Cognitive Alignment through Artefacts in Distributed Innovation:

The Role of Initial Code Release in Open Source Software[1]

Abstract

This paper casts light on the role of artifacts for cognitive alignment of creative workers in distributed product development projects. Drawing on the research on system integration, modularity and artifacts as coordination devices, we develop two hypotheses stating that (1) the provision of an initial version of the artifact to be jointly developed fosters the cognitive alignment of distributed innovators, and that (2) such alignment increases the probability that a project effectively produces an outcome. We test both hypotheses studying a sample of 5703 open source projects hosted on SourceForge during 2005 and 2006, and find that the provision of some initial code fosters cognitive alignment of programmers, and that this in turn increases the probability of observing the release of a new version of the program later on.

[1] Please, quote as: Becker M., Rullani F., Zirpoli F., "Cognitive Alignment through Artefacts in Distributed Innovation: The Role of Initial Code Release in Open Source Software", *Academy of Management (AOM) 2013 Annual Meeting Conference*, August 9-13, 2013 - Lake Buena Vista (Orlando), FL.

1. Introduction

The development of complex products, such as automobiles, aircraft or software, typically takes place in networks that draw on multiple actors such as suppliers, customers, or universities (von Hippel, 1988; Womack *et al.,* 1990; Powell et al., 1996, Chesbrough, 2003; David and Foray, 2003). One of the key challenges in developing complex products is how to coordinate the involvement of many actors in the new product development process so as to achieve the desired outcome. This challenge can be massive in developing highly complex products such as cars (e.g., Zirpoli & Becker, 2011), aircraft (Brusoni et al., 2001) or software, especially open source software (e.g., Dahlander et al., 2008).

Prior research on distributed innovation in different industries that face difficult coordination challenges has identified the important role of factors such as artefacts, informal relations, and trust (Star & Griesemer, 1989; Carlile, 2002; Ouchi, 1980). In this paper, we consider open source software (OSS) development, which requires coordinating many independent developers who work in a self-organized fashion. This setting is particularly interesting for our aim, as self-organization implies the need to coordinate without direct supervision and authority. Research in this industry has identified the crucial role of coordination mechanisms such as modularity (MacCormak et al., 2006; Giuri et al, 2010), leadership (e.g., Lerner and Tirole, 2002), legitimacy (O'Mahony and Ferraro, 2007), and virtual artefacts such as code and online conversations (Lanzara and Morner, 2005; Kuk, 2006). One of the strong empirical regularities concerning the organization of software projects is that at the beginning of many successful OSS projects, there was an initial code release acting as an enabler for further development (e.g. Lerner and Tirole, 2002). While prior research on OSS has documented this fact providing different explanations (e.g. Haefliger et al., 2008), it is not entirely clear yet what exactly is the role of such initial code

for coordination, if any. So far, this issue has remained implicit and no theory has been tested. Investigating this issue is of interest for understanding how to foster coordination of distributed product development efforts in general, and especially in cases of highly complex products developed in a self-organized fashion.

In this paper we theorize on the role of initial code in OSS as a coordination device operating through cognitive alignment, i.e., guiding the participants in the distributed innovation process towards aligning their visions of the final product to that of the initial code provider. In this way, the initial code provider guides them to maintain the development trajectories of the modules they work on within a limited (even if still large) set of potential visions of the final product. Thanks to the provision of initial code, thus, it is possible to compensate the difficulties, due to self-organization, in explicitly setting a full-fledged architecture at the beginning of the project: the artifact itself will contribute to the needed alignment also without such architecture. As a consequence, we should observe more effective coordination than without provision of an initial code, and the product should be released on the market with a higher probability.

We test the previous hypotheses on a dataset of 5703 open source software projects registered on SourceForge during a two-year period. We find that initial code release is indeed associated with increased cognitive alignment of developers, and a higher chance that software development projects will actually release further code subsequently. This result adds to previous literature on coordination in distributed innovation and project development by improving understanding of how artifacts contribute to cognitive alignment and effective coordination.

2. Background and prior research

2.1. Distributed innovation

Many studies of distributed innovation processes consider the organizational challenges from the perspective of the business firm. This perspective explains why the expression "involvement of external sources of innovation" (e.g. Nishiguchi, 1994, Clark and Fujimoto, 1991) has emerged and is so dominant in an important part of the innovation management literature. A previously "autarchic" (Christensen, 2006) view of the business firm and a previously strong 'inward-looking focus' has rapidly shifted focus to distributed innovation processes even in traditional industries.

The major motivation for involving external sources of innovation in product development is that doing so can increase the performance of new product development projects (Clark, 1989; Womack et al., 1990; Clark & Fujimoto, 1991; Wheelwright & Clark, 1992). Prior literature has also identified some of the main causes of problems in achieving high project performance in distributed innovation. Among these, the most important one especially when products are complex, is the management of technical interdependencies (Sosa et al., 2004). It is not trivial to coordinate a system when the innovation process is broken down into smaller parts and such smaller development tasks are allocated to a number of different actors. The problem is that without effective coordination, technical interdependencies may result in inconsistencies between changes in one component or subsystem and innovation in other components or subsystems. As a consequence, lack of effective coordination leads to low project performance. Prior research has identified four issues firms have to address to achieve high project performance: (1) how to divide the development task, (2) how many (and which) sources of external innovation to allocate the sub-tasks to, (3) how to coordinate the actors that develop components and subsystems, and (4) once the components and subsystems have been improved and developed, how to integrate the components and subsystems into a whole that has high product performance

(Baldwin & Clark, 2000; Takeishi, 2001, 2002; Brusoni et al., 2001; Laursen and Salter, 2006). In this article, we focus on the third issue, how to coordinate the actors that develop components and subsystems.

2.2. The coordination of distributed innovation processes

The literature on innovation management – in particular for highly complex systems – identifies two ways of tackling the organizational challenges involved in distributed innovation. First, a focal firm acts as system integrator (Hobday, Davies and Prencipe, 2005). This approach consists in orchestrating the other partners involved in the product development process through a top-down hierarchical approach. The key challenge involved is to take into account all interdependences between actors in aligning their actions. The second possibility is to rely on modular product architecture, i.e., an architecture where interdependences are bundled within modules while modules are independent of each other, and that have standardized interfaces (Baldwin & Clark, 2000). Such a modular product architecture supposedly allows using a modular organization structure (both within the firm and in the value chain) (Sanchez & Mahoney, 1996). It provides a powerful possibility: external sources of innovation can accomplish their development tasks independently and do not require explicit coordination, as the standardized interfaces and independence between modules assure that modules will fit together even without coordination when integrated into the overall product (Baldwin & Clark, 2000). In both approaches, the coordination problem is moved to the level of the architecture and, ultimately, to the architect who designs it. In self-organized social bodies it is however difficult for architects to master the process in the same way they could do in a firm. The main problem is that it is very difficult to develop a full-fledged architecture at the beginning of a self-organized process of development, because self-organization implies a risk of the emergence of paths that were not visible at the

beginning. As an example of self-organized distributed innovation and to see how coordination is achieved in such an environment, consider open source software (OSS, e.g., von Hippel and von Krogh, 2003).

Open source software (OSS) is an emblematic example of self-organized distributed innovation that posed further empirical and conceptual challenges related to the issue of coordination to the managerial literature. Markus (2007) provides a comprehensive survey of the studies of coordination in OSS. His distinction of the different definitions of governance structures includes coordination as one of the central challenges they address: "In the operational coordination literature, OSS governance is understood as a solution to [… the problem of] loss of operational control, and the solution is techniques for managing the process of OSS development work" (p. 156). A peculiar feature of the OSS innovation model is that individuals self-select the tasks they perform (Langlois and Garzarelli, 2008), leading to the emergence of self-organized systems (Kogut and Metiu, 2001; Lee and Cole, 2003; David and Rullani, 2008) where authoritative structures (Mateos-Garcia and Steinmueller, 2008) and leaders (Lerner and Tirole, 2002) are continually created, renewed or destroyed. Social processes leading to this emergence, such as criticism of the status quo (Lee and Cole, 2003) or creation of specific patterns of social ties facilitating the making of individuals into leaders (Dahlander and O’Mahony, 2011), became the center of many recent studies in the field. By the same token, attention was devoted to the limits of such processes, investigating how conflicts are resolved (Elliott and Scacchi, 2003) and how authoritative structures are challenged and changed (O’Mahony and Ferraro, 2007). This stream of literature has identified two coordination mechanisms that are applied in the OSS setting (Markus, 2007; O’Mahony and Ferraro, 2007; Dahlander and O’Mahony, 2011): coordination by project leaders and by artifacts. The next two sections report on these two issues.

2.2.1. Leadership

The first coordination mechanism identified is *leadership*. Indeed, Giuri et al. (2008) show that in OSS leaders are endowed with wider skill sets, enabling them to combine the heterogeneous inputs coming from a differentiated set of contributors. Involving voluntary participation, OSS collaboration implies a very peculiar nature of control, which can be centralized into a leader (as happens for Linux, Raymond, 1998; Lerner and Tirole, 2002) but needs to be continually reproduced and legitimated by those constituting the "lower layers" of the pyramid. Raymond (1998) argues that legitimation as leader comes naturally as a consequence of project foundation: the founder considers it recognized by other participants that she has the right to take the final decision with respect to the development of the project. However, legitimation is not a static concept, and has to be recreated and renewed each time (O'Mahony and Ferraro, 2007). Muller (2006) shows through a simulation model how leadership emerges as the result of a dynamic legitimation process among peers clustering around 'opinion leaders'. This effect, however, obtains a weaker support by O'Mahony and Ferraro (2007), who find that the antecedents of leadership acquisition are more related to face-to-face meetings and impact of members' contributions than to online communication. Nevertheless, they admit this may be the result of the impossibility to account for the content of the online communication in their econometric analysis, as their ethnographic study reaches the opposite conclusion. The authors connect these results to the evolution of the OSS project they study (Debian) and identify the transformation the authoritative structure and the governance mechanisms go through when the project moves from one phase of development to a more complex one. Their description of the dynamic transformation of governance highlights the passages from an autocratic leadership towards a formalized authority structure that acquires legitimation through the construction of democratic regulatory processes. This echoes Lee and Cole's (2003) identification of the community

debate as the key mechanism through which the OSS community evolves. Following the same dynamic perspective, Mateos-Garcia and Steinmueller (2008) argue that "as the capabilities the integrator has for keeping up with a project's development pace start diminishing, a structure with layers of trusted individuals emerge as a way of helping her or him cope with the increased complexity and size of the project. These layers will be composed of proficient individuals with experience in the project. The vision they have of the project will also concur with that of the leader in some essential points" (p. 22). Thus, a pyramidal structure gradually emerges, but again based on legitimation mechanisms involving the leader's vision as well as the technical capabilities of the developers coming to populate the intermediate layers.

Elliott and Scacchi (2003) apply a perspective that allows a more fine-grained definition of the legitimation process. Inspired by the studies that highlight the community-related aspects of OSS projects (e.g. Amin and Cohendet, 2004) the authors describe how developers resolve conflicts over the legitimacy of undertaking certain disputed actions (such as using software tools that are not open source to produce material for an open source project). The reference to common values and shared norms are the main rhetoric instruments used to solve the conflicts arising when some one's actions are disputed. What is interesting to highlight here is that the authors recognize not only the importance of the norms themselves, but also the fact that they are embodied in the online discussion stored in easily accessible web repositories. "This fast access to archived information perpetuates the cultural beliefs that have been articulated and assists in resolution of conflicts" (Elliott and Scacchi, 2003; p. 9).

2.2.2. Artefacts

A recent stream of literature tries to merge the social side dominating the previous points of view with the "materiality" of the production process itself (Orlikowski, 1992,

2000; D'Adderio, 2003; Cacciatori, 2008). Software developers, in fact, do not interact only exchanging opinions, but also act on artifacts – the lines of source code composing the software – that they exchange and jointly develop (Lanzara and Morner, 2005). The structure of the code, and in particular its modularity, has been identified as a crucial issue, because when interaction between individuals is mediated by artefacts, it is their very characteristic to allow the achievement of coordination. David and Ghosh (2008) have shown that the structure of the technical interdependencies between the different module composing Linux have a certain degree of correspondence to the pattern of social ties the authors of those modules have built through past collaboration. A similar perspective emerges from the analysis undertaken by Narduzzo and Rossi (2005) in their effort to define modularity in the OSS context. In OSS the architecture of the software is likely to be constantly changed over time, and so is the degree of modularization it embodies. Narduzzo and Rossi study how a community of developers copes with the emergence of interdependencies, and notice that every time a new dependency connects two modules challenging the current architecture, the main principle of modularity – information hiding – is reversed: developers exchange module-specific information with the aim of discussing a common understanding of the new logic underpinning the product. Again, the artifact's structure and the social side are coupled in the process of product development. MacCormack et al. (2006) focus on a similar process but with an opposite perspective: they also study the transformation of a weakly-modular OSS product into a highly modularized software, but in the context of a transformation implemented top-down by a firm (Netscape) opening the source code of its software as a strategy to attract external developers. They observe that the project was initially stagnating because its weakly-modular structure increases the cost of external developers' contribution, and that the subsequent increase in modularizing determined instead its success. The strict link between participation and modularity has been further developed by Baldwin and Clark

(2006). Modularity is seen not only as the determinant of a lower cost of contributing. Moreover, a high level of modularity assures a higher option value for external developers in terms of the possible future configurations of the product, and thus increases their willingness to participate. Modularity is again seen has having a social effect: decreasing free-riding. The other social effect the artifact structure has is more relate to our topic: coordination. As Lanzara and Morner (2005) explain, in OSS the code is “exposed”, everyone can read it and evaluate it, run it and contribute to it. The code thus sends signals that the individuals interacting with it receive. These signals are relative to the parts of the program that need further development, to the functions that do not perform properly, and the like. For example Dalle et al., (2011) show that the level of complexity and the level of modularity of different software packages are among the main determinants of the level of collaboration emerging around each specific package. But the code sends signals also about the different rewards one can expect to receive when contributing to a particular module instead of another. For example, modules with more collaborators or modules at the core of the program assure higher visibility, and thus can attract by developers moved by reputation concerns (Lerner and Tirole, 2002). To the contrary, developers with specific needs that the software fails to fulfill might want to contribute to an obscure module if it is crucial for the function they care of (Dalle and Jullien, 2003). In this way, different modules attract different developers. The allocation patterns of developers’ contribution emerging from this process will in turn change the structure of the code, and thus the rewards associated to each module in the next round. A new allocation of developers’ effort will emerge in response to this new pattern, and so on. The final code will be the result of this co-evolving process between the structure of the code and the motivations of the developers.

3. Hypothesis development: the role of initial code release in open software development

The literature review has highlighted the important role that emergent authority structures on the one hand, and code as artifact on the other, have played in the literature on open source software development. The existence and emergence of unexpected interdependencies, a problem self-organized structures need to cope with almost by definition, is not however clear, and neither are the associated costs. Leaders can act as system integrators and specify the architectural principles to be followed by the other participants (i.e., the interfaces) from the beginning. They do, however, face limits imposed by the self-organization of the innovation process. These limits result in the impossibility for boundedly rational agents to forecast all the possible interdependencies between modules that might emerge in an environment where developers choose their tasks and efforts and can question each decision, thereby challenging the authority and its technical procedures (including the architecture) at every moment. In OSS, the coordinative power of leadership is limited by the fact that any authoritative structure in which few developers hold the right to specify the initial architecture need to engage in a process of continuous legitimation, through democracy and information disclosure, and ultimately re-negotiation of the tasks division between the contributors. Conflict resolution and legitimation procedures able to sustain the authoritative structure of the organization (David and Rullani, 2008; Lanzara and Morner, 2005) affect the development path of the project. The social and the technical side are so interwoven (David and Ghosh, 2008) that the former interfere with the latter, possibly altering the product's structure of interdependencies. When redesign becomes a necessity, developers need to gather information on other modules and discuss the architectural changes to be made (Narduzzo and Rossi, 2005). Thus, on top of the limits typical of bounded rationality, OSS

also faces the limits imposed by self-organization to the possibility of specifying a full-fledged architecture since the beginning. In sum, even despite leadership, system integration and modularity, "[o]pen source software suffers from some lack of coordination" (Lerner and Tirole, 2005: 107).

The second coordination mechanism that the literature argues is being used in open source software development, code as an artefact, has been also investigated quite extensively. Some studies have focused on the fact that, when a project is created, the founder often provides "some running code" (Lerner and Tirole, 2002). This provision of initial code has been interpreted as a (mostly unintentional) strategy to attract other developers. Because of the importance of intrinsic motivation (Lakhani and Wolf, 2005) such as fun in coding and in solving challenges among the main incentives driving developers' participation (Ghosh et all., 2002; David et al., 2003) for deciding to join a particular project (David and Shapiro, 2008), the fact that programmers can directly test the code, see what works and what does not, find interesting problems and "scratch a personal itch" of theirs (Raymond, 1998) has been indicated as a crucial mechanism in attracting developers to projects.

Connected to this, Lerner and Tirole (2002) noticed that initial "runnable and testable" releases of the code attracted new developers that see interesting challenges. The authors also notice that initial code has a legitimation function: it shows to other potential developers that the code has some merit and that developing it further will not be a waste of time. This concept is further developed by Haefliger et al. (2008), who discuss the fact that a project can attract more developers if its initial code release effectively carries the "credible promise" of a stream of interesting challenges for future developers. The capability of code to attract interest and new developers has also been empirically documented by von Krogh et al. (2003), who notice that in the case of Freenet no code was provided at the beginning but

rather, only 15 weeks later, it was only after this release that the project witnessed a steep increase in attention, number of contributors and of discussions.

Haefliger et al. (2008) discuss the role of initial code relating it to code reuse (Sojer and Henkel, 2010). In OSS, code is very rarely created from scratch, as most developers tend to re-use code already available to save time and focus on the most interesting and still unresolved issues. The initial code, thus, does not need to be brand new, but can be – and often is (Haefliger et al., 2008) – existing code adapted to new purposes. Moreover, the reuse of the initial code itself, combined with other existing code and with brand-new code, will be the basis of the future versions of the program, increasing the path dependence of the development process (David, 1985).

Initial code is thus the starting point from which the program will be developed. In this perspective the initial code can be seen as a artefact whose elements will shape the possible future direction of development of the product, and which might instill path-dependence (David, 1985). From this perspective, the advantage of having some running code to work on is directly connected to the other developers' need to improve, instead of create, the code that provides the basis of the project (Sojer and Henkel, 2010; Haefliger et al., 2008). The difference from creating from scratch is substantial: creating from scratch also means imagining the future structure of the program, and having at least a vague idea of the whole system's functions. The initial code is, hence, an artifact that defines the technological space where the subsequent innovation will take place.

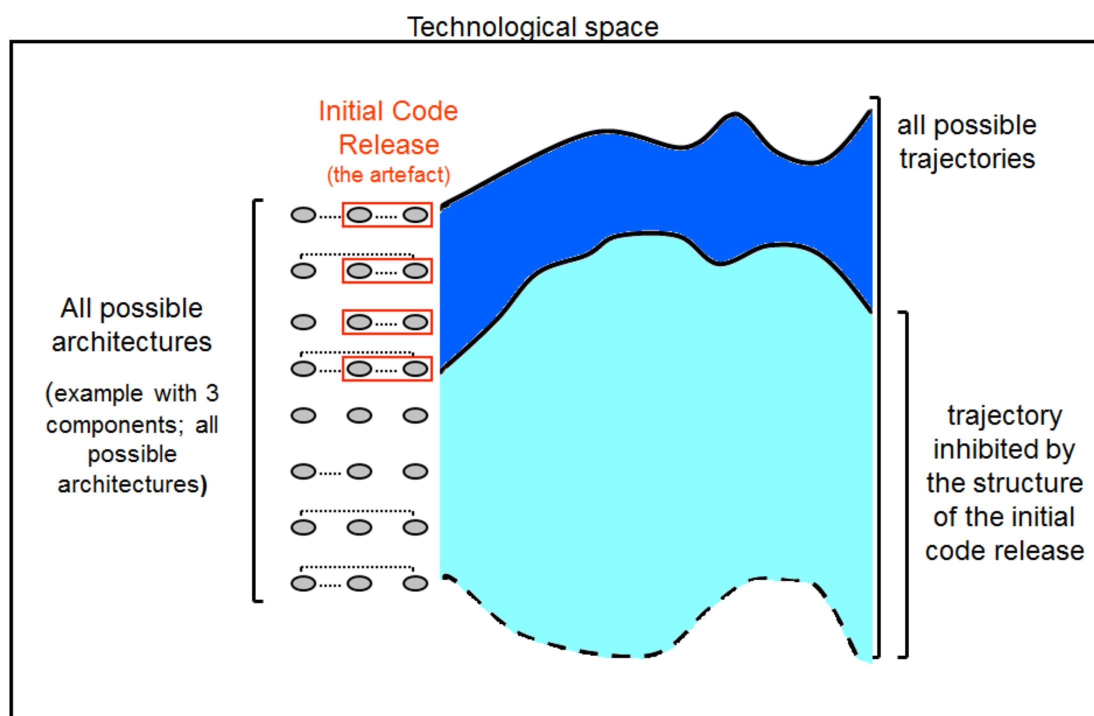
Notice that this intuition is consistent with the role initial code holds in the previously surveyed literature: the attractiveness of the project quickly decays if the founder simply asks other developers to join a technological space defined only by a project description (even if

well-detailed). Some code is needed to give a practical and not only theoretical definition of the expected "shape" of that space.

To have a clear idea of this, consider that the initial code release provides *in nuce* an indication of the possible software architectures that could be developed. Consider for example a software program composed by n components. Combining these components in a Design Structure Matrix, it is easy to see that the number of elements composing the off-diagonal upper-right triangle of the DSM is $N = \frac{n(n-1)}{2}$, and thus that the number of all possible relations between them is $1 + \sum_{k=1}^{N} \frac{N}{k!(N-k)!}$. This is an explosive function of n, leading to an enormous amount of possible future architectures even for quite small n. This number can be diminished if the initial release of the code contains m components and relates them in a specific manner, creating path-dependence (David, 1985). Even if $m \ll n$, m is a small fraction of the n components, the chosen set of preferred logics of component coupling emerges clearly from the simple observation of the software release: the m components are related by a specific set of relations that make some of them more independent, while others are much more tightly linked. This eases any further development of code that embeds the same decomposition logic, and increases the costs for developments that presume a different architecture requiring the reconfiguration of the relationship between the m components. Thus, future developments crossing the lines of the decomposition logics embedded in the first code released are more costly in terms of the number and importance of the modifications than the developments these lines naturally imply. Vice versa, those improvements and enlargements that respect the way in which the interdependencies are managed according to the decomposition embedded in the initial software release, will be perceived as natural developments.

The artifact, thus, has the same function the technological paradigm has at the level of the whole industry (Dosi, 1982). The initial code, in this light, can be considered the technological paradigm that constrains the vision and the innovation effort of the group of developers within a given set of problems and trade-offs, determining which technological trajectories can be developed and which cannot. See Figure 1 for a graphical representation on the technological space (Olsson and Frey, 2002) of how the artefact guides the future paths of the development of the distributed innovation project in the case in which we have three components with an artefact (the initial code release) spanning two of them (i.e., $n=3$ and $m=2$).

Figure 1. Initial code inhibits some development trajectories and favors others.



In this role, the initial code has an impact on actors' cognitive alignment, i.e., it helps actors develop a common understanding about the likely development of the OSS project as a whole and the role of their own specific contribution, limiting the problems arising from self-

organization (i.e., the impossibility to provide a full-fledged specification of the whole architecture since the beginning).

In turn, such alignment will increase the probability that the OSS project actually delivers a new release of the program.

The previous theorizing leads us to formulate the following hypotheses:

H1: The presence of an initial artifact (a code release) increases the probability of future cognitive alignment between OSS project members.

H2: The higher the cognitive alignment between OSS project members, the higher the probability that the project produces a subsequent code release.

Notice that we do not claim any role of the initial code in reaching a higher degree of innovation. The effect of the initial code is here evaluated only with reference to its capability to produce subsequent code release(s), irrespectively of the quality of the software released, e.g. whether it fosters innovation or produces radical vs. incremental innovation.

In the next section, we turn to empirically testing our hypotheses on a sample of projects hosted on SourceForge during 2005 and 2006.

4. Empirical investigation

4.1. Evidence from SourceForge

4.1.1. Data

Our analysis of the OSS context draws on data relative to all the projects populating the SourceForge platform from November 1999 to October 2008 (Madey, 2009; Gao et al., 2007). Out of this population we selected 5810 projects registered on the platform from January 15th, 2005, to April, 15th 2005. We tried to select the most recent projects we could, provided that we had enough data to analyze subsequent developments of the projects. Moreover, as the data consist of monthly snapshots of the situation observed on the platform, and as the way data were stored and managed changed over time, we needed to carefully select the period of analysis in order to preserve data consistency between the different snapshots. The final sample (5703 projects) has been obtained considering only the projects still registered as "Active" in September 2006 and with at least one member in July 2005 and March 2006.

We divided the period in three parts: a first period, called t_0, spans the first months of the projects' life up to July 8th, 2005. A second period (t_1) starts from this date and reaches March 20th, 2006. A third time window (t_2) is opened from that day to September 20th, 2006. We will use these time windows to make sure that in every equation we estimate, the independent variables are lagged by one period with respect to the dependant variable, thus reducing endogeneity.

4.1.2. Measures

Within this set of data we need to capture the connection between the provision of an initial piece of code, the process of alignment as captured by a reduction in the mismatch between

the developmental trajectories followed by the participants in the project, and the subsequent capability of the project to produce a newer version of the code.

We capture the first element of this sequence by *Initial code* t_0. It is a dummy variable equal 1 if in t_0 the project team releases an initial version of the software to be developed. In order to set a minimum threshold for the definition of "initial code", we restricted our analysis to the provision of "running code", i.e. to the presence of files released by the project members as official releases, in agreement with Lerner and Tirole's (2002) claim that a minimum amount of running code would be necessary to attract developers and let them play with it (Raymond, 1998).

Capturing alignment is much more tricky. We need to identify an observable process that proxies the cognitive level, where alignment happens. We thus need to rely on the actions developers undertake as a consequence of alignment to actually detect the presence and intensity of such alignment. To do this we can exploit the fact that in SF.net projects are self-categorized by their members in different categories. These are: intended audience of the project (end users/desktop, developers, …), the programming languages it employs (C++, Java, …), the operating systems it runs on (Windows, Linux, …), the topic the project tackles (communications, security, games/entertainment, …), the environment it populates (X11 applications, web environment, …), and the language used by the developers to interact one another. Each one of these macro categories is then organized into lower level categories. So for example the category "topic" is divided into different subcategories among which "communications", which in turns contains - among others - the subcategory "chat", that leads to two possible final subcategories: "AOL Instant Messenger", and "ICQ" (see Table A1 in the appendix for another example[2]). The tree of the categories offers a portrait of the

[2] A last category is represented by the development status of the project. As the project progresses, the team updates this category to inform the public that the project has passed, for example, from its beta version to its mature version. We do not include this category into the analysis, as it is directly capturing the productivity of the project rather than simply describing it.

project along several dimensions and indicates what is the trajectory along which the development is moving. For example, if a project indicates that its programming language is C++, we know that developers are aligned in considering C++ the main language they should use to develop the program. In sum, in every period in time categories represent a synthesis of the different trajectories followed by the project participants. This synthesis is however not free of clashes and heterogeneity of visions and ideas. Categories in fact are "truces" between the different visions of the members, and hold only until the misalignment between these visions is below a certain threshold. When attrition raises above that threshold, there is a clash between the different visions of the product and of the development process, and categories may change consistently to reflect the new truce found in the team, precisely as it happens in organization when new routines are formed (Nelson and Winter, 1982). Thus we can count the number of changes occurred in the classification categories over a period of time to have a sense of the number of re-alignments occurred in the team, and thus of the number of episodes in which the divergence of the development trajectories followed by the project members has increased above a certain threshold. The higher the number of observed changes in the classification in categories in certain period, the higher the number of episodes of misalignment in that period, the lower the cognitive alignment the team experiences in that period. We thus build our measure considering the 8 months spanned by t_1 and comparing the list of categories each project posts at the beginning of that period to the same list at the end of the period, counting a change in the list every time a new category is added or an old category is dropped.

The last step of our analysis deals with capturing the probability that the project produces an outcome. We measure this considering whether the project has produced and released any file over the 6 months composing period t_2. In OSS many projects are just dormant and produce no activity at all (Krishnamurthy, 2002). In this environment releasing some code, even if far

from being a precise measure of productivity, is also a reasonable proxy discriminating between productive and non-productive projects.

Eventually, we need to include a number of controls to take into account possible confounding factors and alternative stories. We report them in the Table 1 together with the main variables described above (see Table A2 and A3 in the appendix for the summary statistics and the correlations).

In sum, the data described above are used to investigate if the provision of an *initial code* at the moment of a project's foundation (t_0) leads to a reduction of the *number of category changes* during the development of the project (from the beginning to the end of t_1), and if this effect in turn increases the probability of observing a *new release* of the code in t_2.

Table 1 – Variables used in the regression analysis

Variable	*Description*
INITIAL_CODE$_{t0}$	dummy variable equal to 1 if there was at least one file posted by the project in the first months of its activity on SF.net, i.e. between its foundation and July 8th, 2005.
CATEGORIES_CHANGES$_{t1_t2}$	number of categories the project has "changed", i.e. acquired or lost, between July 2005 and March 2006
CATEGORIES_CHANGES$_{t1_t2_p}$	number of categories the project has "changed" between July 2005 and March 2006 as predicted by the first equation of the model
CODE_RELEASED$_{t1}$	dummy equal 1 if the project has posted at least one file between July 8th, 2005 and March 20th, 2006
CODE_RELEASED$_{t2}$	dummy equal 1 if the project has posted at least one file in the 6 months from April 2006 to September 2006 included
MEMBERS_TENURE$_{t1}$	registered date of those who were project members at July 2005 (average)
MEMBERS_TENURE$_{t2}$	registered date of those who were project members at March 2006 (average)
NUM_MEMBERS$_{t2}$	number of project members at March 2006 (average)
NUM_MEMBERS$_{t1}$	number of project members at July 2005 (average)
REGISTRATION_DATE$_{t0}$	registration date of the project on the platform
USE_CVS_TOOL$_{t0}$	dummy equal 1 if the project uses the CVS, concurrent versioning system, a tool to manage distributed software development (May 2005)
USE_FORUMt_0	dummy equal 1 if the project uses forums (May 2005)
DUMMY_CATEGORIES$_{t0}$	dummies for language, programming language, license, operating system, development status, topic, retrieved July 2005, i.e. end of period t_0

**Notice: dates are measured in UNIX time, i.e. in number of seconds from midnight of January 1, 1970, a standard measure in computer science.*

The estimation we run is aimed at showing the relationship between the three main variables defined above. In order to do this we run a first estimation, using ***INITIAL_CODE***$_{t0}$ as independent variable and ***CATEGORIES_CHANGES***$_{t1_t2}$ as dependant variable. Then we predict the values of ***CATEGORIES_CHANGES***$_{t1_t2}$ (calling this new variable ***CATEGORIES_CHANGES***$_{t1_t2_p}$) and use them as the main regressor in a second equation, where the dependent variable is code release in t_2 (***CODE_RELEASED***$_{t2}$). In this way we capture the effect of number of changes in categories on code subsequent release(s) when those changes are *exclusively* due to the presence of initial code. We do this to exclude any impact of ***CATEGORIES_CHANGES***$_{t1_t2}$ and ***CODE_RELEASED***$_{t2}$ not due to ***INITIAL_CODE***$_{t0}$ (and the controls of the first stage).

The first equation we need to estimate relates ***INITIAL_CODE***$_{t0}$ and ***CATEGORIES_CHANGES***$_{t1_t2}$. Measuring the independent variables at time t_0 and the number of changes in the categorization as the delta between the beginning of t_1 and its end (and also controlling for the specific categories projects listed at the end of t_0) should diminish the possible endogeneity problem. As ***CATEGORIES_CHANGES***$_{t1_t2}$ is a count variable we could use a Poisson specification. However, the presence of overdispersion pushes us to prefer a Negative Binomial. By the same token, the presence of many zeros implies the use of an additional equation preceding the estimation stage and predicting the probability of being a project which has structurally 0 changes in the categories it lists. The results of this "zero-inflation" process are used in the main estimation equation. The final model for the first stage is then a Zero-Inflated Negative Binomial, ZINB (in the table the results relative to a Zero-Inflated Poisson, or ZIP, are also reported as a robustness check). Notice that the sample of projects is also reduced to 5709 due to the fact that we excluded about 100 project that were deleted from SourceForge before March 2006.

Once the estimation of this first stage has been carried out, the predicted values of ***CATEGORIES_CHANGES***$_{t1_t2}$ are used as independent variable in the second equation, which includes ***CODE_RELEASED***$_{t2}$ as dependant variable. ***CODE_RELEASED***$_{t2}$ being a dummy variable, it seems appropriate to use a Logistic Regression Model.

As said, some controls have been included in the two equations to make sure no confounding factors are at work. In particular, ***CODE_RELEASED***$_{t1}$ and ***INITIAL_CODE***$_{t0}$ are also included in the

second stage. ***CODE_RELEASED****$_{t1}$* controls for the most recent performance of the project (in period *t1*) while introducing ***INITIAL_CODE****$_{t0}$* also in the second equation controls for the direct effect of ***INITIAL_CODE****$_{t0}$* on ***CODE_RELEASED****$_{t2}$*. This last passage reinforces the claim we made before: our coefficient captures the impact of ***CATEGORIES_CHANGES****$_{t1_t2}$* on ***CODE_RELEASED****$_{t2}$* when ***CATEGORIES_CHANGES****$_{t1_t2}$* is determined exclusively by ***INITIAL_CODE****$_{t0}$* (and other controls).

Figure 2 provides an overview of the estimation procedure, while Tables 2a and 2b report the results of the estimates.

Figure 2. The estimation strategy

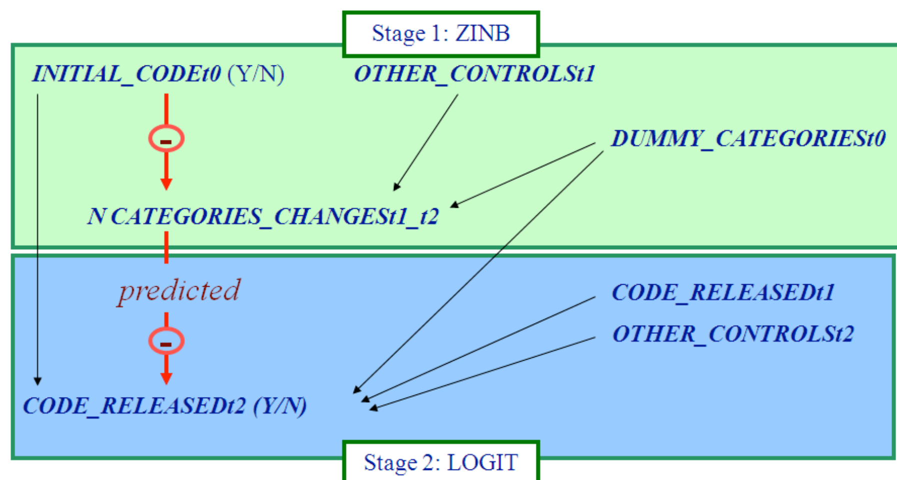


Table 2a – Results of the regression analysis – stage 1

Stage 1	*CATEGORIES_CHANGES$_{t1\ t2}$*	
	ZINB	*ZIP*
	b/se	*b/se*
INITIAL_CODE$_{t0}$	-0.312***	-0.251***
	[0.094]	[0.083]
REGISTRATION_DATE$_{t0}$	0.000	0.000
	[0.000]	[0.000]
NUM_MEMBERS$_{t1}$	0.042*	0.044***
	[0.024]	[0.016]
MEMBERS_TENURE$_{t1}$	0.000	0.000
	[0.000]	[0.000]
USE_FORUMt$_{0}$	-0.053	-0.047
	[0.170]	[0.146]
USE_CVS_TOOL$_{t0}$	-0.466**	-0.443***
	[0.186]	[0.167]
DUMMY_CATEGORIES$_{t1}$	YES	YES
Constant	-16.584	-19.879
	[22.675]	[20.744]
Inflation Model: logit		
INITIAL_CODE$_{t0}$	-0.236**	-0.180
	[0.115]	[0.113]
REGISTRATION_DATE$_{t0}$	0.000	0.000
	[0.000]	[0.000]
NUM_MEMBERS$_{t1}$	-0.077***	
	[0.026]	
MEMBERS_TENURE$_{t1}$	0.000	0.000
	[0.000]	[0.000]
USE_FORUM$_{t0}$	0.232	0.286
	[0.209]	[0.208]
USE_CVS_TOOL$_{t0}$	0.280	0.282
	[0.240]	[0.240]
DUMMY_CATEGORIES$_{t1}$	YES	YES
Constant	34.407	30.367
	[26.244]	[24.773]
lnalpha	-.786***	
	[0.167]	
alpha	0.456	
	[0.076]	
N (nonzero)	5703 (461)	5703 (461)
ll	-2533.426	-2644.177
chi2 (df)	145.95 (24)	
Pr > chi2	0.000	
Vuong	9.02	
Pr>z	0.000	

Table 2b – Results of the regression analysis – stage 2

Stage 2	$CODE_RELEASED_{t2}$	
	LOGIT for ZINB	*LOGIT for ZIP*
	b/se	*b/se*
$CATEGORIES_CHANGES_{t1_t2_P}$	-0.775**	-0.700*
	[0.325]	[0.413]
$NUM_MEMBERS_{t2}$	0.156***	0.123***
	[0.027]	[0.022]
$MEMBERS_TENURE_{t2}$	-0.000	-0.000
	[0.000]	[0.000]
$CODE_RELEASED_{t1}$	2.495***	2.502***
	[0.113]	[0.113]
$INITIAL_CODE_{t0}$	0.831***	0.841***
	[0.131]	[0.131]
$DUMMIE_CATEGORIES_{t0}$	YES	YES
Constant	-2.240*	-2.312*
	[1.312]	[1.312]
N	5703	5703
Ll	1288.1201	-1290.0610
Pseudo R-squared	0.291	0.290
LR chi2 (df)	1059.50 (23)	1055.62 (23)
Prob >	0.000	0.000

In the first stage, the coefficient of ***$INITIAL_CODE_{t0}$*** is negative and highly significant. The provision of an initial piece of code thus leads to a decrease in the number of category changes during the course of the project. In the second stage, the coefficient of the predicated values of ***$CATEGORIES_CHANGES_{t1_t2}$*** is also highly significant, and also negative. This means that the smaller the number of changes in a project's categories (as predicted by the first equation), the more likely a project will effectively produce and release some files in the latest period considered. Moreover, as we have used the predicted values of ***$CATEGORIES_CHANGES_{t1_t2}$*** in the second stage, and as we have controlled for ***$INITIAL_CODE_{t0}$***, for ***$CODE_RELEASED_{t1}$***, and for the residuals of the prediction, we can also state that our regression captures ***exclusively*** how ***$CATEGORIES_CHANGES_{t1_t2}$*** -as determined by ***$INITIAL_CODE_{t0}$*** - affects ***$CODE_RELEASED_{t2}$***.

These results allow us to conclude that the presence of initial code at t_0 reduces the number of category changes in the course of the project (between t_1 and t_2), and that this restriction is positively correlated with the release of code in the course of the project (t_2).[3]

5. Discussion and conclusion

Our study shows that an initial code release in a new open source project aligns the cognitive structure of open source software developers, constraining the possible trajectories of development they could follow. We also argue that this, increases the chance of future code releases. The number of category changes in the course of the project measures the former effect, the existence of a subsequent line of code the latter. We interpret this finding as the consequence of the fact that the initial code helps contributors to restrict their search space, improve cognitive alignment and, as a consequence, coordination.

The support we provide for our first hypothesis shows that by making the architecture of the software (including the key interdependences among the components/functions) visible, initial code helps developers direct their efforts in a similar direction from early stages of the project development. Lanzara and Morner (2005) and Narduzzo and Rossi (2005) propose this link between code exposure and capability to create effective coordination, while Baldwin and Clark (2006)

[3] Note that this econometric analysis could be expanded. We cannot apply directly Sobel-Goodman test for mediation because we have no OLS and because of the different time frames of the 2 stages that imply different set of controls. To give an idea of the possible result Sobel-Goodman test (which is however only indicative), we reproduced that technique comparing the previous equations with an equation testing the direct effect of ***INITIAL_CODE$_{t0}$*** on ***CODE_RELEASED$_{t2}$*** (including 2 stage and 1 stage controls alternatively and contemporaneously, in order to take into account the phenomena relative to the whole time span). Considering only 2 stage controls the direct effect of ***INITIAL_CODE$_{t0}$*** is .8548467 with 95% Conf. Interval (.599205, 1.110488) and z=6.55. A similar figure is obtained when both stage 1 and 2 controls are considered. Including only 1 stage controls, instead, enhances the effect of ***INITIAL_CODE$_{t0}$*** up to 1.568536 (Std. Err.=.1187749) and 95% Conf. Interval (1.335742, 1.801331) with z=13.21. Ergo, including ***CATEGORIES_CHANGES$_{t1_t2_p}$*** reduces the coefficient and the z values of ***INITIAL_CODE$_{t0}$***, but only when early stage phenomena are controlled for. When also later phenomena enter the picture, the remaining reduction -even if still present- is not enough to be significant.
Ergo, when later phenomena enter the picture, the inclusion of ***CATEGORIES_CHANGES$_{t1_t2_p}$*** reduces the coefficient and the z values of ***INITIAL_CODE$_{t0}$***, but not enough to be significant. When instead early stage phenomena are controlled for, the reduction of the coefficient and the z values of ***INITIAL_CODE$_{t0}$*** is clear and significant, confirming our analysis.

suggest that the visible architecture of OSS allows to solve free riding problems, again fostering coordination. We move a step forward by making this link explicit, and showing that initial code fosters cognitive alignment between participants. This idea is then supported by a two stage analysis able to isolate and expose the underlying mechanism.

Our second hypothesis links cognitive alignment to the chance that new code releases will be produced. This is a distinguishing factor in OSS development. As noted elsewhere (Krishnamurthy, 2002) only a very limited percentage of projects advance, and coordination plays a key role in improving the chance that the project stays alive and makes progress. Our result adds a twofold contribution to this point. In the first place, we add and test the impact of a new mechanism for OSS coordination, i.e. the initial code release. While previous literature has shown that initial code has an important role in OSS projects, we specify and document the specific impact of initial code release cognitive alignment and the coordination (alignment of action) that is required for joint action to lead to results. Secondly, the mechanism by which such effective coordination is achieved seems to open up new opportunities for conceptual and empirical developments, also beyond OSS. We show that the mechanism by which effective coordination is achieved is that of making the structure of the software under development more transparent, i.e. exposing the interdependences among the components of the software. Although only preliminary, this result hints at a novel role played by the initial code release that goes beyond the role of "boundary object" or "modular interface", i.e. of the two other concepts that make explicit reference to the link between artifacts and coordination. While previous literature has emphasized the role of boundary objects in "translating" messages from different epistemic communities and "connecting" such communities (Carlile, 2002), the initial code release, or better the potential future architectures which the current interdependences of this initial software facilitate, allow each participant in the joint production process to understand which trajectories will be feasible and which will be more complicated, making their visions of the overall final version of the product converge to a narrower set of possibilities. This is what we have called "cognitive alignment", a process that has nothing do to with the concept of "translation" but rather deals with a sort of implicit coordination, where effective coordination is achieved limiting the scope of the search process of each actor.

Concerning the difference between the role of the initial code release and modularity, our study contributes to a recent stream of literature that discusses the very role of modularity in the actual coordination of complex products in self-organizing environments. Modularity works thanks to the existence of interfaces that, in turn, help decoupling the development of each single component making up the system. Previous research has shown that in self-organized and distributed innovation processes (Narduzzo and Rossi, 2005) and in other complex settings (Brusoni, 2005, Zirpoli and Becker, 2011, Camuffo and Cabigiosu, 2012), it is difficult to realize the assumption behind modularity, i.e., that the ex ante definition of the interfaces is possible and implementable by a *hierarchical* structure solving all the problems of decomposition. The effect of the initial code release works on the basis of a totally different mechanism. Instead of assuming the possibility of decoupling the design activity of each single component ex ante, it acknowledges the existence of complex interdependences and makes their structure and logic transparent to everybody by means of the exposure of the code. Each single developer exposed to the initial artifacts is also exposed to the potential trajectories it embodies.

This solves also another problem intrinsic to modularity. Architects are humans and thus rationally bounded. Any vision or set of rules explicitly designed by the architect thus contain only a limited amount of information: that known by the architect. As a consequence, all these tools will be also bounded. The initial code, and the initial artifacts in general, instead have a much wider reach. Irrespectively of what the initial artifact provider consciously places in the artifact, the object itself has a materiality that is not constrained by the bounded rationality of the provider. When participants other than the initial artifact provider are in doubt in judging whether a certain trajectory they could follow will raise a series of unforeseen interdependencies, while they cannot rely on the bounded rationality of the initial artifact provider to have an answer, they can instead just turn to the object and observe the structure of the artifact. Every trajectory whose implementation would imply an objective conflict with the artifact will be outside the possibility space, while those trajectories consistent with it can be easily implemented.

This discussion shows that our results have clear implications for research on coordination. Classic organization theory distinguishes coordination by plan and by feedback (Cyert & March,

1963). While the mechanisms of providing coordination ex ante by plans are reasonably well understood, our understanding of how coordination is achieved without such mechanisms is less strong. The topic is of great interest due to limits to ex ante coordination and planning especially in uncertain and dynamic circumstances such as those typical of self-organizing social bodies (Anderson, 1999). An active stream of research in the organization literature focuses on emergent and implicit coordination (e.g., Gittell, 2000; Rico et al., 2008; Okhuysen & Bechky, 2009). Our findings add to this stream of research and to the understanding of what contribution artefacts can make to the mechanisms underlying emergent coordination. By laying out key interdependences in the complex product, initial artifacts help cognitive alignment to emerge. This contrasts with other possible detailed mechanisms underlying emergent coordination such as affordances that objects can make, or creating shared representations of each others' actions (Knoblich et al., 2012). By linking up recent research on the mechanisms underlying emergent coordination with identifying the impact of artefacts, our paper points to a line of research that can cast light on the role of artefacts for emerging coordination, for joint product development and for innovation management in distributed environments.

References

Amin, A., Cohendet, P, 2004. *Architectures of Knowledge: Firms, Capabilities and Communities*. Oxford University Press: Oxford, UK

Anderson, P., 1999. Complexity theory and organization science. *Organization Science*, 10: 216-232

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

Baldwin, C.Y., Clark, K.B., 2006. The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model? *Management Science*, 52: 1116-1127.

Brusoni, S., 2005. 'The Limits to Specialization: Problem-solving and Coordination in Modular Networks', *Organization Studies*, 26/12: 1885–1907.

Brusoni, S., Prencipe, A. and Pavitt, K., 2001. Knowledge specialization, organization coupling, and the boundaries of the firm: Why do firms know more than they make? *Administrative Science Quarterly*, Vol. 46, No. 4, 597-625

Cabigiosu A., Camuffo A., 2012. Beyond the "Mirroring" Hypothesis: Product Modularity and Interorganizational Relations in the Air Conditioning Industry. *Organization Science*, 23(3): 686-703

Cacciatori E., 2008. Memory objects in project environments: Storing, retrieving and adapting learning in project-based firms, *Research Policy*, Vol. 37, No. 9., pp. 1591-1601.

Carlile, P. R., 2002. A Pragmatic View of Knowledge and Boundaries: Boundary Objects in New Product Development. *Organization Science*, Vol. 13, No. 4: 442-455.

Chesbrough, H., 2003. *Open Innovation*. Free Press, New York

Christensen, J. F., 2006. Wither Core Competency for the Large Corporation in an Open Innovation World?, in Chesbrough, H., Vanhaverbeke, W. and West, J. (eds.), *Open Innovation: Researching a New Paradigm*. Oxford: Oxford University Press, 35-61.

Clark, K. B., 1989. Project scope and project performance: the effect on parts strategy and supplier involvement in product development, *Management Science*, 35: 1247-1263.

Clark, K. B. and Fujimoto, T., 1991. *Product Development Performance*. Boston, MA: Harvard Business School Press.

Cyert, R.M. and March J.G., 1963/1992. *A Behavioral Theory of the Firm*. 2nd ed. Blackwell, Oxford.

D'Adderio, L., 2003. Configuring software, reconfiguring memories: the influence of integrated systems on the reproduction of knowledge and routines, *Industrial and Corporate Change*, vol. 12(2): 321-350.

Dahlander L, Frederiksen L, Rullani F., 2008. Online communities and open innovation: governance and symbolic value creation. *Industry and Innovation* ,15(2): 115-123.

Dahlander L., O'Mahony S., 2011. Progressing to the Center: Coordinating Project Work, *Organization Science*, 22(4): 961-979.

Dalle, J.-M., and Jullien, N., 2003. 'Libre' software: Turning fads into institutions?, *Research Policy*, 32, January, pp 1-11.

Dalle, J.-M., Paul A. David, P.A., Rullani F., 2011. Linking coordination, motivations and code structure in successful open source projects: A 'stigmergic' approach, presented at the *Academy of Management Annual Meeting*, August 2011, San Antonio, TX, US.

David P., Foray D., 2003. Economic fundaments of the knowledge society, *Policy Futures in Education* 1(1), January.

David P.A. and Ghosh R.A., 2008. Relating social structure to technical structure: A study of the Linux kernel, presented at the DIME - DRUID Fundamental on Open and Proprietary Innovation Regimes: "*Opportunities and limitations of the open source models of innovation and the role of intellectual property rights*", Copenhagen Business School, Copenhagen, Denmark, June 17, 2008

David P.A., Waterman A., Arora S., 2003. The free/libre/open source software survey for 2003, preliminary draft, September 2003, quoted with authors' permission, at http://www.stanford.edu/group/floss-us/report/FLOSS-US-Report.pdf.

David, P.A, Rullani F., 2008. Dynamics of Innovation in an “Open Source” Collaboration Environment: Lurking, Laboring and Launching FLOSS Projects on SourceForge, *Industrial and Corporate Change*, 17(4), p. 647-710.

David, P.A. 1985. Clio and the Economics of QWERTY. *American Economic Review*: 75(2): 332-337

David, P.A., J.S. Shapiro. 2008. Community-based production of open source software: what do we know about the developers who participate? *Information Economics and Policy* 20 (4) 364–398.

Dosi, G., 1982. Technological paradigms and technological trajectories. *Research Policy*, 11, 147−162.

Elliott M. and Scacchi W., 2003. Free Software Developers as an Occupational Community: Resolving Conflicts and Fostering Collaboration, Proc. ACM Intern. Conf. Supporting Group Work (Group'03), Sanibel Island, FL, November 2003: p. 21-30

Gao Y., Van Antwerp M., Christley S. and Madey S., 2007. A Research Collaboratory for Open Source Software Research, In the Proceedings of the 29th International Conference on Software Enginering + Workshops (ICSE-ICSE Workshops 2007), International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS 2007), Minneapolis, MN, May 2007.

Ghosh R.A., Krieger B., Glott R., Robles G., 2002. Free/Libre and Open Source Software. Part IV: Survey of Developers, International Institute of Infonomics, Berlecom Research GmbH, at http://www.infonomics.nl/FLOSS/report/Final4.pdf

Gittell, Jody Hoffer, 2000. Organizing work to support relational co-ordination. *International Journal of Human Resource Management*, Vol. 11, No. 3, 517-539

Giuri P., Ploner M., Rullani F., and Torrisi S., 2010. Skills, Division of Labor and Performance in Collective Inventions: Evidence From Open Source Software, *International Journal of Industrial Organization*, 28(1), 54-68.

Giuri P., Rullani F. & Torrisi S. 2008. Explaining Leadership in Open Source Software Projects, *Information Economics and Policy*, 20(4): 305-315.

Haefliger, S., G. von Krogh, S. Spaeth. 2008. Code reuse in open source software. *Management Science*, **54** 180–193.

Helper, S.R., MacDuffie, J.P. and Sabel, C., 2000. Pragmatic Collaborations: Advancing Knowledge While Controlling Opportunism, *Industrial and Corporate Change*, **9**, 443-488.

Hobday, M., Davies, A. and Prencipe, A., 2005. Systems integration: a core capability of the modern corporation. *Industrial and Corporate Change*, 14/6: 1109-1143

Knoblich, Günther, Stephen Butterfill and Natalie Sebanz, 2011. Psychological Research on Joint Action: Theory and Data. In Ross, Brian (eds) *The Psychology of Learning and Motivation*, Vol. 54, Burlington: Academic Press, 59-101

Kogut, B. and A. Metiu. 2001. Open-Source Software Development and Distributed Innovation, *Oxford Review of Economic Policy,* 17(2): 248-64.

Krishnamurthy, S., 2002. Cave or community? An empirical examination of 100 mature open source projects. *First Monday* 7(6).

Kuk G., 2006. Strategic Interaction and Knowledge Sharing in the KDE Developer Mailing List, *Management Science*, Vol. 52, No. 7, July 2006, p. 1031–1042.

Lakhani, K.R., R. G. Wolf. 2005. Why hackers do what they do: understanding motivations and effort in free/open source software projects. J. Feller, B. Fitzgerald, S. Hissam, K.R. Lakhani, eds. *Perspectives on Free and Open Source Software*. MIT Press, Cambridge MA.

Langlois, Richard N. and Garzarelli, Giampaolo, 2008. Of Hackers and Hairdressers: Modularity and the Organizational Economics of Open-source Collaboration, *Industry & Innovation*, 15(2), 125-143

Lanzara G.F., Morner M., 2005. Artifacts rule! How organizing happens in opens source software projects, in: Czarniawska B. and Hernes T., *Actor-Network Theory and Organizing*, Copenhagen, Copenhagen Business School Press, p. 67 - 90.

Laursen, K. and Salter A., 2006. Open for Innovation: The role of openness in explaining innovative performance among UK manufacturing firms, *Strategic Management Journal*, Vol. 27(2), pp 131-150.

Lee G.K., & Cole R.E., 2003. From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development, *Organization Science*, 14(6): 633-649.

Lerner J., Tirole J., 2002. Some simple economics of Open Source, *The Journal of Industrial Economics*, vol. L number 2, p. 197-234.

Lerner J., Tirole J., 2005. The Scope of Open Source Licensing, *Journal of Law, Economics, and Organization*, 21, 20-56.

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

Madey G., ed., 2009. The SourceForge Research Data Archive (SRDA). University of Notre Dame (February 2009) http://zerlot.cse.nd.edu/

Markus M.L., 2007. The governance of free/open source software projects: monolithic, multidimensional, or configurational? *Journal of Management Governance*, 11:151–163

Mateos-Garcia, J., Steinmueller W.E., 2008. The institutions of open source software: examining the Debian community. *Information Economics and Policy* 20 333–344.

Muller P., 2006. Reputation, trust and the dynamics of leadership in communities of practice, *Journal of Management and Governance* 10, p. 381–400

Narduzzo A., Rossi A., 2005. The Role of Modularity in Free/Open Source Software Development, in S. Koch (ed), *Free/Open Software Development*, Idea Group.

Nelson, R. and Winter S., 1982. *An Evolutionary Theory of Economic Change*. Belknap Press of Harvard University Press, Cambridge/MA

Nishiguchi, T., 1994. *Strategic Industrial Sourcing*. New York: Oxford University Press.

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

Okhuysen, G.A. and Bechky B.A., 2009. Coordination in Organizations: An Integrative Perspective. *The Academy of Management Annals*, Vol. 3, No. 1, 463–502

Olsson, O., and Frey B.S., 2002. Entrepreneurship as Recombinant Growth, *Small Business Economics*,19: 69–80

Orlikowski, W.J., 1992. The duality of technology: Rethinking the concept of technology in organizations. *Organization Science* 3(3): 398-427.

Orlikowski, W.J., 2000. Using Technology and Constituting Structures; A Practice Lens for Studying Technology in Organizations. *Organization Science* 11(4): 404-428.

Ouchi, William G., 1980. Markets, Bureaucracies, and Clans. *Administrative Science Quarterly*, Vol. 25, No. 1, 129-141

Powell, W.W, Koput, K.W., Smith-Doerr, L., 1996. Interorganizational collaboration and the locus of innovation: networks of learning in biotechnology, *Administrative Science Quarterly*, 41, 116-145.

Raymond, Eric S., 1998. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly Associates.

Rico, R., Sanchez-Manzanares M., Gil F. and Gibson C., 2008. Team Implicit Coordination Processes: A Team Knowledge-Based Approach. *Academy of Management Review*, Vol. 33, No. 1, 163–184.

Sanchez, R. and Mahoney, J.T., 1996. Modularity, Flexibility, and Knowledge Management in Product and Organization Design, *Strategic Management Journal*, 17, 63-76.

Sojer, M. and Henkel, J. 2010. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments, *Journal of the Association for Information Systems*, 11(12), Article 2

Sosa, M. E., S. D. Eppinger, C. M. Rowles, 2004. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, **50**(12), 1674–1689

Star, Susan Leigh and Griesemer, James R., 1989. Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social Studies of Science* Vol. 19, No. 3: 387-420

Takeishi, A., 2001. 'Bridging inter- and intra-firm boundaries: management of supplier involvement in automobile product development', *Strategic Management Journal*, 22, 403-433.

Takeishi, A., 2002. Knowledge Partitioning in the Inter-Firm Division of Labor: The Case of Automotive Product Development. *Organization Science,* 13: 321-338.

Von Hippel, E. and von Krogh, G., 2003. Open source software development and the private-collective innovation model: Issues for organization science, *Organization Science*, Vol. 14, No. 2, , pp. 208-223

von Hippel, Eric, 1988. *The Sources of Innovation*. Oxford University Press, Oxford.

Wheelwright, S. and Clark, K., 1992. *Revolutionizing Product Development*. New York: Free Press.

Womack, J.P., D.T. Jones and D. Ross, 1990. *The Machine that Changed the World*, Rawson Ass.: New York.

Zirpoli, Francesco and Markus C. Becker, 2011. The limits of design and engineering outsourcing: performance integration and the unfulfilled promises of modularity. *R&D Management*. Vol. 41, No. 1, 21-43