Modularity in Action: GNU/Linux and Free/Open Source Software Development Model Unleashed

Alessandro Narduzzo[*] e Alessandro Rossi[†]

May 27, 2003

Quaderno DISA n. 78

[*]Dipartimento di Discipline Economiche–Aziendali, Alma Mater Studiorum – Università di Bologna e ROCK (Research on Organizations, Coordination and Knowledge), Università di Trento, email:narduz@economia.unibo.it.

[†]Dipartimento di Informatica e Studi Aziendali e ROCK (Research on Organizations, Coordination and Knowledge), Università di Trento, email:arossi@cs.unitn.it.

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 the GNU/Linux project as a whole, as well as to the outcomes of some of its sub–projects. 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.

1 Introduction

The GNU/Linux operating system growing popularity 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 relationships, 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.

From a variety of complementary perspectives, a growing number of studies are analyzing some key issues, in the attempt to understand the bases of such successful phenomenon. Our study is a rigorous and analytical attempt to interpret the design and the development of F/OSS in terms of a theory of modularity.

We argue that modularity, which can be regarded as an innovative manufacturing paradigm for the design and the production of complex artifacts, is a key element in explaining the development and the success of many F/OSS projects, and it offers a comprehensive explanation of many key issues in F/OSS development, such as how division of labor takes place within developers, how coordination is achieved and how code testing and integration is deployed, how innovation occurs, and so on.

Our reconsideration of the history of the GNU/Linux operating system highlights how the development process 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), while, at the same time, many critical pitfalls usually related to the architectural design of modules and interfaces were avoided. As we show in the paper, the design of the system was clearly inherited from a previously existing operating system platform, namely Unix.

Further, we will argue that, while traditionally information hiding has been viewed as the key principle guiding the design and the implementation of modular software artifacts, F/OSS development successful case studies seem to massively disregard this principle at the implementation level.

As a result, the GNU/Linux case study and the empirical observation on the rise and success of many F/OSS projects, suggest a possible step further in one of the most critical and controversial software engineering debate (Parnas's vs. Brook's view on information hiding of software modules). Furthermore, this empirical study suggests some neat observations pushing ahead our view of modularity, more and more often proposed as a new paradigm for design and production, even though it is still regarded by some authors as a black box. Moving from a stereotypical definition, centered on modules and interfaces, this paper will sketch some answers for several critical unsolved issues related to modularity, such as *(i)* the relationships between organizational structure and architecture design in modular projects, *(ii)* the role of the designer of a modular architecture as problem solver, *(iii)* how modularity copes with unforeseen interdependencies.

The paper is organized as follows: Section 2 introduces some of the most relevant topics of modularity in management and organization science. Then, it turns to software development, suggesting the idea of modularity as one of the fundamental issues in the evolution of the software engineering debate. In Section 3 some F/OSS case studies are described and interpreted through the lens of the theory of modular systems. Finally, Section 4 sketches some reflections on how F/OSS methodologies and practices may fully benefit by 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 represented by F/OSS may help in refining both existing theories of modularity, and their empirical application to domain different from software production.

2 Modularity

Modularity has being receiving increasing attention and popularity in a variety of fields, like neuroscience, artificial intelligence, architecture and urban design, management. 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.

The 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 [Schilling, 2000]. Literature on modularity emphasizes the importance of structures and relationships, and the outlined models all rely on an underlying system theory which provides a framework for understanding and describing in a suitable way 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.

In the paper we argue that, in order to outline a theory of modularity it is important to make a step forward and understand how modularization (modularity in action) takes place. From this point of view, the design of modular architectures gains special attention, as well as the process by which modular representations of the problem space are introduced and used in the design.

Herbert Simon's influence in the way modularity has been represented, is particularly evident [Devetag and Zaninotto, 2001]. 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, Simon's analysis of the Artificial as a complex system is extended to both activity (tasks) and objects (hierarchies), involves the decomposition of the system and have to deals with residual interdependencies [Simon, 1981]. As a matter of fact, speculations about modularity in management science are addressed to both production [Schilling, 2000, Badwin and Clark, 2000, Langlois, 2002] and product [Ulrich, 1995, Schilling, 2000].

Nonetheless, Simon's lesson goes further and has to be extended in order to understand the modularity black–box. Tempus and Hora parable is often introduced as a general problem analogous to the one modularity deal with: decomposition of product and task. Unfortunately, the decomposition itself is a complex problem to solve, since decompositions in sub-subproblems imply that the problem solvers have already structured a situation and they rely on that representation in order to define the problem space. Then, from the problem solver perspective the representations of the problem space of new complex artifacts, like for instance an operative system, do no exist per se, and need to be encountered in problem-solving fashion[1]. Designers are, first of all, cognitive problem solvers that deal with the representation of new, undetermined objects. They usually develop and introduce concepts, beliefs, practices and routines that are they use to progressively structure in a very specific manner what at the beginning was a blurry context. Likewise, modular architectures of complex artifacts are a consequence of particular kinds of problem space representations that do not exist by themselves, but that come out at the end of process of exploration (generation and evaluation) of possible solutions. Among all the possible representations, modular architectures are distinguished for some fundamental traits: the decomposition of the problem in sub-problems binds most of the interdependencies inside sub-problems (modules) that are usually specialized and that interact through standardized units (interfaces).

2.1 Modularity in management and organization science

In management and organization science literature, modularity has been introduced as a new paradigm for firm manufacturing [Ulrich, 1995, Schilling, 2000], organization design [Badwin 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 of products variety that is obtained by recombination (mix and match) of components. Modularity is viewed as a base for differentiation strategy: firms may enrich their product catalog and adapt to customers needs with limited additional costs. Moreover, modularity has also a great impact on production process as it positively affects flexibility, division of labor and specialization [Devetag and Zaninotto, 2001].

[1]Some relevant analysis along this direction has been done [Marengo et al., 2001, 2000, Gavetti and Levinthal, 2000], even though it still lacks of strong emprical evidence.

According to Badwin and Clark [2000] modularity in production systems is obtained following some general rules, originally drowned 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) architecture definition, b) interfaces specifications and c) modules integration tests. The amount of information about the inner description of each modules and their working is hidden from outside; it does not need to be defined ex–ante or shared during the process, since modules interactions follow exclusively the rules specified in 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 upon and the system does not work properly. The most common reason for this failure is the emergence of some interdependencies which were left out at the beginning, at the time of architecture and interfaces definition. The decomposition of a complex system is not at all a trivial activity, especially because most of the time it produces quasi–decomposable modules with some degree of interdependencies that may be out of control.

In our view, modularity design and development (modularization) are the key activities that have to be investigated in order to get into this black box, as Brusoni and Prencipe [2001] call it. Who is the designer? How come a designer develop from scratch a representation of the complex integrated system that has to be reduced in terms of quasi–decomposable modules (modularization)? Even in Badwin and Clark [2000] remarkable study on design and modularity, these issues are not fully considered and modularization is mainly regarded as abstract definition of some design rules about the architecture, the interfaces and the integration protocols. From our perspective, modularization needs to be analyzed as an activity and our attention is focused on three main research goals: first, understanding the initial problem which is a cognitive activity that comes out with the definition of modules boundaries. Second, finding empirical evidence that modular design may be more complex than interconnected design, due to the strong constraints posed by the ex–ante definition of the visible information. Third, considering under which conditions the amount of interdependencies that have been neglected at the design stage may be reduced further on during implementation of the architecture.

2.2 Modularity in software development

The notion of modularity is central in the design and production of software artifacts, especially for large and complex projects. Since from 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 textbook, clearly recognized that small sharp teams performed better than large ones, but they were not enough 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 overwhelming needs 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 divide labor vertically so to separate as much as possible high level activities (such as the design of a software artifact) from lower ones (such as 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, therefore complex, systems, where interactions among parts are nontrivial and generate high communication and coordination costs. Vertical division of labor is viewed as the way to avoid as much as possible these cost inefficiencies by concentrating design and architectural activities on few heads. What is clearly overlooked from Brook's perspective is that interdependencies may be not only considered as given constraints, but rather they may be strongly reduced at the architectural design level, by effective 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 one to each other [von Hippel, 1990, Langlois, 2002]. Hence, when subparts are almost independent it is possible to divide labor minimizing in miscoordination and communication costs.

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 prescribe to treat software modules as opaque entities, whose relevant information is only available to its inner programmer, while not being accessible to external programmers. Here module interfaces are the only information that is revealed, while all information regarding the design and how the module works is not communicated.

This approach to the design of software artifacts has been radically criticized since the beginning by Brooks and other scholars. As noted by Brooks [1995]:

“Harlan Mills have 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”.

Conversely, information hiding limits this process in that the innards of modules not their own are not available so that one has to infer what’s on the other side of the fence (the module interface) just looking at the interface details. As a matter of fact, if we take a look at current software practices nowadays, the widespread adoption of object oriented languages, the diffusion of component based development and many other popular trends in software engineering seems to have at large affirmed information hiding and modularity as the winners in the debate between interconnected versus modular software design. Even Brooks, 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”.

One of the key ideas of this paper is to show that the rise and affirmation of F/OSS development techniques, may be interpreted as the latest step in this 30–year old debate, since it shows that eventually Brooks was not completely wrong on information hiding. We will highlight in the following advantages and drawbacks of modular design and production of software artifacts. In the next Subsections, then, we will prove evidence of how F/OSS coding practices can be seen as unorthodox implementation of standard modular production techniques. Notably, we will argue that the power and the real novelty of F/OSS techniques lie in being modular at the design and early implementation stages of a project, while at the same time avoiding the typical pitfalls and limits of information hiding at the integration, debugging and testing phases of a project, where strict encapsulation and hiding is violated on a regular basis and where programming activities again are regarded, as in the Brooks' original vision, as a "public process".

In order to do that, we previously have to speculate a little bit more on the topic of 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]. Software seem to be a peculiar artifact if compared to physical artifacts when it comes to the topic of component interactions. As a matter of fact, components interface specification, defining interactions between components, seems, at a first glance, to result in system of less rigid constraints for the modules. In particular, physically specifications on how one component has to be connected to the other ones ("attachment interfaces", following the taxonomy developed by Sanchez [2000]), 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"[2]), clearly do not apply to the case of software

[2]More precisely, it should be noted that while this may be true for "hardware–based" environmental interactions, other kinds of environmental interactions, software–based, may emerge, such as the influence in common resources affecting the use of those common

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 theoretical software engineering literature and empirical case studies of software product development suggest that integrating software components may be harder 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 help us in refining our explanations of why integrating software modules and thus as a whole producing modularized software may be difficult [Brooks, 1986]. The author speculate 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 to enumerate (not to mention understand) all the possible state 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 preserve the initial conceptual integrity of the software project by 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 ele-

resources by other components.

ments ("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 compared to the production of manufacturing artifacts. This is largely due to the fact that even the act of decomposing a large software project into components is an activity that, given the multidimensional and invisible nature of interdependencies combined to bounded rationality of human designers of architectures, result, at its best, in a suboptimal outcome where while some source of interdependencies are well determined and taken into account in the design of components and in the design of interfaces, others are not. In some sense, even careful modularization of large software projects tends to be accomplished making trade–offs between source 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 modularization result in even greater problems at the final stages of code integration.

This is what originally struggled Brooks when he was criticizing Parnas' principles of information hiding. Brooks thought that programming should have been a public process where all information should have been completely available in order for failures in the design of software to become evident and to be corrected.

In the following we will review some well known case studies of F/OSS projects, trying to describe how F/OSS practices have uniquely adopted 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 try to better understand how F/OSS practices and pragmatics result in an improved implementation of the paradigm of modular software production, where the principle of information hiding is invoked at the design level while being later disregarded, at the implementation level, in the later stages of the process, where the free flow of information is effectively used in order to speed up the production process, taking advantage of what Brooks originally described as "programming as a public process".

3 Rise and success of a F/OSS project: the GNU/Linux case

GNU/Linux is part of a wider phenomenon that has its historical roots in an attempt to use intellectual property rights against a group of software developers. The leader of this group of hackers was Richard Stallman, the software that was covered by copyright was Unix, an operative system developed in a few years by a group of developers working for universities and private companies, the company that suddenly patented the system was AT&T, the time was 1979. As a reaction to the legal ties imposed on what had been developed as shared and open product, in 1984 Stallman started a new project called GNU (GNU is Not Unix) to develop free software. According to the GNU manifesto, people are free to read and use the software, free to change and improve it, free to make copies and distribute it. The only restriction is that new modified code inherits the same freedom of the original code. This form of hereditary freedom rights in using, changing and copying software was stated in special kind of copyright, the GNU Public Licence (GPL, also known as "copyleft"). This way Stallman succeeded in using copyright laws to preserve freedom on software use and manipulation. A growing community of software developers started to publish their application under GPL and in a few years a library of stable, efficient and well–done GNU applications were available. What GNU was still lacking of was the core part of an operative system, the kernel, as the GNU kernel project (HURD) was developing very slowly. Linus Torvalds' kernel filled this gap and GNU/Linux, a complete free operating system alternative to existing commercial and proprietary developed ones, started to be distributed in 1991.

3.1 Free/Open Source Software: origins, myth and stereotypes

Put it this way, the most prominent and visible example of F/OSS development seems to loose much of the radically revolutionary elements suggested by many popular accounts on the rise and success of the Linux operating system [Raymond, 1999]. Rather than being a completely innovative piece of code that was obtained using radically new techniques of software development, more mature interpretations of the phenomenon have already highlighted the existing similarities both at the architectural level with older operating systems (namely, the Unix operating system) and at the development level with formerly diffused coding practices among computer scientists and hackers (namely, to share the source code) [Stallman, 1999, Bezroukov, 1999a, Kuwabara, 2000].

As Bezroukov [1999b] neatly stated regarding open source:

"To me it is just another form of a scientific community. Similarly for me the development of Linux is not a new and revolutionary model, but just a logical continuation of the Free Software Foundation's (FSF) GNU project."

Accordingly, in the following we argue that, in order to understand the real revolutionary terms of GNU/Linux, one has in the first place to adopt an historical perspective on the evolution of GNU/Linux, through the lens of the theory of modular systems design and production.

3.2 Mimicking a previously existing design

At a closer inspection the Unix operating system can be viewed as a highly modular, scalable and portable platform. 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 "pipes" (making possible to connect the standard output of one program as the standard input of another one, allowing to communicate between different program and to execute complex tasks by means of composition of elementary actions), and sharing as a fundamental interface of communication text streams (also known as the "Unix philosophy" as formulated by McIllroy, 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 different hardware platform [Evers, 2000].[3]

The highly modular nature of its design structure had strong consequences both at the level of developers coding activities and at the level of users experience. Developers were able, thanks to the 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 not only allowed parallel development but also contribution of new components and modules allowing to substantially improving the overall design of the system via module innovation and competition between similar projects (both completely new modules and variation and improvements in existing ones). Since the design of Unix elicited experimentation and exploration of new solutions via module innovation, the community of hackers and computer scientists quickly developed systems to share working solutions to

[3]As originally noted by Badwin 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 treats computer resources (dynamic design modularity). Here we mainly concentrate on the first facet of modularity, while we will comment dynamic memory management in the final Section.

common problems both in the terms of improvements of existing modules and creation of new modules [Badwin 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 variety of different implementation 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 and pasdarans underline. They are rather mostly inherited by Linux direct ancestor, the Unix operating system. 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, represented at its beginning a titanic effort to offer a free alternative to currently existing commercial and proprietary operating system. In Stallman [1999] words:

“The principal goal of GNU was to be free software.”

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 reinforces one of the key argument about modularity, that was introduced in the former section: the act of properly designing a complex system characterized by a modular architecture is not a trivial task. On the contrary, modularity bears high costs: careful modularization is a cognitive challenging activity, since it translates in devising a decomposition of the whole system in autonomous subparts whose interdependencies are reduced to the minimum. Moreover, failure to perfectly modularize an architecture results in costs of dealing with fine tuning and tempering activities aimed at solving unexpected and unforeseen interdependencies. We will speculate further on this principle in Subsection 3.5, when discussing Torvalds' kernel architectural choice for GNU/Linux. Within the present Subsection, it is enough to note that the architectural choice followed since the beginning 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.

As was suggested in the Halloween–I document:

"The easiest way to get coordinated behavior from a large, semi–organized mob is to point them at a known target".

In this respect, the FSF community was 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 project upon a stable, mature and carefully modularized architecture was the key element to profit 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.[4]

[4]See also further on how in the case of the GNU/Linux kernel failure to correctly

3.3 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 Linux development.

One of the most criticized principle of the by other means 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 possible to observe 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 [1999b], Jones [2000]). Modularity allows us to refine and clarify those critics 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, since the key aspect in this respect is represented by the degree of task interdependency between the various members belonging to the project. In this view, the high productivity experimented in the GNU/Linux development is interpreted as largely due to the massively modularized structure of the project, allowing to have highly independent sub–projects joined by a limited number of developers, resembling in essence the theory of Brooks' surgical (small, skilled and focused) team, while the role of the Internet in

modularize the architecture resulted in serious troubles for the developers of the HURD micro–kernel.

this interpretation is of mere medium of exchange allowing distant communication.

Actually, our latest claim seems to run straightforward if we look at a general sub–project within the whole GNU/Linux architecture, but seems to be rather difficult to accept in some other cases, such as in the case of the development of the kernel, that have been undertaken thanks to the coordinated effort of hundreds of contributors. Consequently, we need to extend and clarify our latest claim, since it is not possible to address the problem of Brooks' Law and division of labor only at the horizontal level. We need to extend our analysis in order to consider also vertical division of labor.

3.4 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][5]

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 interpretation 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, simplifying a bit, 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 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 claim that the modular architecture of Linux is characterized by being quite hierarchical, rather than flat.

Reducing to the essence, it is indeed possible to distinguish 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 where 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

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

a small group of zealots. 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 hierachical 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 of 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 underlying bazaar activities, 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 GNU/Linux 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. To refine further our analysis we need to admit that, albeit largely based on the Unix architecture, there does exist something truly innovative and original in GNU/Linux. This pertains to the operating system kernel. In the following Subsection we reflect on its origins, highlighting the different approaches to modularity and interdependencies decomposition followed by two different projects: the Linux project and the HURD project.

3.5 The kernel issue: modularizing a monolith

Compared to integrated models, modular architectures are characterized by specialized, loosely–coupled components. Perfect decomposability of complex systems, like total integration, are idealtypical cases, while real–life systems are rather characterized by some degree of modularity. Most of the time, modular design and modular production leave some residual interdependencies among components, that are not managed by the interfaces. GNU/Linux is known for being a modular complex artifact and its successfull 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 that eventually acquired a modular structure. Like other analyses (see for instance [Moon and Sproull, 2000]) our examination is based on indirect sources of data that come from quite large mass of original documents (e.g. users groups archives) and first hand evidence (e.g. interviews and papers written by the key actors themselves.

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.[6]

At the time Linus Torvalds started to work on its kernel, a long debate was growing around the advantages offered by an alternative architecture, called microkernel, designed to work in all possible and different processors.[7] Compared to traditional, hardware dedicated kernels, microkernels appeared to be more complex and less efficient. They were more complex because even simple problems are treated as instances of general tasks that may involve a higher number of specifications and instructions to interact with other parts of the kernel; therefore, they may result to be less efficient as they do 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 much worse performance [Torvalds, 1999]. By the way, when Torvalds started to work on its kernel the Free Software Community and GNU partisans were already involved in the development of a microkernel (called HURD), but the task seemed to be much far away from its conclusion.

[6]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.

[7]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.

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 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 Alpha (another popular processor different from Intel 80386 and Motorola 68K), he realized that the original design was no longer effective and in 1993 he started to rewrite completely the kernel code. 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. Therefore, the general kernel model made use of modules and it was conceived having in mind those elements common to all typical architectures (even though it was not as rigorous and general as microkernels are). Following this scheme Torvalds could treat separately and confine in modules out of the core kernel all the hardware–specific pieces of code. These modules could be later updated or changed by Torvalds himself and by the other Linux developers with no effect on the kernel core.[8] Device drivers structure is a good example of the *third way* followed by Torvalds. One extreme solution is to put all the hardware 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 compromise the accomplishment of the project. To avoid such risks and keep as low as possible the

[8]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.

degree of complexity of the project, he decided to design a monolith and he actually wrote all the architectural specs himself,[9] avoiding all the hassles due to collective projects (e.g. division of labor, coordination, communication). On the other hands, Torvalds was aware of other advantages Linux could gain from introducing some degree of modularity, besides the porting issue: modular architectures would allow an incremental development of the system through the involvement of a mass of hackers working in parallel on different components.

"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]

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.

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

We already showed how critical were the consequences of inheriting a modular Unix–like architecture based on complementary and interconnected components. To a 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 emergence of unforeseen interdependencies within the modules that may rise with the future development of hardware and software. While HURD rose as an attempt to develop a fully general and modular system, Linux's 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 successful endurance 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 faced by rewriting or adding some modules, as it happened for instance with a memory manager module; sometimes the adopted solutions are not adequate and the communities of developers that not agree with the final decision may introducing new alternative versions of the system. These forks are expression of a coordination failure when a community does not converge on an 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 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:

- The design of modular architectures from scratch may reveal to be extremely complex tasks and therefore designers may prefer integrated solutions that are easier to manage.

- To avoid out of control increasing of interdependencies the first phase of new projects may be more efficiently accomplished by individuals, rather than groups of developers; on the other hand, most of the most successful F/OSS stories have been started to solve specific problems and at the beginning are one-man projects: Sendmail was initially developed by Eric Allman to route email for other users within UC Berkely, Perl by Larry Wall to solve some bothersome problems in system administration, World Wide Web by Tim Berners-Lee as group environment for academic information sharing among high-energy physicists, etc. [O'Reilly, 1999].

- The evolution of Linux kernel towards modular design offers some insights to a general theory of modularity: it is possible and effective to combine together under the same architecture modular components and integrated parts. Later on the designers may introduce a higher degree of modularity by adapting the original architecture. In other words, modularity arises more as a process of evolutionary design (modularization), rather than as ultimate property of an artifact.

- Our intuition is that the general violation of the information hiding principle allowed by the openness of the source code was especially important in this case. In general terms a partially-modular architecture, as the one designed for Linux kernel by Torvalds, seems to be more vulnerable by the emergence of unforeseen interdependencies that come out on later development, compared to a fully-modular architecture (like HURD). Nevertheless Torvalds architecture could bear these situations more easily because of the violation of one the fundamental rules of modular implementations, that is the information hiding. Since the source code is open, developers may better handle with the weaknesses of the architecture, avoiding or preventing major inconsistencies.

3.6 Beyond the principles of modularity

We have so far advocate the perspective that the theory of modular systems may help in interpreting the rise and success of GNU/Linux and of F/OSS methodologies in general. Nevertheless, in the previous sections, we have also argued that modular design and production of software artifacts is not new to software engineering and that it would be misleading and unrepresentative of reality to attribute its original genesis or to restrict its current application to the world of F/OSS development.

As a matter of fact, successful examples of production of modularized software may be found also in proprietary developed code: take 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 9 month 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.”[10]

Hence, the occasional reader may be concerned over our emphasis on the role of modularity in explaining the emergence of F/OSS development. In particular, he might as well treat modularity as one of the common trends present both in F/OSS and proprietary software development, and look elsewhere for differences and peculiarities of F/OSS practices. This is for instance what happened to some of the assumed peculiarities of Linus's development style highlighted by Raymond [1999]: subsequent contributions have

[10]Cited in MacCormack [2001]).

suggested close resemblance of some of these elements to practices of rapid development common in closed and proprietary development.[11]

Still, we are convinced that modularity is helpful in clarifying the debate and has a great explaining power in characterizing F/OSS. Hence, to address this potential critiques one has to take a step forward in the theory of modularity and has to reflect on how GNU/Linux, and more generally F/OSS development, originally adapted the principles of modularity.

We have already speculated on advantages and risks of modularity: a well–decomposed architecture seems to reconcile considerations about division of labor and size of a project with concerns of high speed of development. Nevertheless, for large complex software artifacts it may be almost impossible to separate *ex-ante* all interdependencies, so unforeseen coupling between components at later stages (like for instance, integrating new and existing modules), may strongly affect the final outcome of the process.

The fundamental innovation in F/OSS practices lies in how the basic postulate of information hiding is adapted to face these difficulties. While information hiding is clearly at the core of designers activities when initially decomposing a software project in modules, the same principle is then 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 makes programming a true public process, since good coding solution are shared and adapted to solve new similar problems, and *ex-post* interdependency conflicts are handled by employing a wider set of fine tuning strategies. The absence of code ownership, typical of proprietary closed development, allows to cope with unforeseen interdependencies in many alternative ways: bugs are highlighted and corrected by

[11]See for instance Bezroukov [1999b] addressing a critical revision of some Raymond's postulates or Cusumano and Selby [1995] on rapid development methodologies in a large software corporation).

other eyeballs, design incoherencies are anticipated and discussed by peers reviewing code elsewhere written, and so on.

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

4 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 [Badwin 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). Modularity copes with complex systems, as it comprises the interdependencies at the level of modules. Modularization is a design process aimed to specify the architecture, the interfaces and the protocols of integration and testing. The architecture design encompasses to state which are the modules and what they do and it shapes the border between visible and hidden information; the interfaces set the rules of interactions among the modules. Finally, integration and testing steps assess whether unexpected inconsistencies come out when modules are combined together, revealing some weaknesses in the architecture or in the interface design. After some refinements the design rules set is complete and the final architecture is supposed to take into account all the (relevant) interdependencies.

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

Unfortunately, this perspective underestimates that modularization of complex systems generally resolves on a quasi–decomposition 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.

GNU/Linux is a story of modularity and modularization, but things are less neat and smooth than it is spelled out by the theory. Mimicking UNIX, GNU/Linux inherits its modular structure; analyzing UNIX architecture, Baldwin and Clark (2000, p. 334–336) point out that its modular design has a static and a dynamic dimension. This operative system has modular static design based on programs, file structure and pipes. Programs are the actual modules that are activated to perform the tasks, file structure defines how the modules are organized and pipes allow to concatenate different modules in order to accomplish compound tasks. Moreover, UNIX is modular in a dynamic sense, as the activity of the modules is regulated according to an hierarchical organization (runtime configuration) that allows multiple users working on multiple tasks to minimize the interdependencies in time sharing of common resources (processors, printers and other devices).

GNU/Linux, on the other hand, is a story of *unorthodox modularity*: information hiding principle is significantly ignored and the artifact evolves mainly through a repertoire of practices (e.g. peer coding and debugging, frank discussions, open decisions) where developers and users work aside, tinkering and patching the original modular product and, overall, violating another of the law of the olympic modularity stating that the only operators are manipulation at the module level (splitting, substituting, augmenting, excluding, inverting, porting) (Baldwin and Clark (2000, p. 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 standards (e.g. POSIX) 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 architectures for complex products expose the designers to the risk of unforeseen interdependencies that may ultimately endanger the whole project. Besides, as F/OSS developers are distributed organizations and the community members communicate only remotely, coordination and collective decision making seem to be two fundamental issues in F/OSS development. Following the same rationale, F/OSS communities should be aware that forking may become a serious potential thread, as the in the long run a growing number of alternative, incompatible modules ultimately increases the degree of interdependence of the system. In fact, new modules (programs, upgrades, patches) either deepen the forking effect or try to handle it and in this case they need to take into account all the possible different user configurations.

Our analysis seems to clearly highlight the strengths of employing a modular approach to software production, suggesting that the success of F/OSS methodologies may rely on the power of modularity in leveraging massively parallel and distributed development. Moreover, F/OSS practices seem to overcome one of the traditional limit of modularity. Disregarding the information hiding principle at the implementation level allows to fix more effectively flawed modularization managing unforeseen interdependencies between modules. Nevertheless, the principles of modularity have still a positive impact in architecture design, since careful ex-ante modularization of complex software architectures immensely speed up the development phases. The modularization phase (the decomposition in modules) has been until now in most of the successful F/OSS projects handled out largely by inheriting and adapting existing architectures. As long as nowadays the F/OSS community seems to be more and more committed in developing state of the art software based on innovative complex architectures, answering to questions about who should be the designer and how should innovative architectures look like, seems to us one of the most challenging issue that the F/OSS movement will be soon facing.

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 out as the operative system evolve, revealing some inconsistencies are faced by violating the information hiding principle. In questioning how exendible this experience is 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 F/OSS world imitation and copy are encouraged and protected by a reverse form of copyright (copyleft). On the contrary, in many other contexts of production, organizations and individual are characterized by actively preventing this sharing, at least at the inter–organizational level.

Second, GNU/Linux case is characterized by a deep overlap between producers and users. At least at the beginning of the story most users were developers or had some skills that allowed them to perform successful tinkering. Again, most of the traditional ways to conceive innovation and product development in other domains keeps separated producers and users, even though today customers are more and more often directly involved in the definition of their own product.

Acknowledgments

The authors would like to thank the ROCK (Research on Organizations, Coordination and Knowledge) Group members at the University of Trento, for their helpful comments and 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. 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 Giovani Ricercatori 2001 is gratefully acknowledged.

References

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

C. Y. Badwin and K. B. Clark. *Design Rules. Vol. I: The Power of Modularity.* The MIT Press, 2000.

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

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

F. P. Brooks. *The Mythical Man–Month. Essays on Software Engineering.* Addison Wesley, 1975.

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

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

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

M.A. Cusumano and R.W. Selby. *Microsoft Secrets How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, 1995.

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

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

S. Evers. An introduction to Open Source software development. Diploma thesis, 2000.

G. Gavetti and D. Levinthal. Looking forward and looking backward: Cognitive and experiential search. *Administrative Science Quarterly*, 45:113–137, 2000.

P. Jones. Brooks' law and Open Source: The more the merrier? *IBM*, 4(10), 2000. Accessed Jan 2, 2003, from http://www-106.ibm.com/developerworks/library/merrier.html.

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

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

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

L. Marengo, G. Dosi, C. Pasquali, and M. Valente. The structure of problem–solving knowledge and the structure of organizations. *Industrial and Corporate Change*, 9(4):757–788, 2000.

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

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

T. O'Reilly. Lessons from Open–Source software development. *Communication of the ACM*, 42(4):33–45, 1999.

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

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

D. K. Rosenberg. *Open Source. The Unauthorized White Papers.* IDG Book Worldwide, 2000.

H. P. Salus. *A Quarter Century of UNIX.* Addison–Welsey, 1994.

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

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

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

H. A. Simon. *The Sciences of the Artificial, 2nd ed.* The MIT Press, 1981.

R. Stallman. The GNU operating system and the free software movement. In C. DiBona, S. Ockman, and M. Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates, 1999.

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

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

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