# Modular Design and the Development of Complex Artifacts: Lessons from Free/Open Source Software

Alessandro Narduzzo

Dipartimento di Discipline Economiche–Aziendali

Università di Bologna

narduz@economia.unibo.it

Alessandro Rossi

ROCK (Research on Organizations, Coordination and Knowledge)

Dipartimento di Informatica e Studi Aziendali

Università di Trento, arossi@cs.unitn.it

Quaderno DISA n. 80-2003

Università degli Studi di Trento

---

Mimeo version, September 26, 2003. Comments are welcome. A much shorter account, titled "The Role of Modularity in Free/Open Source Software Development" is forthcoming in S. Koch (ed.), *Free/Open Source Software Development*, Hershey, PA: Idea Group, 2004.

## Abstract

Organizational and managerial theories of modularity applied to the design and production of complex artifacts are used to interpret the rise and success of Free/Open Source software methodologies and practices in software engineering. Strengths and risks of the adoption of a modular approach in software project management are introduced and are related to the achievements of various Free/Open Source Software projects (among them: the GNU operating system, the Linux kernel, the HURD kernel). It is suggested that mindful implementation of the principles of modularity may improve the rate of success of many Free/Open Source software projects. Specific case studies here depicted, as well as indirect observation of common programming practices employed by Free/Open Source developers and users, suggest a possible revision towards an improved theory of modularity that may be extended also to settings different from software production.


*Keywords:* modularity, software project management, Free/Open Source Software, division of labor, coordination, information hiding.


## Introduction

The popularity of the GNU/Linux operating system has conveyed increasing attention to the Free/Open Source Software (F/OSS) development model, usually described as a radically different system of rules, practices, and methodologies, shared within a large and virtually distributed software developers community, alternative to proprietary and closed development techniques employed by traditional hierarchical organizations in the software industry (Raymond, 1999; Kogut and Metiu, 2001; Feller and Fitzgerald, 2000).

A growing number of studies has analyzed the F/OSS movement, from a variety of perspectives, aimed to understand the bases of such successful phenomenon.[1] Our study

[1] Theoretical and empirical analyses of motivational (Hertel and Herrmann, 2003; Zeitlyn, 2003), cultural (Lancashire, 2001), organizational (Kuwabara, 2000; Tuomi, 2001; Moon and Sproull, 2000), economic (Lerner and Tirole, 2002; Bonaccorsi and Rossi, 2003) dimensions of F/OSS have been developed by large and heterogeneous communities of authors.

offers a complementary analysis of the design and the development of F/OSS in terms of a theory of modularity.

Modularity, which from the perspective of management scholars can still be regarded as an innovative manufacturing paradigm for the design and the production of complex artifacts (Schilling, 2000), is a key element in explaining the development and the success of many F/OSS projects, since it offers a comprehensive explanation of many key issues such as how division of labor takes place between developers, how coordination is achieved, how code testing and integration is deployed and how innovation occurs.

Our reconsideration of the accounts of GNU/Linux and other F/OSS projects highlights how they benefited from the typical advantages of implementing modular architectures (e.g. fast speed of development, recombination of modules, innovation through projects competition, reuse of previously developed code (Hatch, 2001; Jackson, 1998; Feller and Fitzgerald, 2000)), while, at the same time, many critical pitfalls typically related to managing modularity (the architectural design of modules and interfaces) were avoided. Let us summarize three interrelated strategies, or design shortcuts, that appear to have been particularly effective in this respect.

First of all, the architectural guidelines of many complex systems were clearly inherited from previously existing modular software projects (see, for instance, the GNU project and the FreeBSD project, closely resembling the UNIX architecture). By imitating a well–established architecture, developers were able to avoid the problems related to designing modular architectures from scratch, namely, devising a decomposition of the whole system in independent sub–parts, or modules (as we will see further on, this is not a trivial task).

Secondly, when devising modular architectures that are considerably innovative, so that it is not possible to rely on blueprints of existing software, another design shortcut is to think of modularity not in terms of a static and ex–ante design principle but, rather, as a dynamic activity of problem solving that starts from fairly interconnected architectures,

that are repeatedly fine–tuned and reworked, leading over time to more modular outcomes ("evolving modularization"). In this respect, we found it useful to analyze the evolution of the GNU/Linux kernel. Conversely, pursuing full modularity from the beginning may be very risky, and may eventually lead to serious difficulties (as in the case of the development of the HURD microkernel).

Finally, F/OSS development style seems to suggest a third effective design shortcut. While traditionally information hiding has been viewed as the key principle guiding both the design and the implementation of modular software artifacts, F/OSS seems to substantially disregard this principle at the implementation level. For instance, empirical evidence shows that F/OSS developers systematically improve parts of the project they are working on by tinkering with the code of multiple modules, taking advantage of the source availability and of the absence of code ownership.

Our account on how the principles of modular design have been originally adapted by F/OSS development allows us to move away from the stereotypical definition of modularity. As a matter of fact, while it is more and more often proposed as a fundamental paradigm for the design and production of artifacts (Baldwin and Clark, 2000), it is still regarded by some authors as a black box (Brusoni and Prencipe, 2001; Devetag and Zaninotto, 2001). The empirical domain of F/OSS allows to shed some light on many critical issues related to modularity. Among them: *(i)* the design of modular architecture viewed as a complex activity of problem solving, *(ii)* the relationships between organizational structure and architecture design in modular projects, *(iii)* how modularity copes with unforeseen interdependencies.

Our examination is based on published and unpublished data: interviews and papers written by key actors, analyses developed by scholars and quite a large mass of original documents made available through Internet websites.

The paper is organized as follows: Section 2 surveys some of the most relevant topics of modularity in management and organization science. Then, it turns to software

development, characterizing modularity as one of the fundamental topic in the software engineering debate. In Section 3, the F/OSS development is interpreted through the lens of the theory of complex modular system. We summarize the accounts of specific projects and we advance some stylized facts of F/OSS development. Finally, Section 4 sketches some reflections on how F/OSS methodologies and practices may fully benefit from employing a mindful modular approach to the design and implementation of complex software projects, and suggests how the peculiar implementation of the principles of modularity shown by F/OSS may help in refining both existing theories of modularity, and their practical application to domains different from software production.

## Modularity

Modularity has been receiving an increasing amount of attention in a variety of fields, from neuroscience and artificial intelligence to architecture, urban design and management (Baldwin and Clark, 2000). Nowadays a modular approach is applied to complex projects in R&D, industrial manufacturing and software engineering, and modularity has been assumed as a key–concept in the design and production of a great number of artifacts, both physical, like buildings, cars, furniture, and immaterial ones, like software (Schilling, 2000).

This interdisciplinary interest is largely due to the fact that modularity is regarded as a general property of complex systems, pertaining to the degree of decomposability of a system in loosely coupled sub–parts made by tightly coupled components. Literature on modularity emphasizes the importance of structures and relationships, and the outlined models all rely on an underlying system theory that provides a comprehensive framework for understanding and pertinently describes the specific object of study (artifacts, objects, machines, tasks, molecules, spaces, projects, etc.). A modular system is thus represented as a complex of components or sub–systems, where designers try to minimize and standardize the interdependencies among modules. As a matter of fact, speculations about

modularity in management science are addressed to both production (Baldwin and Clark, 2000; Langlois, 2002) and product (Ulrich, 1995; Schilling, 2000) domains.
Herbert Simon's influence in the way modularity has been conceived is particularly evident. First of all, modularity is often introduced within a problem solving framework and modular design is regarded as a solution to cope with uncertainty and variability. Second, as in Simon's analysis of the artificial, modularity in complex systems regards both goals and hierarchies. Third, modular solutions are based on problem decomposition; fourth, since complex systems are not quite entirely decomposable, modular design eventually needs to deal with residual interdependencies (Simon, 1981).

*Modularity in management and organization science*

In management and organization science literature, modularity has been introduced as an innovative paradigm for firm manufacturing (Ulrich, 1995; Schilling, 2000), organization design (Baldwin and Clark, 2000) and for a general theory of the firm (Langlois, 2002). Modularity provides relevant advantages that have been neatly identified in the literature. Modularity allows for product variety that is obtained by a recombination (mix and match) of components (Langlois and Robertson, 1992). Modularity is viewed as a base for differentiation strategies: firms may enrich their products catalog and adapt to customers' needs with limited additional costs (Camuffo, 2002). Moreover, modularity has also a great impact on production processes as it positively affects flexibility, division of labor and specialization (Devetag and Zaninotto, 2001).
According to Baldwin and Clark (2000), modularity in production systems is obtained by following some general rules, originally drawn from computer science and software development, concerning two different categories of information: visible and hidden information. Modular systems design needs to specify only the visible rules, namely the information about: *(a)* the definition of the architecture, *(b)* interfaces specifications and *(c)* modules integration tests. The inner description of each module and how it works are hidden from outside: it does not need to be defined ex–ante or communicated during the

process, since modules interactions exclusively follow the rules specified by the interfaces parameters.

Unfortunately, this neat description of modular design sometimes does not succeed; most of the times, after the integration of the independently developed modules, inconsistencies come up and the system does not work properly. The most common reason for this failure is that the decomposition of a complex system is not at all a trivial business. Most of the times, the activities of decomposition are suboptimal and result, at their best, in a quasi–decomposable architecture, where some degree of interdependency between modules is still at work. As we will see in the next Subsection, residual and unforeseen interdependencies seem to be particularly relevant in the production of software artifacts.

### *Modularity in software development*

#### *The "power of modularity" in software engineering.*

The notion of modularity is central in the design and production of software artifacts, especially for large and complex projects. Since the early days of software engineering the issue of designing, developing, testing and releasing a large software project brought into discussion the trade–off between simplicity and speed of development. The dilemma that software engineers were facing is, in the words of Brooks (1975), the following one: "For efficiency and conceptual integrity, one prefers a few good minds doing design and construction. Yet for large systems one wants a way to bring considerable manpower to bear, so that the product can make a timely appearance. How can these two needs be reconciled?"

Frederick Brooks, the author of one of the most influential software project management handbooks, clearly recognized that small sharp teams performed better than large ones, but they were not sufficiently staffed to deliver large software projects under schedule pressure. Conversely, while larger teams potentially increased the pace of the development process, they also resulted in an overwhelming need for coordination of individual efforts and in diminishing marginal returns of manpower on productivity (also known, in the

extreme case of negative marginal returns, as the “Brooks’ Law”). As a result, efficiency and conceptual integrity of the whole project were at risk, since men and months where not fully interchangeable units in the decision of dimensioning and staffing of a project.

Efficient software engineering methodologies are meant to solve this fundamental trade–off between task partition and division of labor, on the one hand, and coordination and communication costs, on the other one. Brooks’ recipe for coping with the design and the production of complex software was to vertically divide labor in order to separate high–level activities as much as possible (such as the design of a software artifact) from lower ones (such as the implementation of code). As a result, even a large software project might have been guided by a small number of architectural designers, hence reducing coordination and communication costs needed to conceive the architectural blueprint of the project. A second related element in Brook’s recipe was then to assign the implementation of each part of the project to small and focused teams (the so called “surgical team”).

In terms of a modern theory of modularity, the basic assumption inside Brook’s seminal work is that large software projects are integral and non–decomposable systems, where interactions among parts are nontrivial and generate high communication and coordination needs. Vertical division of labor is viewed as the way to avoid as much as possible these inefficiencies by concentrating design and architectural activities on few heads. What is clearly overlooked from Brook’s perspective is that interdependencies may not only be considered as given constraints, but rather they may be strongly reduced at the architectural design level, by effectively decomposing the complex system in quasi–independent subparts.

As a matter of fact, the introduction of a fully modular approach in modern software engineering methodologies has been fostered by the recognition that the degree of interdependencies may be strongly reduced if a complex software project can be decomposed in independent subparts, that is, dividing the whole project in smaller

components that are loosely coupled and highly independent on each other (von Hippel, 1990; Langlois, 2002). Hence, when subparts are almost independent, it is possible to divide labor minimizing the risk of coordination failures.

Conceiving the design of a complex software artifact as a modular system means to apply the basic principle of "information hiding", originally developed by Parnas (1972), that prescribes treating software modules as opaque entities, whose relevant information is only available to its inner programmer, while not being accessible to external programmers. Here the only information revealed is embedded in the interfaces, while the information regarding the design and how the module works is not communicated.

Nowadays, the widespread adoption of object oriented languages and the diffusion of component based development as well other popular trends in software engineering seem to have affirmed at large this information hiding principle and the paradigm of modularity as common software practices, aimed at speeding up the development process.

Consider, for instance, the case of the development of Microsoft Internet Explorer 3.0, Microsoft first internally developed Internet browser, that hit final product release less than nine months after the design of the first initial specifications.

As one developers described it:

"We had a large number of people who would have to work in parallel to meet the target ship date. We therefore had to develop an architecture where we could have separate component teams feed into the product. [. . . ] In fact if someone asked what the most successful aspect of IE3 was, I would say it was the job we did in 'componentizing' the product."[2]

The above discussion seems to push modularity and information hiding as the landmark principles for combining concerns of size and division of labor with high speed of development of a software project. Nevertheless, the"dark side" of modularity, namely the pitfalls of system integration and testing of modular design, seems to be particularly

[2]Cited in MacCormack (2001)).

substantial in the field of software artifacts.

*Modularity as a complex design activity: managing unforeseen interdependencies in software modules*.

A software product architecture may be defined as a mapping of required functions of the product in functional components. The system as a whole is decomposed in a set of functional modules whose interactions provide the overall functionality of the system (Ulrich, 1995; Sanchez and Mahoney, 1996). As in the case of hardware artifacts, one has to determine not only how to divide the whole system in subparts, and how to assign functional requirements to subparts, but also how any component has to communicate and interact with every other component in the architecture (Sanchez, 2000).

When it comes to the topic of component interactions, software seems to be a particular artifact with respect to physical artifacts, since components interface specification, defining interactions between modules, result in system of less rigid constraints in the case of software artifacts. In particular, physically specifications on how one component has to be connected to the other ones (*attachment interfaces*[3]), spatial, volume, weight constraints of a component (*spatial interfaces*) and other environmental interactions pertaining the generation of heat, vibrations, magnetic fields bearing consequences to other components (*environmental interfaces*), clearly do not apply to the case of software modules.

Then, at a first sight, it may be reasonable to expect that, given the existence of fewer sources of components interactions, designing and developing loosely coupled software artifacts would be easier than the case of hardware products such as standard physically assembled goods.

On the contrary, both software engineering literature (Brooks, 1975; Pressman, 2000; Schach, 2002) and empirical case studies of software product development (Glass, 1997; Solheim and Rowland, 1993) suggest that integrating software components may be harder

[3]We follow the taxonomy introduced by Sanchez (2000).

than assembling hardware artifacts.

Brooks' famous essay on the difficulties of software engineering techniques in granting improvements in productivity, reliability and simplicity in developing software programs, may support us in refining our explanations of why integrating software modules and thus producing modular software may be difficult (Brooks, 1986). The author speculates on the fundamental properties of software entities that may account for the difficulties in separating interdependencies and decompose large software projects: software entities differ from physical artifacts for their highly nonlinear complexity, leading to the impossibility of enumerating (not to mention understanding) all the possible states of a program. As the size of a software project increases, it becomes more and more difficult to decompose interdependencies and to design an architecture that preserves the initial conceptual integrity of the software project by a combination of loosely coupled functional software components. Moreover, software is invisible. The same intangible attributes that seem to free software entities from standard physical constraints that hardware ones have to satisfy, seem at the same time to affect human abilities of anticipating correctly component interface specifications and interdependencies. While geometric abstraction are powerful tools that may help the architectural design for assembly goods ("Contradictions become obvious, omissions can be caught." (Brooks, 1986)), similar geometrical representations do not help much during the design phase of software structures because source of interdependencies are more subtle, not visible, and related to a series of elements ("flow of control, flow of data, patterns of dependency, time sequence, name–space relationships" (Brooks, 1986)) whose interrelations may be only partially caught by diagrams and flow charts.

As a consequence, the process of modular software design tends to be a faulty one, where testing, debugging and integration phases may be much more relevant in terms of resources needed when compared to the production of physical artifacts. This is largely due to two intertwined aspects: *(a)* designers are boundedly rational decision makers

(Simon, 1957) and *(b)* the nature of interdependencies between modules is mainly multidimensional and invisible. As a result, the act of decomposing a large software project into components is an activity that results, at its best, in a suboptimal outcome: some sources of interdependencies are well determined and taken into account in the design of components and interfaces, while others are not. In some sense, even careful decomposition of large software projects tends to be accomplished making trade–offs between sources of interdependencies, recognizing the more visible ones and disregarding the less evident or less important ones. These reasons, combined with the huge size of large software project, account for the difficulties in the subsequent integration – testing – assembly phases. Likewise, less careful decomposition results in even greater problems at the final stages of code integration.

In the following, we will discuss how F/OSS style of development has benefited from adapting the paradigm of modularity. In particular, we will try to highlight and to relate success or failure of specific projects to advantages and strengths of modularity, on the one side, and to risks, pitfalls and drawbacks of modularity on the other hand. Furthermore, we will describe how F/OSS practices and methodologies represent an improved characterization of the paradigm of modular software production, where the information hiding principle is invoked at the design level, while it is later disregarded, at the implementation level, since all the available information is effectively used in order to speed up the production process, taking advantage of what Brooks originally defined "programming as a public process".

## Modularity in F/OSS development

### *Imitating a previously existing design*

Modular design of complex systems is a demanding job since all the modules interfaces have to be defined ex-ante. How can designers cope with such degrees of complexities? One of the lessons coming from the accounts of some well–known F/OSS projects is to

take advantage of existing templates, rather than to develop a brand new project from scratch. The Free Software Foundation (FSF) GNU project and the FreeBSD operating system project are two relevant instances of this approach to the modular design of complex artifacts, as their kinship with UNIX operating system is openly recognized.

UNIX operating system was a milestone in the computer software history and it is usually described as a highly modular, scalable and portable platform (Ritchie and Thompson, 1974; Gancarz, 1994). The UNIX architecture is a complex and massively decomposed architecture of independent modules, characterized by high specialization of programs ("programs that do one thing and do it well"), working together by means of structures, "pipes"[4], and sharing as a fundamental interface of communication text streams (also known as the "UNIX philosophy" as formulated by Douglas McIlroy, the inventor of pipes (Salus, 1994)). UNIX was the first modern operating system not developed using a hardware dependent assembly language. The kernel was written in C, ensuring portability to various hardware platforms (Johnson and Ritchie, 1978; Miller, 1978).[5]

UNIX highly modular architecture had strong consequences both at the level of developers coding activities and at the level of users' experience. Developers were able, thanks to its modular design, to carry out development of specific parts of the system in autonomy and without any need to coordinate their efforts with other sub–projects. Modularity allowed for both parallel development and contribution of new components; furthermore, the overall design of the system was significantly improved by the development of innovative modules and competition between similar projects (Baldwin and Clark, 2000). At the end–user level, modularity invited mere users to employ mix and match strategies (recombination of different modules), allowing them to generate a wide

[4]By pipe technology it is possible to connect the output of one program to the input of another one, and therefore to execute complex tasks by sequences of elementary programs linked together

[5]As originally noted by Baldwin and Clark (2000), another interesting feature of UNIX is represented by being modular not only at the architectural level (static design modularity) but also in the way, as an operating system, it deals with computer resources (dynamic design modularity).

variety of different implementations of the operating system where a large part of the modules pertaining to the user space were highly customizable and were chosen according to specific tastes or needs.

Even through this rather short and incomplete account of the early days of UNIX hackerdom, the past arguments should suggest that many of the elements pertaining to the decentralized and spontaneous nature of Linux development process are not as innovative and original as many Linux advocates often underline. They are rather mostly inherited from Linux direct ancestor, the UNIX operating system (McConnell, 1999). Strangely enough, this almost self evident argument seems to be mysteriously overlooked in many popular contributions to the Linux debate.

The GNU project, started in 1984 by Richard Stallman, represented at its beginning a titanic effort to offer a free alternative to currently existing commercial and proprietary operating systems. In this respect, Stallman's design strategy consisted in cloning an already existing project, a stable and mature architecture that had been originally conceived around fifteen years before. As suggested by Rosenberg (2000):

"Stallman says that he chose UNIX as his model because that way he would not have to make any design decisions."

As a matter of fact, the whole GNU project represented the attempt to recreate the pre–AT&T UNIX arcadic era, where the original architecture was preserved in essence and only some limited and marginal reworking in the design took place, in order to solve some minor technical disadvantages of UNIX (e.g. the introduction of 32–bit support). This architectural choice followed by Stallman, and later widely adopted by the hacker community, has been a conservative one. A more risky option such as undertaking a radically innovative project based on the design of a new architecture was disregarded in favor of a safe and well known alternative.

FreeBSD is another important operating system that deliberately mimicked the architecture of the UNIX operative system. Again, by adopting an existing architecture,

the community spent its attention on incremental development, rather than on design discussions (Jorgensen, 2001). To conceive a new operating system characterized by a modular architecture is a challenging cognitive activity of modules and interfaces definition. First, the designer needs to conceive a system of modules, by decomposing the whole system in quasi-independent components. Second, failures in the decomposition phase results in extra costs for fine–tuning and fixing activities aimed at solving unexpected and unforeseen interdependencies.

In this respect, the FSF and the FreeBSD community were able to consciously handle what, through the lens of the theory of modularity, is a fundamental trade–off between threats at the design level and opportunities at the implementation level. As a result, the decision of establishing the GNU and the FreeBSD projects upon a stable, mature and carefully modularized architecture was the key element to benefit from the typical advantages of modularity (concurrent engineering, division of labor, decentralized development, innovation via module based evolutionary dynamics, and much more), while at the same time avoiding the classic pitfalls and drawbacks of modularity, concerning the risks of imperfect decomposition in the design of an innovative modular architecture as the backbone for the project.[6]

*Horizontal division of labor, task interdependencies and Brooks' Law*

The perspective of modularity seems also to offer other different interpretations contrasting many other recurring stereotypes in the debate over the revolutionary nature of F/OSS development.

One of the most criticized principles of the otherwise seminal and evocative essay *The Cathedral and the Bazaar* (Raymond, 1999) is the one prefiguring the demise of Brooks' Law within F/OSS development. This view is supported by a *reductio ad absurdum* argument, claiming that, if Brooks' Law were at work, it would not be possible to observe

[6]See also further on how, in the case of the GNU project, the failure to correctly modularize the architecture resulted in serious troubles for the developers of the HURD micro–kernel.

such a thing as Linux development. Conversely, the observation of the Linux case study suggests to the author that the effects of Brooks' Law may be overcome by other forces, such as the project leader's capabilities in attracting, motivating and coordinating a team of skilled and talented developers, in a distributed process strongly facilitated by Internet connectivity as a shared medium of communication. This argument, that Brooks' Law does not apply to Internet–based distributed development, has been widely criticized by many authors (see for instance Bezroukov (1999); Jones (2000)).

Modularity allows us to refine and clarify these criticisms suggesting that a large number of participants in a project may be not a sufficient condition to generate dysfunctional effects such as diminishing or negative marginal return of manpower to productivity. The key aspect, in this respect, is represented by the degree of task interdependency between the various members belonging to the project. Thus, the high productivity experienced in the GNU/Linux development is interpreted as largely due to the massively modularized structure of the project, enabling the existence of highly independent sub–projects joined by a limited number of developers, resembling in essence the theory of Brooks' surgical (small, skilled and focused) team (Brooks, 1975), while the role of the Internet in this interpretation is of mere medium of exchange allowing distant communication.

Actually, our latest claim seems to be straightforward if we look at a typical sub–project within the GNU/Linux architecture. Furthermore, if we look more generally at the world of F/OSS projects, there is growing empirical evidence showing that the number of participants involved in a project is on average very small (Krishnamurthy, 2002; Capiluppi et al., 2003). Despite this, in some specific cases, such as in the development of the kernel for the GNU operating system, that has been undertaken thanks to the coordinated effort of hundreds of contributors, we need to clarify our point and to address the relationships between Brooks' Law and division of labor in the case of vertical division of labor.

*Vertical division of labor and organization and architectural ladders*

Another rather famous postulate in Raymond's *The Cathedral and the Bazaar* is the following:

"I had been preaching the UNIX gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time. Linus Torvalds's style of development – release early and often, delegate everything you can, be open to the point of promiscuity – came as a surprise. No quiet, reverent cathedral–building here – rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from anyone) out of which a coherent and stable system could seemingly emerge only by a succession of miracles." (Raymond, 1999)[7]

While finding this quote intriguing and insightful in many senses, since it clearly describes the evolutionary dynamics nature of GNU/Linux development (Kuwabara, 2000), we also are convinced that it conveys many misleading interpretations of the F/OSS phenomenon as a whole. Many authors have criticized the cathedral versus bazaar metaphor. We hereby are particularly concerned with a serious and common misinterpretation of this metaphor when it comes to the topic of the architectural characteristics of GNU/Linux.

The misinterpretation of the above quote runs, slightly simplifying, as follows: GNU/Linux comes out of the blue from a chaotic mess of contributions and organizes itself as a coherent system in an apparently self–regulating way, showing a mysteriously spontaneous order. This emergent view of the genesis of GNU/Linux is misleading in that it suggests the existence of a deregulated and emergent flat architecture. In contrast, we

[7] See also Subsection 3.6 for related comments on these statements.

claim that the modular architecture of GNU/Linux is characterized by being quite hierarchical, rather than flat.

Basically, it boils down to the distinct possibility of distinguishing at least two different and hierarchically ordered ladders in GNU/Linux: a higher level, the kernel space, and a lower one, the user space. As it happens, the celebrated babbling bazaar, representing the decentralized and anarchic distributed process, takes place at the user level and it is fostered by the highly modular architecture, as described formerly. Conversely, at the higher inner level of the operating system, the development process seems to be rather different: Linux inner core started to be developed as a one–person hack and only at a subsequent stage of the process contribution from other developers were introduced. Moreover, while contributions to the kernel represent an open process, the integration of code within the kernel has been a process firmly regulated by the same Torvalds, at the beginning, and later supported by a small group of "trusted lieutenants" (Franck and Jungwirth, 2002; Dafermos, 2001). In the next Subsection this process will be described in much more detail, here we are specifically concerned of describing its consequences at the organizational level. In order to preserve integrity and coherence within the most important and complex part of the system, at the kernel space ladder all initial relevant design decisions were largely taken by Torvalds and by an inner team of developers. The same holds for most of the subsequent activities of kernel development. While one has to acknowledge the role of code contribution from the bottom (the hacker community), it is also indisputable that its incorporation in the project has been fueled by a highly structured and hierarchical process of review and selection (albeit not based on formal authority but rather on competence and reputation).

Sanchez and Mahoney (1996) where the first to highlight a basic feature of modular product architectures, namely the isomorphic relationship between product architecture and organization traits. This seems to be indeed the case for GNU/Linux that emerged as a stable system not by a succession of miracles, but rather by exploiting modularity at the

user space level, encouraging decentralization, and carefully crafting and controlling the overall consistency of the design at the kernel space, imposing a cathedral–like hierarchy in code evaluation and integration.

To summarize our point, we find the cathedral vs. bazaar distinction seriously misleading. Hence, if one really wants to compare the GNU/Linux architecture to a bazaar–like structure, he should not look at an ordinary bazaar, but rather at Kapali Carsi, Istanbul Grand Bazaar, the oldest (15th century) and largest (over 4400 shops on 30 hectares of land) marketplace of the world. The most prominent and uncommon feature of this marketplace is that it is not uncovered and out in the open as usually bazaars are. On the contrary, it is a covered structure owning a complex architecture protecting a giant labyrinth of shops and various commercial activities. It has been observed by many that the covered architecture is a fairly regular structure, which makes the underlying bazaar even more maze–like and confusing in practice. Just as the building architecture is not affected by the underlying bazaar activities, likewise, GNU/Linux higher ladder, i.e. the kernel, is largely shielded from decentralized evolutionary dynamics happening at the user space level.

We have until now emphasized that the GNU operating system is a massive modular architecture, mostly inherited from a previous design and characterized by a hierarchical two–ladder architecture that hardly resembles the flatness of the common bazaar at all. To further refine our analysis we need to admit that, albeit largely based on the UNIX architecture, there does exist something truly innovative and original in the GNU operating system. This pertains to its kernel. In the following Subsection we reflect on its origins, highlighting the different approaches on modularity and interdependencies decomposition followed by two different competing projects: the Linux project and the HURD project.

*Ex–ante modularity versus evolving modularization: the development of a kernel for the GNU operating system*

Literature both in management and in computer science has clearly pointed out the pros and cons of modular design and we have already discussed the undervalued difficulties that designers face when they invent modular architectures for complex systems. Along with Simon's perspective, it has been shown that the decomposition of complex problems in nearly–independent sub–problems (i.e. modules) is a complex activity itself (Marengo et al., 2001). At the beginning, designers do not know precisely how to conceptualize the modules of new artifacts; later, when a first conceptualization is reached, they still vaguely know how good is the chosen architecture, compared to the other that have not been considered.

If we underestimate the problems posed by modules identification and decomposition of new architectures, we hardly understand why modular design of complex products is so difficult and unpredictable. Another way to grasp this issue is to consider that many modular products were originally developed from interconnected solutions. While this is not a general rule, it was definitely true for Linus Torvalds's kernel: the GNU operating system is known for being a modular complex artifact and its successful development, accomplished by a distributed community of hackers, largely benefited from that. Therefore, it may be surprising that its core–component, the so–called kernel, was initially conceived as a highly integrated product and only eventually acquired a modular structure.

As a developer, Linus Torvalds' major effort to the project afterward called GNU/Linux was aimed to conceive and write the kernel, that is the core part of the operative system that could use all the applications and the libraries of software that had already been

developed within the GNU project.[8]

At the time Linus Torvalds started to work on his kernel, a long debate was mounting around the advantages offered by an alternative architecture, called microkernel, designed to work in all possible and different processors.[9] Compared to traditional, hardware dedicated kernels, microkernels appeared to be more complex and less efficient. They were more complex because even simple problems were treated as instances of general tasks that might have involved a higher number of specifications and instructions to interact with other parts of the kernel; therefore, they might have resulted to be less efficient as they did not take advantage of specific features of the hardware they run on. While microkernel architecture appeared to be a better solution because of its recognized technical superiority, Torvalds decided to develop his kernel in less general terms, thinking that microkernels at the beginning of the '90 were still experimental and too complex projects (at that time Microsoft was developing its new Windows NT using a microkernel structure) and they were exhibiting a much worse performance (Torvalds, 1999). By the way, when Torvalds started to work on its kernel the Free Software community and the GNU partisans were already involved in the development of a microkernel (called HURD), even though f the task seemed to be much far away from its conclusion.

Therefore, the very first version of Linus' kernel had a monolithic structure and was also extremely hardware specific, since it was conceived for working on Intel 80386 processors

[8]By the time Torvalds started to conceive the Linux kernel, the GNU project had developed to the stage of an almost complete free operating system, including all the major system components, such as terminals, assemblers, compilers, interpreters, debuggers, text editors, mailers, and many more, but the fundamental one: the kernel.

[9]As Torvalds put it "When I began to write the Linux kernel, there was an accepted school of thought about how to write a portable system. The conventional wisdom was that you had to use a microkernel–style architecture." (Torvalds, 1999). See also the well–known "Linux is obsolete" flamewar in the comp.os.minix newsgroup (reported in Appendix A of DiBona et al. (1999)), where Linus Torvalds, Andrew Tanenbaum and other relevant hackers passionately debated on OS design issues and on the strength and weakness of micro versus monolithic kernels.

only. The first effort to port Linux kernel to another processor (Motorola 68K) showed all the drawbacks of having a hardware–specific architecture, since the developers of 68K Linux had to write another hardware–specific kernel from scratch. When Torvalds started to think about porting Linux to the Alpha platform, he realized that the original design was no longer effective and in 1993 he started to rewrite the kernel code completely. He decided to keep a monolithic architecture, but he introduced some degree of modularity in the system design, in order to simplify the portability task and to incentive parallel development in some less critical parts of the system. In Torvalds words:

"With the Linux kernel it became clear very quickly that we want to have a system which is as modular as possible. The open–source development model really requires this, because otherwise you can't easily have people working in parallel. It's too painful when you have people working on the same part of the kernel and they clash.

Without modularity I would have to check every file that changed which would be a lot, to make sure nothing was changed that would effect anything else. With modularity, when someone sends me patches to do a new filesystem and I don't necessarily trust the patches per se, I can still trust the fact that if nobody's using this filesystem, it's not going to impact anything else."(Torvalds, 1999)

Therefore, the general kernel model made use of modules and it was conceived bearing in mind those elements common to all typical modular architectures (even though it was not as rigorous and general as microkernels are). Following this scheme, Torvalds could deal with them separately and confine all the hardware–specific pieces of code in modules out of the core kernel (de Goyeneche and Apolinario Fernández de Sousa, 1999). These modules could be later updated or changed by Torvalds himself and by the other Linux developers with no effect on the kernel core.[10] Device drivers structure is a good example of the *third way* followed by Torvalds. One extreme solution is to put all the hardware

[10] Version 2.1.110, released in July 1998, counts around 1,5 million lines of code: 29% is the kernel and the file systems, 54% are platform–independent drivers and still 17% is architecture–specific code.

specific into the core kernel: this is easier to do, it increased the performance, but the kernel is totally unportable. The other extreme solution, consistent with the microkernel design, urges to leave all the specific in the user space, which declines the performance and the stability of the system.

In later discussions Torvalds explained the reasons for its choice: a fully modular architecture, like the one adopted for HURD, would have posed problems to a degree of complexity that it could have compromised the accomplishment of the project. To avoid such risks and keep the degree of complexity of the project as low as possible, Torvalds decided to design a monolith and he actually wrote all the architectural specs himself,[11] avoiding all the problems related to collective projects (e.g. division of labor, coordination, communication).

On the other hand, the HURD micro–kernel, a project in direct competition with the Linux kernel, has paid for the choice of pursuing a fully modular approach from the beginning in terms of the continuous delays that have plagued its development. Nowadays, it is still under active development and still lacks the stability and performance assured by the Linux kernel.

The validity of Torvalds choice is under our eyes and it is difficult to overestimate the consequences of this modular solution with regard to the subsequent portability and extensibility of the system trough the distributed effort of the community. Nowadays Linux run on an increasing number of computers, from workstations to handheld devices and its development is assured by the effort of tens of thousands developers in the world. Torvalds and a few other people close to him control the kernel core and have the final word in the decisions related to the development of the system. Other developers, on the other hand, offer their contribution to improve and upgrade the system.

We already showed how critical were the consequences of inheriting a modular UNIX–like architecture based on complementary and interconnected components. To a

[11] Releasing version 0.11 in December 1991, he credited on three other people.

more hidden and critical level the development of the core of the operative system, the kernel, followed an analogous destiny. The modular structure adopted by Torvalds for its kernel happened to be successful, nevertheless it does not prevent the system from the emergence of unforeseen interdependencies within the modules that may arise with the future development of hardware and software. While HURD established itself as an attempt to develop a fully general and modular system, Linux kernel took advantage of some architectural shortcuts: as it is, the problem related to emergent interdependencies that were not expected at the beginning may become a problem for the future enduring success of Linux, even though this can be regarded as a future cost for the straightforwardness of its design. Some of these emergent interdependencies may be solved by tinkering, reworking and re–designing (Ratto, 2003); sometimes the adopted solutions are not adequate and the communities of developers that do not agree with the final decision may introduce alternative versions of the system. These forks may express a coordination failure when a community does not converge on a unique satisfying solution. Further, unanticipated interdependencies may end up in more serious problems than just forks, as it happens when the existing operative systems reveal itself to be inconsistent with the architecture of new processors. Torvalds himself is fully aware of this situation when he describes a future scenario of Linux's decline:

"They'll say Linux was designed for the 80386 and the new CPU's are doing the really interesting things differently. Let's drop this old Linux stuff. This is essentially what I did when creating Linux. And in the future, they'll be able to look at our code, use our interfaces, and provide binary compatibility, and if all that happens I'll be happy."(Torvalds, 1999)

It is worthwhile to point out some observations that are suggested by this story:

- even when task partitioning and division of labor issues do not really matter, the design of modular architectures from scratch may reveal to be an extremely complex task; therefore, designers may prefer integrated solutions that are easier to devise and handle;

• a modular architecture is more vulnerable to design faults, especially when the task is complex and the amount of resources are limited; in particular, an ineffective definition of modules that are not loosely–coupled enough produces an increasing amount of interdependency, rather than its opposite. As a result, individuals, rather than groups of developers may more efficiently accomplish the early stages of new projects. Some successful F/OSS stories experienced this destiny, as they have been started as one-man projects aimed to solve specific problems and eventually evolved in structured projects involving a large number of people (e.g., Sendmail was initially developed by Eric Allman to route email for other users within UC Berkely, Perl by Larry Wall to solve some annoying problems in system administration, World Wide Web by Tim Berners-Lee as group environment for academic information sharing among high-energy physicists, an so on).

• Torvalds' kernel story enriches the perspective offered by the Conway's law about the isomorphic structure of product and process (Conway, 1968). Modularity, in facts, seems to be pursued not as a dogmatic feature of the product, but it arises as a general design rule and it is boosted only when it provides some direct advantage. Therefore, the evolution of the Linux kernel towards modular design suggests that it is possible to combine together, under the same architecture, both modular components and integrated parts. Later on, the designers may introduce a higher degree of modularity by adapting the originally interconnected architecture. In other words, modularity arises more as a process of evolutionary design (modularization), rather than as an ultimate ex–ante property of an artifact.

*Beyond the principles of modularity*

In the previous Subsections, we have argued that the paradigm of modularity has a great explanatory power in characterizing the F/OSS development style and the success of many software projects: well–decomposed architectures seem to reconcile considerations about division of labor and size of a project with concerns of high speed of development.

Nevertheless, as mentioned earlier, for complex software artifacts it may be almost impossible to separate *ex-ante* all interdependencies, and unforeseen coupling between components at later stages (like for instance, integrating new and existing modules) may strongly affect the final outcome of the process. We argue that F/OSS development style has originally adapted the principles of modularity in order to lower the impact of this "dark side" of modularity.

It is worth mentioning that many scholars have radically criticized the modular approach to the design of software artifacts since its introduction. As noted by Brooks (1995):

"Harlan Mills has argued pervasively that 'programming should be a public process', that exposing all the work to everybody's gaze helps quality control, both by peer pressure to do things well and by peers actually spotting flaws and bugs".

Brooks argued that information should be completely available in order for failures in the design of software to become evident and be corrected (Brooks, 1975). Conversely, in accordance with the principles of modularity, these processes of peer review, control and contribution to others' source code are strongly limited by information hiding constraints, since modules are not available to other developers.

Despite these criticisms, information hiding has nowadays become almost ubiquitous in software engineering. Even Brooks (1995), in the 20th year anniversary edition of his *The Mythical Man–Month*, admits the following: "Parnas was right, and I was wrong on information hiding".

We claim that the fundamental innovation of F/OSS practices lies in how the basic postulate of information hiding is adapted to overcome these pitfalls, suggesting a step further in the software engineering debate on the pros and cons of modularity. While information hiding is clearly at the core of designers' activities when initially decomposing a software project in modules, the same principle is later disregarded, at the implementation level, in day by day coding, test and integration activities. As a matter of fact, in the F/OSS community, hackers actually are overexposed to, rather then shielded

from, a huge amount of code.

The free availability of the source and the absence of code ownership make programming a truly public process, since good coding solutions are shared and adapted to solve similar problems (Pavlicek, 2000), and *ex–post* interdependency conflicts are handled by employing a wider set of fine–tuning strategies. A well–known feature of F/OSS methodologies is parallelized and distributed code debugging, where bugs are highlighted and corrected by others' "eyeballs" (Raymond, 1999; Iannacci, 2003). Kuan (2000), for instance, shows that F/OSS has a higher rate of quality improvement than closed source software. Similar results are obtained by Succi and Eberlein (2001). Jorgensen (2001) reports that half of the respondents to his research survey claimed to have received a bug report from someone else within the previous month and nearly half of them credited an external contributor fixing a bug in their code. Likewise, at the code review level, similar parallel and distributed processes of peer review highlight design incoherencies introduced by others.

In other circumstances the "no hiding" principle allows developers to undertake much more sophisticated software engineering activities, such as redefining modules and interfaces specifications in response to the emergence of new interdependencies between separate modules. This is often the case in the introduction of radically new or substantially complex features in stable projects. For instance, the introduction of cryptography in the *Freenet* project (von Krogh et al., 2003) affected many different modules and demanded the whole redefinition of the architecture. In the worlds of developer #101:

"[...] unfortunately, any change you make in that affects not only the protocol, which is what I am working on right now, but it affects how the keys are handled (Module 4), how the client interprets the keys (Module 8), how data is verified. Basically, that little change affects pretty much everything in Freenet and, therefore, the kind of people making those changes, myself and (developer #6) mainly, have to understand everything that happens in

Freenet in order to do it."

The availability of other modules source code is what allowed the two developers to disentangle the complex web of interdependencies introduced by adding a public key to cryptography. Similarly, Jorgensen (2001) underlines how the free flow of information about the whole project helped FreeBSD developers to introduce a radically innovative feature to support multiprocessing (Symmetric Multiprocessing) within a mature software architecture.

In short, the lesson of F/OSS development is the following: since it is impossible to design ex–ante a zero–defect software architecture, it is worthwhile to embrace adaptive and flexible strategies that ease modules integration by using all the available (not anymore hidden) information.

The no–hiding policy bears one additional consequence: it does not only elicit an iterative and distributed process by which previously written code is fine–tuned and optimized by participants in the software community, but it also makes it possible for individual hackers or entire groups to write patches or variations of the original code that are not completely compatible with previous work carried out in the same software project or with respect to other related pieces of software. While incompatibilities are most of the time unintentional and marginal and may be fixed by subsequent coding activities, sometimes these modifications are large and/or intentional and may result in forking, i.e. the introduction of an independent and partially incompatible version of the original software.

As a matter of fact, within the software industry, advocates of corporate closed source software development have argued that, due to the lack of code ownership, F/OSS seems to be particularly prone to develop "multiple incompatible versions of programs, [plagued by] weakened interoperability, [and] product instability" (Mundie, 2001). With respect to software development activities, this may lead to duplication of efforts and may result in an inefficient allocation of scarce resources at the level of the whole F/OSS community.

Nevertheless, other studies have suggested that forking in F/OSS may be much less

frequent than one might expect at a first glance, and may eventually lead to positive, rather than catastrophic, outcomes. Many F/OSS projects have a governance structure (ranging from the project leader benevolent dictatorship to the formation of complex coalitions) that prevents attempts to fork (Kogut and Metiu, 2001). Moreover, the widespread diffusion of the GNU General Public License (GPL), seems to mitigate the incentives to fork an existing F/OSS project since, in essence, it prevents the appropriability of innovations. In fact, while anyone may fork any software project at any time, his subsequent work, due to the GPL "infectious" nature, would be available to the whole community as well. Thus, others may take advantage of the improvements of the fork. In this perspective, forking rarely happens and even when it occurs, this often translates in being beneficial to both competing projects, since the GPL allows each one to study the other and implement the most innovative features (e.g. this seems to have been the case in the rivalry between the *Emacs* and the *XEmacs* projects (Moen, 2003)). As a result, forking seems to take place largely in case of ultimate and irreconcilable differences in views and priorities in the development of a software project, and forks take off and succeed only if they are able to occupy different ecological niches (see for instance the existence of various GNU/Linux distributions), thus offering specialized solutions for a differentiated audience (van Wendel de Joode et al., 2003). Finally, it has been noted that it is not uncommon for forks to merge back with the original project as benefits and drawbacks of "running alone" may change overtime (as in the case, for instance, of the *egcs* project, re–merged by the FSF with the original *gcc* project in 1999) (Moen, 2003). Anyway this topic calls for more rigorous and analytic case studies aimed at understanding better the advantages and drawbacks in the emergence of forking within F/OSS projects.

## Discussion

In the end, modularity may be conceived as simple as it is, as long as we do not open the black box and keep track of the organizational processes behind the structure. Most

quoted contributions in management studies (Baldwin and Clark, 1997, 2000; Ulrich, 1995; Sanchez and Mahoney, 1996) unfold a neat and smooth theory of modularity, introduced as a cornerstone for artifact design.[12] According to this Olympic version, modularity is defined as a "particular design structure, in which parameters and tasks are interdependent within units (modules) and independent across them" (Baldwin and Clark 2000, p. 88). Unfortunately, this perspective underestimates that the decomposition of complex systems generally resolves on a quasidecomposition and not in a full decomposition, as some interdependencies may not be predicted or are left out on purpose, simply because they are regarded as marginal ones. Our reconsideration of the development of some F/OSS projects show how the modularity principles may in practice differ from what the theory prescribes. GNU/Linux and, more generally, F/OSS represent an instance of unorthodox modularity: the information hiding principle is significantly disregarded as the artifact evolves mainly through a repertoire of practices (e.g. peer coding and debugging, frank discussions, open decisions) where developers and users work apart, tinkering and patching the original modular product and, overall, violating another of the law of the Olympic modularity stating that the only available operators are represented by manipulation at the module level (splitting, substituting, augmenting, excluding, inverting, porting, Baldwin and Clark (2000, pp. 123–146)).

In our view, reading the GNU/Linux case according to the modularity perspective provides a complementary understanding of the F/OSS phenomenon and, at the same time, offers some insights to think about the way we conceive a theory of modularity for complex systems.

With respect to the first issue, taking advantage of existing architectures like UNIX and related standards (e.g. POSIX) it has been a successful strategy as the community of developers avoided to design a modular structure from scratch. The comparison between the HURD project and Torvalds monolithic kernel shows that to develop decomposable

[12] For an insightful assessment of this topic see also Langlois (2002) and Devetag and Zaninotto (2001).

architectures for complex products exposes the designers to the risk of unforeseen interdependencies that may ultimately endanger the whole project. Besides, as F/OSS projects are developed by distributed organizations and the community members communicate only remotely, coordination and collective decision making seem to be two fundamental issues in F/OSS development. In other words, our study of F/OSS projects through the lens of the theory of modularity outlines three main strategies that characterize the design and the development of complex systems: *i)* inheriting existing modular architecture, *ii)* evolving towards increasing degrees of modularity and *iii)* violating the information hiding principle. This repertoire of practices, or shortcuts as we called them in our introduction, emerge as effective and robust routines that seem to fit very well with the actors involved, i.e. distributed communities of developers, and the problem solving activity they embrace.

GNU/Linux case, on the other hand, suggests some general reflections on modularity and modularization. F/OSS developers exploit all the advantages of a modular architecture as the massive parallel activity within modules/programs witnesses; on the other hand, the modularization does not stop with the architecture design. The unforeseen interdependencies that come to the surface as the operative system evolves, revealing some inconsistencies, are met by violations of the information hiding principle.

In questioning how this experience may be extendible to other contexts where modularity has already started to represent a promising approach, there are at least two fundamental conditions that need to be clearly spelled out.

First, F/OSS distinctive trait is represented by the open access to knowledge (source code and documentation) stored in the modules. In the F/OSS world, imitation and copy are encouraged and protected by a reverse form of copyright (copyleft). According to the Economics of Innovation standard models, copyleft should inhibit any investment in innovations, since anybody may take advantage of any innovation and there are no incentives for the innovators. F/OSS apparently contravenes this rule and this is way

motivational analyses based on various perspectives, i.e. psychological, cultural, sociological, seem to be urgent to support an economic explanation of this phenomenon. To our viewpoint, the apparent paradox of compelling innovations in a copyleft regime is due to the second fundamental condition that characterizes the F/OSS movement, that is a deep overlap between producers and users. At least, at the beginning, most users were developers or had some skills that allowed them to perform successful adaptations. Again, most of the traditional ways to conceive innovation and product development in other domains keeps producers and users separated, even though today customers are more and more often directly involved in the definition of their own product.

As long as developers and users communities deeply overlap, copyleft regime does not inhibit innovation, but rather it ensures its open and free diffusion. On the other hand, when the communities start to be more and more different from each other, when developers are viewed as producers and users as customers, the natural system of reciprocal benefits becomes less and less salient. Therefore, looking for a possible generalization of F/OSS experience should push us towards other economic contexts where developers and users are able to establish strong relationships; in this respect, settings where customers actively participate in the development of new products (see for instance von Hippel (1998)) seem to represent a promising milieu for empirical investigation.

## Acknowledgments

The authors would like to thank ROCK (Research on Organizations, Coordination and Knowledge) members at the University of Trento, the participants to the track session “Modularity and division of innovative labour: design, organisation and cost analysis” at EURAMs 2nd conference on Innovative Research in Management, May 9-11, 2002, in Stockholm, Stefan Koch and three anonymous referees for their helpful comments on earlier drafts of this paper. The usual disclaimer applies. Financial support from MIUR under the projects COFIN 99 (Innovating by modular projects, division of labor and compatibility

standards: models of organization and industrial dynamics) and COFIN 02 (Language and coordination in managerial distributed decision making) is gratefully acknowledged.

## References

Baldwin, C. Y. and Clark, K. B. (1997). Managing in the age of modularity. *Harvard Business Review*, 75(5):84–93.

Baldwin, C. Y. and Clark, K. B. (2000). *Design Rules. Vol. I: The Power of Modularity*. Cambridge, MA: The MIT Press.

Bezroukov, N. (1999). A second look at the cathedral and bazaar. *First Monday*, 4(12).

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

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

Brooks, F. P. (1986). No silver bullet. In Kugler, H. J., editor, *Information Processing 1986, Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–1076. Amsterdam: Elsevier Science.

Brooks, F. P. (1995). *The Mythical Man–Month. Essays on Software Engineering, Anniversary ed.* Reading, MA: Addison Wesley.

Brusoni, S. and Prencipe, A. (2001). Unpacking the black box of modularity: Technologies, products and organisations. *Industrial and Corporate Change*, 10(1):179–205.

Camuffo, A. (2002). Rolling out a “world car”: Globalization, outsourcing and modularity. 2nd EURAM Conference, Stockholm, Sweden.

Capiluppi, A., Lago, P., and Morisio, M. (2003). Characterizing the oss process: a horizontal study. 7th European Conference on Software Maintenance and Reengineering, Benevento.

Conway, M. (1968). How do committees invent. *Datamation*, 14(10):28–31.

Dafermos, G. (2001). Management and virtual decentralised networks: The linux project. *First Monday*, 6(11).

de Goyeneche, J. and Apolinario Fernández de Sousa, E. (1999). Loadable kernel modules. *IEEE Software*, 16(1):65–71.

Devetag, M. and Zaninotto, E. (2001). The imperfect hiding: Some introductory concepts and preliminary issues on modularity. DISA Working Paper, Università degli Studi di Trento.

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

Feller, J. and Fitzgerald, B. (2000). A framework analysis of the open source software development paradigm. In *Proceedings of the twenty first international conference on Information systems*, pages 58–69. Atlanta, GA: Association for Information Systems.

Franck, E. and Jungwirth, C. (2002). Reconciling investors and donators. the governance structure of open source. Lehrstuhl für Unternehmensführung und–politik Universität Zürich.

Gancarz, M. (1994). *The UNIX Philosophy*. Newton, MA: Digital Press.

Glass, R. (1997). *Software Runaways. Lessons Learned from Massive Software Project Failures*. Upper Saddle River, NJ.: Prentice Hall.

Hatch, N. (2001). Modular stepping stones along the firm's technology path. Nelson and Winter Conference Aalborg.

Hertel, G. N. S. and 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.

Iannacci, F. (2003). The linux managing model. Proceedings of the Third International Conference on Open Source, ICOS 2003.

Jackson, I. (1998). Why is software freedom uselful and what does it mean? SANE'98 (18-20 November 1998).

Johnson, S. and Ritchie, D. (1978). Portability of C programs and the UNIX system. *The Bell System Technical Journal*, 57(6):2021–2048.

Jones, P. (2000). Brooks' law and Open Source: The more the merrier? Retrieved Jan 2, 2003, from http://www-106.ibm.com/developerworks/library/merrier.html.

Jorgensen, N. (2001). Putting it all in the trunk: incremental software development in the Free BSD open source project. *Information Systems Journal*, 11(4):321–336.

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

Krishnamurthy, S. (2002). Cave or community? an empirical examination of 100 mature Open Source projects. *First Monday*, 7(6).

Kuan, J. (2000). Open source software as consumer integration into production. Retrieved Jul 12, 2003, from http://opensource.mit.edu.

Kuwabara, K. (2000). Linux: A bazaar at the edge of chaos. *First Monday*, 5(3).

Lancashire, D. (2001). Code, culture and cash: The fading altruism of open source development. *First Monday*, 6(12).

Langlois, R. N. (2002). Modularity in technology and organization. *Journal of Economic Behavior & Organization*, 49(1):19–37.

Langlois, R. N. and Robertson, P. (1992). Networks and innovation in a modular system: Lessons from the microcomputer and stereo component industries. *Research Policy*, 21(4):297–313.

Lerner, J. and Tirole, J. (2002). The simple economics of Open Source.

MacCormack, A. (2001). Product–development practices that work: How internet companies build software. *Sloan Management Review*, winter:75–84.

Marengo, L., Pasquali, C., and Valente, M. (2001). Decomposability and modularity of economic interactions. In Callebaut, W., editor, *Modularity: Understanding the Development and Evolution of Complex Natural Systems*. Cambridge, MA: The MIT Press.

McConnell, S. (1999). Open–source methodology: ready for prime time? *IEEE Software*, 16(4):6–11.

Miller, R. (1978). UNIX – a portable operating system? *ACM Operating Systems Review*, 12(3):32–37.

Moen, R. (2003). Fear of forking essay. Retrieved July 1, 2003, from http://linuxmafia.com/Erick/essays/forking.html.

Moon, J. Y. and Sproull, L. (2000). Essence of distributed work: The case of the Linux kernel. *First Monday*, 5(11):1–20.

Mundie, C. (2001). The commercial software model. Retrieved July 1, 2003, from http://www.microsoft.com/presspass/exec/craig/05-03sharedsource.asp.

Parnas, D. L. (1972). On the criteria for decomposing systems into modules. *Communication of the ACM*, 15(12):1053–1058.

Pavlicek, R. C. (2000). *Embracing Insanity: Open Source Software Development*. Indianapolis, IN: Sams Publishing.

Pressman, R. (2000). *Software Engineering : A Practitioner's Approach, Fifth ed.* Boston, MA: McGraw Hill.

Ratto, M. (2003). Re–working by the linux kernel developers. Department of Communication, University of California, San Diego.

Raymond, E. S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly & Associates.

Ritchie, D. and Thompson, K. (1974). The UNIX time–sharing system. *Communications of the ACM*, 17(7):365–375.

Rosenberg, D. K. (2000). *Open Source. The Unauthorized White Papers*. Foster City, CA: IDG Book Worldwide.

Salus, H. P. (1994). *A Quarter Century of UNIX*. Reading, MA: Addison–Welsey.

Sanchez, R. (2000). Modular architectures, knowledge assets and organizational learning: New management proceses for product creation. *International Journal of Technology Management*, 19(6):610–629.

Sanchez, R. and Mahoney, J. T. (1996). Modularity, flexibility, and knowledge management in product and organizational design. *Strategic Management Journal*, 17(winter special issue):63–76.

Schach, S. (2002). *Object Oriented and Classical Software Engineering, Fifth ed.* Boston, MA: McGraw Hill.

Schilling, M. A. (2000). Toward a general modular systems theory and its application to interfirm product modularity. *Academy of Management Review*, 25(2):312–334.

Simon, H. (1957). *Models of Man*. New York, NY: Wiley.

Simon, H. A. (1981). *The Sciences of the Artificial, 2nd ed.* Cambridge, MA: The MIT Press.

Solheim, J. and Rowland, J. (1993). An empirical study of testing and integration strategies using artificial software systems. *IEEE Transactions on Software Engineering*, 19(10):941 –949.

Succi, G. Paulson, J. and Eberlein, A. (2001). Preliminary results from an empirical study of open source and commercial software products. International Conference on Software Engineering, Toronto, Ontario, Canada.

Torvalds, L. (1999). The Linux edge. In DiBona, C., Ockman, S., and Stone, M., editors, *Open Sources: Voices from the Open Source Revolution*. Sebastopol, CA: O'Reilly & Associates.

Tuomi, I. (2001). Internet, innovation, and Open Source: Actors in the network. *First Monday*, 6(1).

Ulrich, K. (1995). The role of product architecture in the manufacturing firm. *Research Policy*, 24:419–440.

van Wendel de Joode, R., de Bruijn, H., and van Eeten, M. (2003). Protecting the virtual commons. Self–organizing communities and innovative intellectual property regimes. NWO/ ITeR series.

von Hippel, E. (1990). Task partitioning: An innovation process variable. *Research Policy*, 19(5):407–418.

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

Zeitlyn, D. (2003). Gift economies in the development of open source software: anthropological reflections. *Research Policy*, 32(7):1287–1291.