness of matches to be able to form clusters (Sabherwal & Robey, 1995). Therefore, we were able to analyze the sequences by simple tabulation and counting, though more sophisticated techniques would be useful for larger scale data analysis. In the next Section we present the results of our analysis.

FINDINGS

In this section we present the findings from our analysis of the bug-fixing process in the four projects and the coordination mechanisms employed. Data about the percentage of submitted, assigned and fixed bugs both by team members and individuals external to the team for each project are reported in Table 3. Table 4 summarizes our findings regarding the nature of the bugs fixing process in the four projects.

We now present our overall analysis of the bug-fixing process. Each instance of a bug-fixing process starts (by definition) with a bug submission (S) and finishes with bug closing (C). Submitters may submit problems/symp-

Table 3. The bug-fixing process: Main results

	Kicq	DynAPI	Gaim	phpMyAdmin
Bugs submitted by team members	9.7%	21%	0%	21.6%
Bugs submitted by members external to the team	90.3%	78.9%	100%	78.4%
Bug assigned/self-assigned of which:	9.7%	0%	2%	1%
Assigned to team members	0%	-	100%	100%
Self assigned	66%			0%
Assigned to members external to the team	33%	-	-	0%
Bug fixed	51,6%	42,1%	51%	80%
Fixed by team members	81,3%	50%	84%	90,2%
Bug fixed by members external to the team	18,7%	50%	16%	9.8%

Table 4. Observed characteristics of the bug-fixing processes in the four projects

	Kicq	DynAPI	Gaim	phpMyAdmin
Min task sequence	3	2	2	2
Max task sequence	8	12	9	13
Uncommon tasks (count)	Bug assignment (3)	Bug assignment (0)	Bug assignment (0)	Bug assignment (1)
Community members	18	53	23	20
Team members' participation	2 of 9	6 of 11	3 of 9	4 of 10
Most active team members Role/ name	Project mgr: denis; Developer: davidvh	Admin: rainwater; Ext member: dcpascal	Admin-developer: warmenhoven; Developer: rob-flynn	Admin-developer: loic1; Admin-developer lem9.
Max posting by single community member	2	6	4	3
Not fixable bug closed	8	5	5	-

toms associated with bugs (Ss), incompatibility problems (NC) or/and also provide information about code back trace (BT). After submission, the team's project managers or administrators may assign the bug to someone to be fixed ((A); (A*) if they self-assign the bug). Other members of the community may report similar problems they encountered (R), discuss bug causes (T), identify bug causes (G) and/or verify the impossibility of fixing the bug. Participants often ask for more information to better understand the bug's causes (An). In most cases, but not always, after some discussion, a team member spontaneously decides to fix (F) the bug. Bug fixing may be followed by a test and the submission of a patch (TP). Testing is a coordination mechanism that manages usability between producing and using a patch, by ensuring that the patch is usable. However, as later explained, in the examined projects this type of activity is not often found. The bug is then closed (C). Bugs may also be closed because they cannot be fixed, e.g., if they are not reproducible (CNR), involve functions not supported yet (CNS) and/or are intentionally introduced to add new functionality in the future (CNCP). Notice that the closing activity is usually attributed to a particular user.

For our analysis, we consider *Submission*, *Analysis*, *Fix* and *Close* to be operative activities, while *Assignment*, *Test* and *Posting* are coordination mechanisms. As already discussed, *Assignment* is the coordination mechanisms used to manage the dependency between a task and the actor appropriate to perform it. *Posting* is the mechanisms used to manage the dependency between a task and its customers (it makes the fix available to the persons that need it).

The tasks identified above are linked by sequential dependencies as shown in Figure 3. These dependencies were identified by considering the logical connection between tasks based on the flow of resources. For example, a patch can not be tested before it is created. Because the dependencies can be satisfied in different orders, different sequences of the activities are possible. The tasks and their sequence change from bug to bug. Figure 3 shows the most frequent sequences observed, as identified by tabulating and counting the sequences.

Table 5 shows the portion of processes that follow each possible paths, based on the collected ways the bug-fixing process is observed to be performed within the FLOSS teams. For example, row 1 of Table 5 is read as follows. In the Dynapi project, submission always occurs as the first task (as it does for all of the groups, by definition), while the second task is S in 26% of cases, An in 39% of cases, F in 19% of cases, TP in 1% of cases and C in 15% of cases, and so on.

In Table 6, we describe the occurrences per task for the four projects and the average number of tasks to fix bugs. A χ^2 test shows a significant

Figure 3. Task dependencies in the bug-fixing process

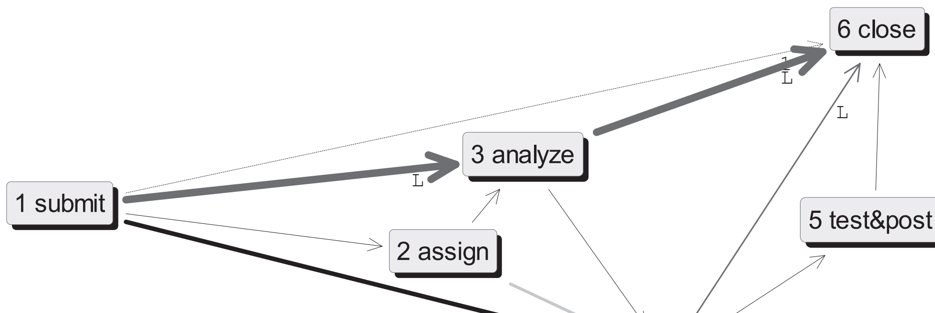


Table 5. Portion of processes for each possible path

i	task i-1	task i	Kicq	Dynapi	Gaim	PhPmyadmin
2	S	S	42%	26%	4%	2%
		As	6%	-	2%	2%
		An	39%	39%	61%	41%
		F	13%	19%	24%	45%
		TP	-	1%	2%	8%
		C	-	15%	8%	2%
3	S	An	38%	36%	50%	100%
		F	62%	40%	50%	-
		TP	-	8%	-	-
		C	-	16%	-	-
	As	An	-	-		100%
		F	50%	-	100%	-
		TP	50%	-	-	-
	An	S	8%	-	-	5%
		An	25%	41%	58%	52%
		F	8%	11%	3%	29%
		TP	-	-	3%	-
		C	58%	49%	35%	14%
	F	An	-	11%	-	13%
		F	50%	22%	8%	4%
		TP	-	6%	-	4%
		C	50%	61%	92%	78%
	TP	An	-	-	-	50%
		F	-	100%	100%	-%
		TP	-	-	-	-50%
		C	-	-	-	-
	C	An	-	7%	-	-
		C	-	93%	-	-
4	S	S	-	-	-	-
		An	100%	-	-	-
		F	-	-	-	100%
		TP	-	-	-	-
		C	-	-	-	-
	An	S	-	4%	5%	-
		An	13%	48%	53%	50%
		F	25%	11%	21%	11%

continued on following page

Table 5. continued

i	task i-1	task i	Kicq	Dynapi	Gaim	PhPmyadmin
		TP	-	4%	-	6%
		C	63%	33%	21%	33%
	F	S	-	-	-	-
		As	8%	-	-	-
		An		11%	20%	
		F	33%	16%	-	14%
		TP	-	5%	-	29%
		C	58%	68%	80%	57%
	TP	S	-	-	-	-
		An	-	-	-	-
		F	-	33%	-	33%
		TP	-	-	-	33%
		C	-	67%	100%	33%
	C	C	-	-	100%	-
5	S	AN	-	-	100%	-
		F	-	-	-	-
		TP	-	100%	-	-
	As	F	100%	-	-	-
	An	S	-	-	-	-
		An	50%	27%	73%	67%
		F	-	13%	18%	11%
		TP	-	-	-	11%
		C	50%	60%	9%	11%
	F	An	17%	14%	-	20%
		F	--	-	25%	-
		TP	-	-	25%	-
		C	83%	86%	50%	80%
	TP	An	-	-	-	-
		F	-	-	-	50%
		TP	-	100%	-	-
		C	-	-	-	50%
6	An	S	-		11%	-
		As	50%	-	-	14%
		An	-	20%	22%	43%
		F	-	-	11%	29%
		TP	-	20%	-	-
		C	50%	60%	56%	14%

continued on following page

Table 5. continued

i	task i-1	task i	Kicq	Dynapi	Gaim	PhPmyadmin
	F	S	-	-	-	-
		An	-	-	-	-
		F	-	-	-	-
		TP	-	-	-	33%
		C	100%	100%	-	67%
	TP	An	-	-	-	-
		F	-	100%	-	-
		TP	-	-	-	-
		C	-	-	-	100%
7	S	AN	-	-	50%	-
		©	-	-	50%	-
	As	F	100%	-	-	100%
	An	S	-	-	-	33%
		An	-			33%
		F	-	100%	100%	-
		TP	-	-	-	-
		C	-	-	-	33%
	F	An	-	100%	-	-
		F	-	-	-	-
		TP	-	-	-	-
		C	-	-	100%	100%
	TP	F	-	100%	-	100%
8	S	An	-	-	-	100%
		F	-	-	-	-
	An	An	-	100%	-	-
		F	-		100%	100%
	F	An	-	50%	-	-
		TP	-	-	-	50%
		C	100%	50%	100%	50%
9	An	An	-	50%	-	100%
		C	-	50%	-	-
	F	AN	-	-	-	100%
		C	-	-	100%	-
	TP	TP	-	-	-	100%
10	An	An	-	100%	-	50%
		F	-	-	-	50%

continued on following page

Table 5. continued

i	task i-1	task i	Kicq	Dynapi	Gaim	PhPmyadmin
	TP		-	-	-	100%
11	An	An	-	100%		50%
		F	-	-	-	50%
	F	C	-	-	-	100%
12	An	An	-	-	-	100%
		C	-	100%	-	-
	F	C	-	-	-	100%
13	An	C	-	100%	-	100%

Table 6. Task occurrences and average number of tasks per projects

Task / Project (bugs)	(S)	(Ag)	(An)	(F)	(TP)	(C)	Avr. tasks per bug
KICQ (31)	44	4	24	23	0	31	4.4
Dynapi (95)	121	0	94	54	9	95	3.8
Gaim (51)	71	1	77	28	4	51	4.2
Phpmyadmin (51)	54	2	66	45	15	51	4.6

difference in the distribution of task types across projects ($p<0.001$). On all projects, *submit* is the task that always appears first, while *analyze* is the most common second task and *fix,* third. The first three most frequent task sequences are reported in Table 7. As noted above, given the limited number of examined sequences, the sequences were manually identified. Finally, in Table 8 we show which tasks are carried out by which roles. Please notice that differences in percentage shown in Table 3 and Table 8 are due to the fact that results reported in Table 8 are calculated based on the total number of tasks carried out per bug. For example, in Table 3 the considered submissions are those carried out only as first task. In Table 8 all submissions tasks (i.e. also those carried out as second, third etc. task) are considered. As reported in Table 2, submissions tasks can be more than one per bug because submissions can occur also in the form of a *submit* sub-task. The same stands for the fixing tasks. In Table 3 only the final fixing tasks are considered.

A detailed description of the process as performed in the four cases is provided below considering both the sequence of tasks and the participation in the bug-fixing process.

Kicq

The minimal sequence is composed of three tasks, the longest by eight. Bug fixing is usually the second task in the sequence, meaning that it is most common for bugs to be fixed immediately after they are submitted, which is different from the overall picture in which analysis was most common. *Bug assignment* is a quite rare task, as only three bugs are formally assigned. Eight bugs were closed because they were considered to be not fixable.

There are 18 identified users, but many (anonymous) users submitted bugs and contributed to analysis and fixing. Team members are not very active in bug fixing, except for one of the two project managers (denis), who is involved in all the tasks and, in particular, in

Table 7. Most frequent task sequences

	First task	Second task	Third task	Fourth Task	Occurrences
Kicq	S	An	C	-	13
	S	F	C	-	11
	S	An	F	C	2
DynAPI	S	An	C	-	34
	S	F	C	-	24
	S	C	-	-	17
Gaim	S	An	C	-	21
	S	F	C	-	13
	S	An	F	C	6
phpMy-Admin	S	F	C	-	19
	S	An	C	-	8
	S	An	F	C	7
All proj-ects	S	An	C	-	76
	S	F	C	-	67
	S	C	-	-	22

Table 8. Tasks carried out by different roles

task	**ROLES/PROJECT Kick**				
	devel	pm			% of total tasks
S		4			9%
As		4			100%
An		18			75%
F	1	15			70%
TP					
total	2	49			
	Dynapi				
	devel	admin	admin/develop	no role	% of total tasks
S	9	6	1	10	21%
As					
An		27		3	32%
F		18	1	2	35%
TP		2	1		33%
total	9	53	3	15	
	Gaim				
	admin/develop	develop	supp. mang.		% of total tasks
S					0%
As		1			100%
An	33	11	1		58%

continued on following page

Table 8. continued

F	17	6			82%
TP					100%
total	52	19	1		
	Phpmyadmin				
	admin/develop	pm			% of total tasks
S	11	1			22%
As	2				100%
An	49				74%
F	40				89%
TP	10				93%
total	115	1			

bug analysis and fixing. Out of 23 fixed bugs, 16 are fixed by denis. Apart from a developer (davidvh), the other project members seem not take part in the bug-fixing process at all. However, it is noteworthy that the bug tracking system register three bugs as submitted and assigned to the administrator (bill), although he does not otherwise take part in the process. Most of the community members have posted just one bug, and only two of them posted 2 bugs each.

Dynapi

The minimal sequence is composed of two tasks, the longest by 12. Again, *bug assignment* is not explicitly carried out; apparently community or team members decide autonomously to take part to the bug-fixing process. However, the system reports that six bugs (out of 95) are assigned to an administrator and the rest to a member external to the team. Five bugs are closed because they are said to be not fixable. Bug fixing is usually the second or the third task in the sequence.

Team members are not very active except for an administrator (rainwater), who is involved in all the tasks and, in particular, in bug *analysis* and *fixing*. The other five team members (two without a specific role, one administrator/developer, one developer and one administrator) are mostly involved in bug *fixing*. The community members involved in the process are 47 persons plus some anonymous posts. Most of them submitted just one bug, but some submitted more (e.g., one submitted six bugs). Community members are mostly involved in bug *submission* but some also carry out other tasks. In particular, one of them (dcpascal) is very active in all the process tasks. Out of 57 fixed bugs, 20 are fixed by a team member (the project manager).

Gaim

The minimal sequence is composed of two tasks, the longest by nine. *Bug assignment* is not explicitly carried out, as community or team members decide autonomously to take part to the bug-fixing process. However, the system reports that 24 bugs (out of 51) are assigned to an administrator (and the rest to member external to the team). Five bugs are directly closed because they are said to be not fixable.

Team members are not very active in bug fixing except for the administer/developer (warmenhoven) and a developer (robflynn), who are involved in many tasks and, in particular, in bug analysis and fixing. Apart from them, just another member of the project team, a developer (lschiere), is also involved in the bug fixing. The community members involved in the process are 21 persons plus some anonymous users. Most of them posted just one bug (2 of them posted five bugs, one 4 bugs). Some of them are also involved in bug analysis and fixing. Out of 29 fixed bugs, 23 are fixed by a team member (the project manager).

Phpmyadmin

The minimal sequence is composed of two tasks, the longest by thirteen. *Bug assignment* is a quite rare task, as only one bug is formally assigned. The assignment is carried out by an administrator/developer (lem9) and directed to a team member (loic1). However, the system reports that all 51 are assigned, of which 40 to team members. Bug fixing is usually the second or the third task.

Team members are not very active in the process, except for two administer/developers (loic1 and lem9), who are involved in all the tasks and, in particular, in bug analysis and fixing (but also submission). Apart from them, two team members take part to the process, a project manager/adminster (swix) and a developer (robbat2), that are involved (not heavily) in bug submission and analysis. The community is composed of 16 members plus some anonymous users. Most of them have just posted one bug (two of them posted 3 bugs), but some are also involved in bug analysis and fixing. Out of 49 fixed bugs, 44 are fixed by team member (administrator/developers).

DISCUSSION

In this section, we discuss the implications of our findings for understanding the coordination of bug fixing in FLOSS teams. Our findings provide some interesting insights on the bug-fixing process for FLOSS development in these teams. First, process sequences are on average quite short (four tasks) and they seem to be quite similar: submit, (analyze), fix and close. As shown in Table 3, formal task assignments are quite uncommon: only few bugs are formally assigned. Coordination seems rather to spontaneously emerge. From bug description and initial analysis, those who have the competencies autonomously decide to fix the bug and simply go ahead and do so. That activity is facilitated by the supplied bug report and analysis, which is often undertaken by several contributors. Apart from the procedure to submit bugs (we analyzed only bugs submitted through the bug tracking system), we do not observe any other formal process: roles are not predefined, delivery dates are not assigned nor are formal-interpersonal, formal-impersonal or informal-interpersonal procedures adopted. The lack of assignment is one of main aspects differentiating the process as it occurs in FLOSS development team from the traditional commercial bug-fixing process described above.

Testing is also quite an uncommon task in the data. Most of the proposed fixes are directly posted, though presumably after personal testing that is not documented. If no one describes the emergence of new problems with these fixes, they are automatically posted and the relevant bug closed without a formal test process. It is important also to note that many of the posted problems do not represent real bugs (i.e. they have been already fixed, are not reproducible, have been intentionally produced, are associated to functions not yet supported or are associated to related programs), so they are directly closed with that explanation.

Another striking finding is that the bug-fixing process is apparently carried out without any explicit discussion about where knowledge is located in the team, contrary to the findings of Faraj and Sproull (2000), who stress the importance of expertise coordination for team effectiveness (they distinguish expertise coordination from what they call administrative coordination, which is the focus of this article). They define expertise coordination as the management of knowledge and skill dependencies. To manage knowledge it is necessary to know where it is located within development team, where it is needed and how to access it. However, in our observations, the knowledge needs seem to emerge by "(informal and asynchronous) electronic meetings".

The bug tracking system represents a sort of organizational memory, storing bug reports and solutions found to submitted problems (which not always are real bugs). However, as discussed in Cubranic (1999), the large number of emails stored makes it difficult for contributors to easily identify the solutions to their own problems, so making different users repeat the same (already fixed or addressed) submission more times. In those cases (i.e., for bugs closed without being fixed or the attended patches posted), it is usually the team members that act as "memory".

A further difference is that in these projects, the process is performed by few team members (usually not more that two or three) working with a member of the larger community. Team members (usually project managers, administrators or developers) are most involved in bug fixing, testing and posting. Surprisingly, only a few members of the team are involved in the process. The other participants are active users who submit bugs or contribute to their analysis. We also noted striking differences in the level of contribution to the process. The most active users in the projects carried out most of the tasks while most others contributed only once or twice. Most community members submit only one bug; only two or three members of the involved community are involved in fixing tasks and can be referred to as co-developers. As expected, the most widely dispersed type of action was submitting a bug, while diagnosis and bug-fixing activities were concentrated among a few individuals.

As we have few members of the team and few members of the community (co-developers) mostly involved in bug fixing and many users/members of the community (active users) mostly involved in bug submission, the organizational models proposed in the literature (Cox, 1998) seem to be valid for the bug-fixing process. It would be interesting to further investigate if those, among the active users also involved in bug fixing also contribute to software coding, e.g., by analysis of contributions of source code independent of bug fixes.

As an apparently less effective project, we expected to find that DynAPI had a smaller active user base than the other projects. However, as noted above, our data shows the opposite. However, our estimation of the effectiveness of the projects is based on activity levels. It appears that DynAPI somehow does not benefit from its larger community in increased activity. One striking difference is the proportion of bugs fixed by the team members, shown in Table 3, which is much lower in DynAPI than in the other projects. This finding suggests that the contribution of core members may be particularly important in the effectiveness of the team. The case studies presented here are not sufficient to test this hypothesis, so it is one that should be followed up in future studies.

CONCLUSION

In this article, we investigated the coordination practices adopted within four FLOSS development teams. In particular, we analyzed the bug-fixing process, which is considered central to the effectiveness of the FLOSS process. The article provided some interesting results. The task sequences we observed were mostly sequential and composed of few steps, namely *submit, fix and close*. Second, our data supports the observation that FLOSS processes seem to lack traditional coordination mechanisms such as task assignment. Third, effort is not equally distributed among process actors. A few contribute heavily to all tasks, while the majority just submit one or two bugs. As a result, the organization structure reflected in the process resembles the one proposed in the literature for the FLOSS development process. Few actors (core developers), usually team project managers or administrators, are mostly involved in bug fixing. Most of the involved actors are active users instead of developers, who just submit bug reports. In between are few actors, external to the team, who submit bugs and contribute to fixing them. Finally, while we did not find obvious associations between coordination practices and project effectiveness, we did notice a link to participation: our least effective team also had the lowest level of participation from core developers, suggesting their importance, even given the more widely distributed participation possible.

The article contributes to fill a gap in the literature by providing a picture of the coordination practices adopted within FLOSS development team. Besides, the article proposes an innovative research methodology (for the analysis of coordination practices FLOSS development teams) based on the collection of process data by electronic archives, the codification of message texts, and the analysis of codified information supported by the coordination theory.

Based on the analysis of task carried out and the attendant coordination mechanisms, we argue that the bazaar metaphor proposed by (Raymond, 1998) to describe the FLOSS organization structure is still valid for the bug-fixing process. As in a bazaar, the actors involved in the process autonomously decide the schedule and contribution modes for bug fixing, making a central coordination action superfluous.

As with all research, the current article has some limitations that limit the scope of our current conclusions and suggests directions for further research. First, although the selected projects are quite different in terms of target audience and topic, other characteristics (not examined because they are not explicitly present on the project web sites) could be shared among projects so affecting the obtained results. In the future, we would like to deepen our knowledge about the coordination practices adopted by the four projects by directly interviewing some of the involved actors. Second, due to the limited number of examined bugs, the process sequences have been manually examined. In the future, we intend to enlarge the number of examined bugs and adopt automatic techniques (e.g. the optimal matching technique) to analyze and classify the task sequences. In particular, we plan to further explore the hypothesis about the importance of core group members by examining a larger number of projects (e.g., to examine the change in the population over time). Finally, in the article we only examined administrative coordination. In the future, we intend to examine also expertise coordination in more detail. A particular interesting consideration here is the development of shared mental models that might support the coordination of the teams' processes.

REFERENCE

Ahuja, M. K., Carley, K., & Galletta, D. F. (1997). *Individual performance in distributed design groups: An empirical study.* Paper presented at the SIGCPR Conference, San Francisco.

Alho, K., & Sulonen, R. (1998). *Supporting virtual software projects on the Web.* Paper presented at the Workshop on Coordinating Distributed Software Development Projects, 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98).

Anthes, G. H. (2000, June 26). Software Development goes Global. *Computerworld Magazine*.

Bandow, D. (1997). Geographically distributed work groups and IT: A case study of working relationships and IS professionals. In *Proceedings of the SIGCPR Conference* (pp. 87–92).

Bélanger, F. (1998). Telecommuters and Work Groups: A Communication Network Analysis. In *Proceedings of the International Conference on Information Systems (ICIS)* (pp. 365–369). Helsinki, Finland.

Bessen, J. (2002). *Open Source Software: Free Provision of Complex Public Goods*: Research on Innovation.

Bezroukov, N. (1999a). A second look at the Cathedral and the Bazaar. *First Monday, 4*(12).

Bezroukov, N. (1999b). Open source software development as a special type of academic research (critique of vulgar raymondism). *First Monday, 4*(10).

Boulding, K. E. (1956). General systems theory—The skeleton of a science. *Management Science, 2*(April), 197–208.

Britton, L. C., Wright, M., & Ball, D. F. (2000). The use of co-ordination theory to improve service quality in executive search. *Service Industries Journal, 20*(4), 85–102.

Brooks, F. P., Jr. (1975). *The Mythical Man-month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Butler, B., Sproull, L., Kiesler, S., & Kraut, R. (2002). Community effort in online groups: Who does the work and why? In S. Weisband & L. Atwater (Eds.), *Leadership at a Distance*. Mahwah, NJ: Lawrence Erlbaum.

Carmel, E. (1999). *Global Software Teams*. Upper Saddle River, NJ: Prentice-Hall.

Carmel, E., & Agarwal, R. (2001). Tactical approaches for alleviating distance in global software development. *IEEE Software*(March/April), 22–29.

Checkland, P. B., & Scholes, J. (1990). *Soft Systems Methodology in Action*. Chichester: Wiley.

Conway, M. E. (1968). How do committees invent. *Datamation, 14*(4), 28–31.

Cox, A. (1998). Cathedrals, Bazaars and the Town Council. Retrieved 22 March, 2004, from http://slashdot.org/features/98/10/13/1423253.shtml

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

Crowston, K., & Howison, J. (2006). Hierarchy and centralization in Free and Open Source Software team communications. *Knowledge, Technology & Policy, 18*(4), 65–85.

Crowston, K., Howison, J., & Annabi, H. (2006a). Information systems success in Free and Open Source Software development: Theory and measures. *Software Process—Improvement and Practice, 11*(2), 123–148.

Crowston, K., & Kammerer, E. (1998). Coordination and collective mind in software requirements development. *IBM Systems Journal, 37*(2), 227–245.

Crowston, K., & Osborn, C. S. (2003). A coordination theory approach to process description and redesign. In T. W. Malone, K. Crowston & G. Herman (Eds.), *Organizing Business Knowledge: The MIT Process Handbook*. Cambridge, MA: MIT Press.

Crowston K., Scozzi B., (2003). Open Source Software projects as virtual organizations: competency rallying for software development. *IEE Proceedings Software*, 149(1), 3-17.

Crowston, K., Wei, K., Li, Q., Eseryel, U. Y., & Howison, J. (2005). *Coordination of Free/Libre Open Source Software development.* Paper presented at the International Conference on Information Systems (ICIS 2005), Las Vegas, NV, USA.

Crowston, K., Wei, K., Li, Q., & Howison, J. (2006b). *Core and periphery in Free/Libre and Open Source software team communications.* Paper presented at the Hawai'i International Conference on System System (HICSS-39), Kaua'i, Hawai'i.

Cubranic, D. (1999). *Open-source software development.* Paper presented at the 2nd Workshop on Software Engineering over the Internet, Los Angeles.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM, 31*(11), 1268–1287.

Curtis, B., Walz, D., & Elam, J. J. (1990). Studying the process of software design teams. In *Proceedings of the 5th International Software Process Workshop On Experience With Software Process Models* (pp. 52–53). Kennebunkport, Maine, United States.

Cutosksy, M. R., Tenenbaum, J. M., & Glicksman, J. (1996). Madefast: Collaborative engineering over the Internet. *Communications of the ACM, 39*(9), 78–87.

de Souza, P. S. (1993). *Asynchronous Organizations for Multi-Algorithm Problems.* Unpublished Doctoral Thesis, Carnegie-Mellon University.

DeSanctis, G., & Jackson, B. M. (1994). Coordination of information technology management: Team-based structures and computer-based communication systems. *Journal of Management Information Systems, 10*(4), 85.

Di Bona, C., Ockman, S., & Stone, M. (Eds.). (1999). *Open Sources: Voices from the Open Source Revolution*. Sebastopol, CA: O'Reilly & Associates.

Drucker, P. (1988). The Coming of the New Organization. *Harvard Business Review*(3–15).

Faraj, S., & Sproull, L. (2000). Coordinating Expertise in Software Development Teams. *Management Science, 46*(12), 1554–1568.

Finholt, T., Sproull, L., & Kiesler, S. (1990). Communication and Performance in Ad Hoc Task Groups. In J. Galegher, R. F. Kraut & C. Egido (Eds.), *Intellectual Teamwork*. Hillsdale, NJ: Lawrence Erlbaum and Associates.

Franck, E., & Jungwirth, C. (2002). *Reconciling investors and donators: The governance structure of open source* (Working Paper No. No. 8): Lehrstuhl für Unternehmensführung und -politik, Universität Zürich.

Gacek, C., & Arief, B. (2004). The many meanings of Open Source. *IEEE Software, 21*(1), 34–40.

Galbraith, J. R. (1973). *Designing Complex Organizations*. Reading, MA: Addison-Wesley.

Grabowski, M., & Roberts, K. H. (1999). Risk mitigation in virtual organizations. *Organization Science, 10*(6), 704–721.

Grinter, R. E., Herbsleb, J. D., & Perry, D. E. (1999). The Geography of Coordination: Dealing with Distance in R&D Work. In *Proceedings of the GROUP '99 Conference* (pp. 306–315). Phoenix, Arizona, US.

Hallen, J., Hammarqvist, A., Juhlin, F., & Chrigstrom, A. (1999). Linux in the workplace. *IEEE Software, 16*(1), 52–57.

Hann, I.-H., Roberts, J., Slaughter, S., & Fielding, R. (2002). Economic incentives for participating in open source software projects. In *Proceedings of the Twenty-Third International Conference on Information Systems* (pp. 365–372).

Herbsleb, J. D., & Grinter, R. E. (1999a). Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*(September/October), 63–70.

Herbsleb, J. D., & Grinter, R. E. (1999b). *Splitting the organization and integrating the code: Conway's law revisited.* Paper presented at the Proceedings of the International Conference on Software Engineering (ICSE '99), Los Angeles, CA.

Herbsleb, J. D., Mockus, A., Finholt, T. A., & Grinter, R. E. (2001). *An empirical study of global software development: Distance and speed.* Paper presented at the Proceedings of the International Conference on Software Engineering (ICSE 2001), Toronto, Canada.

Hertel, G., Niedner, S., & Herrmann, S. (2003). Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel. *Research Policy, 32*(7), 1159–1177.

Humphrey, W. S. (2000). *Introduction to Team Software Process*: Addison-Wesley.

Iannacci, F. (2005). Coordination processes in OSS development: The Linux case study. Retrieved 21 September, 2006, from http://opensource.mit.edu/papers/iannacci3.pdf

Jarvenpaa, S. L., & Leidner, D. E. (1999). Communication and trust in global virtual teams. *Organization Science, 10*(6), 791–815.

Jensen, C., & Scacchi, W. (2005). Collaboration, Leadership, Control, and Conflict Negotiation in the Netbeans.org Open Source Software Development Community. In *Proceedings of the Hawai'i International Conference on System Science (HICSS 2005)*. Big Island, Hawai'i.

Kaplan, B. (1991). Models of change and information systems research. In H.-E. Nissen, H. K. Klein & R. Hirschheim (Eds.), *Information Systems Research: Contemporary Approaches and Emergent Traditions* (pp. 593–611). Amsterdam: Elsevier Science Publishers.

Kogut, B., & Metiu, A. (2001). Open-source software development and distributed innovation. *Oxford Review of Economic Policy, 17*(2), 248–264.

Kraut, R. E., Steinfield, C., Chan, A. P., Butler, B., & Hoag, A. (1999). Coordination and virtualization: The role of electronic networks and personal relationships. *Organization Science, 10*(6), 722–740.

Kraut, R. E., & Streeter, L. A. (1995). Coordination in software development. *Communications of the ACM, 38*(3), 69–81.

Krishnamurthy, S. (2002). Cave or Community? An Empirical Examination of 100 Mature Open Source Projects. *First Monday, 7*(6).

Lawrence, P., & Lorsch, J. (1967). *Organization and Environment*. Boston, MA: Division of Research, Harvard Business School.

Leibovitch, E. (1999). The business case for Linux. *IEEE Software, 16*(1), 40–44.

Lerner, J., & Tirole, J. (2001). The open source movement: Key research questions. *European Economic Review, 45*, 819–826.

Madanmohan, T. R., & Navelkar, S. (2002). *Roles and Knowledge Management in Online Technology Communities: An Ethnography Study* (Working paper No. 192): IIMB.

Malone, T. W., & Crowston, K. (1994). The interdisciplinary study of coordination. *Computing Surveys, 26*(1), 87–119.

Markus, M. L., Manville, B., & Agres, E. C. (2000). What makes a virtual organization work? *Sloan Management Review, 42*(1), 13–26.

Markus, M. L., & Robey, D. (1988). Information technology and organizational change: Causal structure in theory and research. *Management Science, 34*(5), 583–598.

Massey, A. P., Hung, Y.-T. C., Montoya-Weiss, M., & Ramesh, V. (2001). When culture and style aren't about clothes: Perceptions of task-technology "fit" in global virtual teams. In *Proceedings of GROUP '01*. Boulder, CO, USA.

McCann, J. E., & Ferry, D. L. (1979). An approach for assessing and managing inter-unit interdependence. *Academy of Management Review, 4*(1), 113–119.

Metiu, A., & Kogut, B. (2001). *Distributed Knowledge and the Global Organization of Software Development* (Working paper). Philadelphia, PA: The Wharton School, University of Pennsylvania.

Mintzberg, H. (1979). *The Structuring of Organizations*. Englewood Cliffs, NJ: Prentice-Hall.

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies Of Open Source Software development: Apache And Mozilla. *ACM Transactions on Software Engineering and Methodology, 11*(3), 309–346.

Mohr, L. B. (1971). Organizational technology and organizational structure. *16*, 444–459.

Mohr, L. B. (1982). *Explaining Organizational Behavior: The Limits and Possibilities of Theory and Research*. San Francisco: Jossey-Bass.

Moon, J. Y., & Sproull, L. (2000). Essence of distributed work: The case of Linux kernel. *First Monday, 5*(11).

Nejmeh, B. A. (1994). Internet: A strategic tool for the software enterprise. *Communications of the ACM, 37*(11), 23–27.

O'Leary, M., Orlikowski, W. J., & Yates, J. (2002). Distributed work over the centuries: Trust and control in the Hudson's Bay Company, 1670–1826. In P. Hinds & S. Kiesler (Eds.), *Distributed Work* (pp. 27–54). Cambridge, MA: MIT Press.

Orlikowski, W. J. (2002). Knowing in practice: Enacting a collective capability in distributed organizing. *Organization Science, 13*(3), 249–273.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(2), 1053–1058.

Pfaff, B. (1998). Society and open source: Why open source software is better for society than proprietary closed source software. from http://www.msu.edu/user/pfaffben/writings/anp/oss-is-better.html

Pfeffer, J. (1978). *Organizational Design*. Arlington Heights, IL: Harlan Davidson.

Pfeffer, J., & Salancik, G. R. (1978). *The External Control of Organizations: A Resource Dependency Perspective*. New York: Harper & Row.

Prasad, G. C. (n.d.). A hard look at Linux's claimed strengths…. from http://www.osopinion.com/Opinions/GaneshCPrasad/GaneshCPrasad2-2.html

Raymond, E. S. (1998). The cathedral and the bazaar. *First Monday, 3*(3).

Robey, D., Khoo, H. M., & Powers, C. (2000). Situated-learning in cross-functional virtual teams. *IEEE Transactions on Professional Communication*(Feb/Mar), 51–66.

Sabherwal, R., & Robey, D. (1995). Reconciling variance and process strategies for studying information system development. *Information Systems Research, 6*(4), 303–327.

Sandusky, R. J., Gasser, L., & Ripoche, G. (2004). *Bug Report Networks: Varieties, Strategies, and Impacts in an OSS Development Community.* Paper presented at the Proceedings of the ICSE Workshop on Mining Software Repositories, Edinburgh, Scotland, UK.

Sawyer, S., & Guinan, P. J. (1998). Software development: Processes and performance. *IBM Systems Journal, 37*(4), 552–568.

Scacchi, W. (1991). The software infrastructure for a distributed software factory. *Software Engineering Journal, 6*(5), 355–369.

Scacchi, W. (2002). Understanding the requirements for developing Open Source Software systems. *IEE Proceedings Software, 149*(1), 24–39.

Scacchi, W. (2005). Socio-technical interaction networks in Free/Open Source Software development processes. In S. T. Acuña & N. Juristo (Eds.), *Software Process Modeling* (pp. 1–27). New York: Springer.

Stewart, K. J., & Ammeter, T. (2002). An exploratory study of factors influencing the level of vitality and popularity of open source projects. In *Proceedings of the Twenty-Third International Conference on Information Systems* (pp. 853–857).

Taylor, P. (1998, December 2). New IT mantra attracts a host of devotees. *Financial Times, Survey—Indian Information Technology,* p. 1.

Thompson, J. D. (1967). *Organizations in Action: Social Science Bases of Administrative Theory*. New York: McGraw-Hill.

Torvalds, L. (1999). The Linux edge. *Communications of the ACM, 42*(4), 38–39.

Valloppillil, V. (1998). Halloween I: Open Source Software. from http://www.opensource.org/halloween/halloween1.html

Valloppillil, V., & Cohen, J. (1998). Halloween II: Linux OS Competitive Analysis. from http://www.opensource.org/halloween/halloween2.html

Victor, B., & Blackburn, R. S. (1987). Interdependence: An alternative conceptualization. *Academy of Management Review, 12*(3), 486–498.

Walz, D. B., Elam, J. J., & Curtis, B. (1993). Inside a software design team: knowledge acquisition, sharing, and integration. *Communications of the ACM, 36*(10), 63–77.

Watson-Manheim, M. B., Chudoba, K. M., & Crowston, K. (2002). Discontinuities and continuities: A new way to understand virtual work. *Information, Technology and People, 15*(3), 191–209.

Wayner, P. (2000). *Free For All*. New York: HarperCollins.

Webb, E., & Weick, K. E. (1979). Unobtrusive measures in organizational theory: A reminder. *Administrative Science Quarterly, 24*(4), 650–659.

Weber, S. (2004). *The Success of Open Source*. Cambridge, MA: Harvard.

Weisband, S. (2002). Maintaining awareness in distributed team collaboration: Implications for leadership and performance. In P. Hinds & S. Kiesler (Eds.), *Distributed Work* (pp. 311–333). Cambridge, MA: MIT Press.

Zuboff, S. (1988). *In the Age of the Smart Machine*. New York: Basic Books.

ENDNOTE

[1] This research was partially supported by US NSF Grants 03-41475, 04–14468 and 05-27457. An earlier version of this article was presented at the *First International Workshop on Computer Supported Activity Coordination (CSAC 2004)*. The authors thank previous anonymous reviewers of the article for their comments that have helped to improve the article.

Kevin Crowston joined the School of Information Studies at Syracuse University in 1996. He received his PhD in information technologies from the Sloan School of Management, Massachusetts Institute of Technology (MIT) in 1991. Before moving to Syracuse he was a founding member of the Collaboratory for Research on Electronic Work at the University of Michigan and of the Centre for Coordination Science at MIT. His current research focuses on new ways of organizing made possible by the extensive use of information technology.

Barbara Scozzi is an assistant professor at the Politecnico of Bari, Italy. She received her PhD in management engineering from the University of Rome Tor Vergata/Polytechnic of Bari in 2001. Since 1997 she has been involved in many research projects at the Politecnico di Bari. Her main research interests are coordination, knowledge management and innovation in business organizations.