Preliminary Draft
Comments Welcome

Does Code Architecture Mitigate Free Riding in the Open Source Development Model?

Carliss Y. Baldwin
Kim B. Clark

Harvard Business School

This Draft:
June 1, 2003

Our thanks to Jason Woodard, Siobhan O'Mahony, Sonali Shah, John Rusnak, Alan MacCormack, Eric von Hippel, Daniel Frye, Karim Lakhani and members of the Negotiations, Organizations and Markets group at Harvard Business School for generously sharing their time and their insight. We alone are responsible for errors, oversights and faulty reasoning.

Direct correspondence to:
Carliss Y. Baldwin
Harvard Business School
cbaldwin@hbs.edu

Does Code Architecture Mitigate Free Riding in the Open Source Development Model?

Carliss Y. Baldwin and Kim B. Clark

Abstract

This paper argues that the *architecture* of a codebase is a critical factor that lies at the heart of the open source development process. To support this argument, we define two observable properties of an architecture: (1) its modularity and (2) its option values. Developers can make informed judgments about modularity and option value from early code releases. Their judgments in turn will influence their decisions to work and to contribute their code back to the community. We go on to suggest that the core of the open source development process can be thought of as *two linked games* played within a codebase architecture. The first game involves the implicit exchange of effort directed at the modules and option values of a codebase; the second is a Prisoners' Dilemma game triggered by the irreducible costs of communicating. The implicit exchange of effort among developers is made possible by the the non-rivalrous nature of the codebase and by the modularity and option values of the codebase's architecture. This exchange creates value for all participants, both workers and free-riders. In contrast, the Prisoners' Dilemma is a problem that must be surmounted if the exchanges are to take place. It can be addressed through a combination of reducing the costs of communication, providing rewards, and encouraging repeated interactions. Finally, the initial design and "opening up" of a codebase can be seen as a rational move by an architect who is seeking to test the environment in hopes of initiating exchanges of effort with other developers.

Key words: architecture — modularity — option value — public goods — non-rival goods — free-riding — open source — software development — prisoners' dilemma game — institutional economics — organizational economics

JEL Classification: D23, L22, L23, M11, O31, O34, P13

1 Introduction

In the history of software unpaid volunteers have played a remarkable role. Their contributions challenge conventional wisdom, which says that those who play central roles in the creation of commercially valuable artifacts will expect and receive compensation. In times past, unpaid volunteers were critical to the development of UNIX and the Internet. More recently, under the rubric of "open source," large numbers of software engineers are working to design, publish, test, and de-bug non-commercial code. In its earliest forms, this kind of software was mainly of interest to academics and to technologists. Today, however, open source software, its development process, and the software architects and engineers who do the work, have become central factors in the evolution of the computer industry.

The viability of the open source mode of software development is not in question. It exists and works, and has done so for many years.[1] But there are still important questions about this process which, if answered, would illuminate the forces driving the industry today. There is in the first instance the question: how does this work? Is open source development like an exclusive social club, or does it also have a strong economic logic underlying its functioning and coherence? And what are the boundaries of its application? These questions are of interest because we would like to know something about the limits of the phenomenon. A second set of questions concerns the competitive impact of the open source movement. The emergence of open source software like Linux and Apache has clearly affected the competitive strategies of many companies. At the same time, the embrace of open source software by commercial enterprises appears to be critical to the software's success. But will that very embrace undercut the development process?

Finally, proponents of open source software claim that code developed in this way is different in structure and in some ways superior to code developed using proprietary development processes.[2] If

---

[1] On the origins of user groups that produced and shared software, see Bashe *et. al.* (1986) pp. 347-371. On the origins of the "hacker culture," the Unix culture; and the open source software movement, see, among others, Levy (1984); Salus (1994); Raymond (1997) pp. 7-26.

[2] The claim that open-source code is both different and better than closed-source code has been made most forcefully by Eric Raymond. He in turn attributes this view to the so-called "pragmatists" among open-source developers "who [are] loyal … to engineering traditions … [which] included the intertwined technical cultures of Unix and the pre-commercial Internet. … The typical pragmatist's … major grievance against the corporate world … is that worlds's perverse refusal to adopt superior approaches incorporating Unix and open standards and open-source

open source code is indeed superior—or even simply different in structure—from commercial code, the open source development process might have quite powerful and profound effects on the engineering design of future codebases. However, in order to make sense of this claim, the interaction between the open source development process and the design and structure of codebases needs to be better understood. Does the development process affect the evolving structure of a codebase, and if so, how? Then again, does the structure of a codebase enable or constrain the development process? These are the specific questions we aim to address in our work. Broadly speaking, we want to understand the effects of codebase structure on developers' and users' incentives to invest effort in the development process and conversely, the effects of the developers' and users' investments on the evolution of the codebase.

Building a comprehensive model of the way developers and users interact with a complex and evolving codebase is a large undertaking, which goes well beyond the scope of a single paper. Therefore as a first step in this paper, we will focus on the early stages of the development process: the design of the initial codebase, and the developers' decisions to work on it. We will argue that the *architecture* of a codebase is a critical factor that lies at the heart of the open source development process. To support this argument, we will define two observable properties of an architecture: (1) its modularity and (2) its option values. We contend that developers with appropriate skills and training can make informed judgments about modularity and option value from early, partially implemented code releases. Their judgments in turn will influence their decisions to work on the codebase and to contribute their code back to the community. Finally, their contributed work will cause the codebase to evolve along the paths specified by the original design.

The rest of the paper is organized as follows. In section 2, as indicated, we discuss code architecture and define modularity and option value. In section 3, we model critical aspects of the open source development process as a simple game. In sections 4 and 5, we investigate how changes in code architecture affect developers' incentives to work in the context of this game. We will show that more

---

software"(Raymond, 1997, p. 84). Indeed, the belief that open-source code is likely to be better than closed-source code is common among software developers. (For example, see numerous quotes in O'Mahony, 2002) However, the opposite view—that proprietary code is likely to be superior— is common, too. The fact that software developers see differences in the structure of open and closed source software, but disagree on the effects of those differences makes an investigation of the interaction of architecture, codebase, and process especially interesting.

modules and more option value (1) increase developers' incentives to work on the codebase; and (2) decrease the amount of free-riding in equilibrium. These effects occur because modularity and option value create opportunities for the exchange of valuable work among developers. These opportunities for fruitful exchange do not exist to the same degree in "monolithic" codebases, which lack modules and option values.[3]

Section 6 considers developers' incentives to voluntarily reveal code they have already created. In section 7, we place our results in the context of other formal and informal models of the open source development process. Section 8 concludes by assessing the limits of the open source development process and, within those boundaries, the implications for competing firms.

2 Codebase Architecture: What It Is and Why It Matters

The essence of our argument is that the architecture of a codebase affects the open source development process and vice versa. This point is important because architecture is not a matter of natural law, but is to a large degree under the control of an initial designer, the so-called "architect" of the code. As the code is structured by the architect, so will the work of the developers be organized, and so will the implementation of the codebase unfold. In this paper, we will show that a code architecture can be well-suited or ill-suited to the open source development process. Here we begin by introducing two important properties of a codebase architecture: (1) its *modular structure*; and (2) the *option values embedded in the modules*. These properties come into being very early in the development process. By looking at early releases of a codebase, developers other than the initial architect can often tell how modular and how "option-laden" the future, evolving codebase will be.[4] Those judgments in turn affect the developers' incentives to create new code and to contribute it back to a collective development effort.

[3] The term "monolithic" is commonly used by software developers to describe highly interdependent blocks of code. Another, less flattering term is "spaghetti code."

[4] In another paper, we plan to analyze the design and structure of the Linux kernel as documented in its early releases. Our purpose in that undertaking is identify the modules and option values in the codebase as it appeared to the early developers.

2.1 Modularity

A complex system is said to exhibit *modularity* if its parts operate independently, but still support the functioning of the whole. Modularity is not an absolute quality, however. Systems can have different modular structures and different degrees of interdependence between their respective elements.[5]

The different parts of a modular system must be compatible. Compatibility is ensured by *design rules* that govern the *architecture*, the *interfaces*, and the *tests* of the system. Thus "modularizing" a system involves specifying its architecture, that is, what its modules are and what each will do; specifying its interfaces, ie., how the modules will interact; and specifying a set of tests that establish that the modules are compatible and how well each module performs its job.

For our purpose it is useful to divide a codebase and its architecture into (1) a minimal system; and (2) a set of modules. The *minimal system* is the smallest system that can perform the job the system is meant to do. As a matter of definition, systems that lack any component of the minimal system are incomplete and worth nothing to users. *Modules* are distinct parts of the larger system, which can be designed and produced independently of one another but will function together as a whole. The tasks of coding different modules can be divided among people who are not in close day-to-day contact with one another. In addition, one version of a module can be swapped for another without harm to the rest of the system. As a result, the incremental value of a new module (or a new version of an existing module) can be added to the value of the existing system to obtain the new value of the changed system.

We emphasize that, for our purposes, "the minimal system" of a codebase and its "modules" are simply definitions that allow us to characterize the different parts of the codebase and its architecture. We are not excluding any possible codebase or architecture from consideration, we are simply saying that an architecture and its related codebase can be analytically divided into a minimal system and some number of modules.

---

[5] This definition is taken from Rumelhart and McClelland (1995), as quoted in Baldwin and Clark (2000). Although Herbert Simon did not use the term "modularity," the attribute we now call by that name is very similar to the property he famously called "near-decomposability." The essential similarity of these two ideas can be seen by referring to matrix maps of interdependencies for modular and near-decomposable systems. For near-decomposable systems, such maps may be found in Simon (1962) and Simon and Augier (2002.); for modular systems, see Baldwin and Clark (2000), Chapter 3.

2.2 Option Value

Software development is a design process. A fundamental property of designs is that at the start of any design process, the final outcome is uncertain. Once the full design has been specified and is certain, *then the development process for that design is over*.

Uncertainty about the final design almost always translates into uncertainty about the design's eventual value. How well will the end-product of the design process perform its intended functions? And what will it be worth to users? These questions can never be answered with certainty at the beginning of any substantive development process. Thus the ultimate value of a design is unknown when the development process begins and continues to be uncertain while the process is ongoing.

Uncertainty about final value causes new designs to have "option-like" properties. According to modern finance theory, an option is "the right but not the obligation" to choose a course of action and obtain an associated payoff. In the case of designs, the "optional" course of action is the opportunity to implement a new design. A new design creates the ability but not the necessity—the right but not the obligation—to do something in a different way. *But the new way does not have to be adopted*, and indeed, as long as the designers are rational, it will be adopted only if it is better that its alternatives. This in turn means that the economic value of a new design is properly modeled as an option.

The option-like structure of designs has three important, unobvious consequences. In the first place, when payoffs take the form of options, taking more risk creates more value.[6] Risk here is defined as the *ex ante* dispersion of potential outcomes. Intuitively, a risky design is one with high technical potential but no guarantee of success. "Taking more risk" means accepting the prospect of a greater *ex ante* dispersion. Thus a risky design process is one that has a very high potential value conditional on success but, symmetrically, a very low, perhaps negative, value conditional on failure.

What makes the design an option, however, is that the low-valued outcomes do not have to be passively accepted. As we said, the new design does not have to be adopted; rationally, it will be adopted only if it is better than the alternatives, including the *status quo* alternative. In effect, then, the downside potential of a risky design is limited by the option to reject it after the fact. This means that "risk" creates

[6] This is a basic property of options, proved for any distribution of payoffs by Robert Merton in 1973.

only upside potential. More risk, in turn, means more upside potential, hence more value.[7] (Because in a world of options, risk and upside potential are equivalent, we will use the term "technical potential" to characterize the dispersion parameters of technical gambles.)

When payoffs take the form of options, it is also true that seemingly redundant efforts may be value-increasing.[8] Two attempts to create a new design may arrive at different endpoints, in which case, the designers have the option to take the better of the two. Thus faced with a risky design, which has a wide range of potential outcomes, it may be desirable to run multiple "design experiments" with the same functional objective. These experiments may take place in parallel or in sequence, or in a combination of both modes.[9] But whatever the mode of experimentation, more risky designs call for more experiments, other things equal.

Finally options interact with modularity in a powerful way. By definition, a modular architecture allows module designs to be changed and improved over time without undercutting the functionality of the system as a whole. In this sense a modular architecture is "tolerant of uncertainty" and "welcomes experiments" in the design of modules. As a result, modules and "option-like" design experiments are economic complements: an increase in one makes the other more valuable.[10]

We will revisit modularity and option values in sections 4 and 5 when we analyze the effects of codebase architecture on open source developers' incentives to write code. But first we must define the developers' relationship to one another in the context of the open source development process. We turn to that task in the next section.

---

[7] It follows, of course, that if a risky design is "hardwired" into a system so that it must be implemented regardless of its value, then the design process loses its option-like properties. In such cases, "taking more risk" in the sense defined above, will not increase, and may decrease value.

[8] Stulz (1982) first analyzed the option to take the higher-valued of two risky assets. Sanchez (1991) worked out the real option value of parallel design effort in product development.

[9] Loch, Terwiesch and Thomke (2001).

[10] This is the definition of economic complementarity used by Milgrom and Roberts (1990) and Topkis (1998, p. 43). The complementarity of modularity and experimentation was first demonstrated by Baldwin and Clark (1992; 2000, Chapter 10). See Section 5 below.

3 Games of "Involuntary Altruism"

A fundamental feature of software code is that it is a "non-rival" good. This means that the use of the code by one developer does not prevent its use by another developer, or many others. In this section, we model a work environment in which non-rival goods are produced in terms of a simple game. The game we will specify is also characterized by "involuntary altruism" in the supply of effort. By "involuntary altruism" we mean that each player's effort contributes positively to the welfare of the other player(s), and the benefit occurs whether the first player wants to help the other(s) or not.[11]

The idea that code is a non-rival good is built into the basic structure of this game through the assumption that each developer's effort automatically benefits the others. The result is a well-known game form, which in economics is sometimes labelled "the private provision of public goods." This basic game form has been applied to the open source development process by Johnson (2002), who used it to explore the effects of the size of the developer base on welfare and the distribution of effort and costs. Johnson also compared open source and for-profit development models, and looked at codebase "modularity" in a preliminary way.

There are other important features of the open source development process, which we have also built into our formal models. First of all, the developers must be willing participants in the process. To this end, we assume that developers may freely choose to join the game or remain isolated in a "Robinson Crusoe" environment. We will also assume that a developer who joins the game may choose to work or free-ride. Finally, although we will initially assume that a developer's work product is automatically revealed to the rest of the community (the other developers who have joined the game), later, we will assume that developers may choose whether to reveal their code, and if they do so choose, must pay a cost of communication.

Before we go further, we should say that there are many ways to model the specific "production function" in a game of involuntary altruism. Assumptions about the production function, in turn, can

---

[11] In the language of public economics, work products that benefit others whether the worker wills it or not are called "non-exclusive" or "non-excludable" goods. A good that is both "non-rival" and "non-exclusive" in this sense is called a "public good" or sometimes a "collective good." There is a long and interesting history of the evolution of these terms, beginning with Samuelson (1954), and continuing through Head (1962); Olsen (1971); and Chamberlin (1974). Lessig (2001, p. 99-97) applies these concepts to ideas, designs, software and the Internet, and connects them to the property rights theory of Thomas Jefferson.

affect the game's outcomes in important ways, indeed that is exactly what we want to explore. However, in the interest of clarity, we will begin by laying out a very simple, symmetric, two-person, one-shot scenario with no uncertainty as to payoffs. We will then discuss variations of the basic scenario, including multi-person games, games with imperfect information, games with payoff uncertainty, and the aforementioned game of "voluntary revelation" in which developers must decide to publish their code and pay a price for doing so. We now turn our attention to understanding the formal properties of the basic game.

3.1 The Basic Game of Involuntary Altruism

Consider two developers, each of whom needs a specific block of code that does not yet exist. The code's value to either developer is v and the cost of writing it is c. The value to each is greater than the cost: $v > c$, thus either party has an incentive to write code if the other doesn't.[12] Moreover, in a "Robinson Crusoe" environment, in which the developers are isolated and do not communicate, both developers will write code.

For purposes of the game, we assume that the developers can communicate, and that each developer can use the other's code with no loss of value. What's more, as noted above, we assume that *neither can prevent the other from using the code that he or she writes*: if the code is created, it will automatically be revealed to the other party. This last assumption, we emphasize, is not true of the open source development process: open source developers are not compelled to reveal the code that they write for a given codebase. However, for analytic purposes, it is useful to consider this unrealistic environment first, and then introduce a separate "decision to reveal."

We begin by assuming that the codebase architecture is not modular, and the work of writing the code is not divisible.[13] The normal form of this game is shown in figure 1. It is well known and easily verified that this game has two Nash equilibria in pure strategies: these are the shaded, off-diagonal cells in which one developer works and the other doesn't. These equilibria are *efficient*, in the sense that

[12] These values and costs can be interpreted as dollar equivalents of private benefits and costs.

[13] Modularity implies divisibility of effort, however, a codebase can be divisible in terms of effort, but still not modular. See the discussion at the beginning of section 4 below.

minimum effort is expended for maximum total return. However, the equilibria are also *inequitable*, because the non-worker "free-rides" on the effort of the worker.

Figure 1
Normal Form of a Simple Game of Involuntary Altruism

	Developer 2: Don't Work	Developer 2: Work
Developer 1: Don't Work	0 , 0	v, v-c
Work	v-c, v	v-c, v-c

There is also a mixed-strategy equilibrium of this game wherein each developer works with probability π^* or doesn't work with probability $1-\pi^*$.[14] We find π^* by setting the expected payoff of working and not working equal to one another and solving the resulting expression:

$$\pi^* = (v - c)/v \quad .$$

In this particular case, the mixed-strategy Nash equilibrium of the normal-form game is also an evolutionarily stable strategy (ESS) in the corresponding evolutionary game.[15]

---

[14] Throughout this paper, we will use a star " * " or a dagger " † " to indicate specific values of a variable. In this case, π indicates a general probability *variable*, while π^* is the particular probability *value* that characterizes the mixed strategy equilibrium.

[15] Evolutionary games do not require the players to understand in detail the game they are playing. They lend themselves to models of trial-and-error, rules of thumb, and norms of behavior. Formally, an "evolutionary game" is a symmetric game in which players are (1) randomly drawn from a population of "strategy-types," and (2) proceed to play a "stage-game" like the one defined above. An *evolutionarily stable strategy* (ESS) is a distribution of strategy-types that cannot be "invaded." (Maynard Smith, 1982.) To bring this concept down to earth, consider the game of involuntary altruism specified above. Imagine that the two developers are drawn randomly from a population consisting of π Workers and $1-\pi$ Free-riders. If $\pi = 1$, that is, the population is all Workers, then a "mutant" Free-rider will do very well by not working. Thus a population of all Workers is not evolutionarily stable. Conversely, if $\pi = 0$, implying that the population is all Free-riders, a mutant Worker will do better than the *average* member of the non-working population. Thus a population of Free-riders is not evolutionarily stable either. Intuitively, in a population if everyone works, some will find it advantageous to free-ride, but if everyone loafs, some will find it advantageous to work. However, if the fraction of Workers and Free-riders equals the mixed-strategy equilibrium fraction π^* derived above, then a new "mutant" Worker or Free-rider will not have a higher expected payoff than the average member of the population.

As the number of persons playing the game increases, the equilibria change in predictable ways. The pure-strategy equilibria always have the property that one developer works and the others all free-ride. In addition, for a given number of developers, there is always one mixed-strategy equilibrium. The equilibrium "mixing probability," θ^*, decreases as the number of developers increases, but (1) the probability that a given developer encounters at least one worker is always $(v-c)/v$; and (2) the expected payoff per developer is always $v-c$.

3.2 The Attractiveness of the Game to Outside Developers

For developers, the game we have specified is an alternative to coding in isolation. How attractive is it relative to that alternative? Table 1 addresses this question. It compares the per-person payoffs in (1) the Robinson Crusoe environment (what the developers would do in isolation); (2) the pure-strategy equilibria of the involuntary altruism game; and (3) the mixed-strategy, evolutionarily stable equilibrium of the game. (In the latter case, the results are derived by substituting for θ^* and rearranging terms.)

Table 1
Comparison of Equilibrium Payoffs in the Involuntary Altruism Game

	Per Person Expected Payoff
Robinson Crusoe Environment	$v-c$
Pure-Strategy Equilibria (one works, the others do not)	
Free-rider	v
Worker	$v-c$
Mixed-Strategy, Evolutionarily Stable Equilibrium	$v-c$

These results point to a problematic aspect of the involuntary altruism game. In the pure-strategy equilibria, the free-riders do well, but the lone worker does no better than if she were to remain isolated. Her effort is a pure gift to her peers. Similarly, in the mixed-strategy, evolutionarily stable equilibrium,

the expected payoff per person is the same as in the Robinson Crusoe environment. In expectation, a developer does not lose by joining the game, but neither does he benefit from being part of a collective effort. Put another way, a population of isolated developers will do just as well as a population of *cooperating groups of developers* that contains evolutionarily stable proportions of workers and free-riders.

In the next section, we shall see that modularizing a codebase changes equilibrium behavior and can increase the expected value of a collective development effort. As a result, a game of involuntary altruism played within a modular architecture often dominates the Robinson Crusoe alternative, and isolated developers will then have incentives to join the collective process. Why is this? Intuitively, the developers of one module can benefit from code written by others for other modules. Implicitly a modular architecture creates opportunities for developers to exchange their work on different parts of the system. However, the effects of modularity on free-riding are not transparent, and the attractiveness of collective development depends crucially on the endogenous degree of free-riding. Beyond that, the effects of option value on free-riding in equilibrium are not clear at all. Finally, given imperfect information about what others are doing, we shall see that the collective development process in a modular system will generally not be as efficient as a process in which one developer works and the others do not. In this respect, the glass is both half-empty and half-full.

4 Bringing Architecture into the Game

In the game of involuntary altruism described above, the assumptions of the game form did not allow the work of coding to be split between the two developers in any way. There was no division of labor, hence, by our definition, the implicit architecture of the codebase was *not modular*. In fact, in the context of coding, a division of labor (work-splitting) can be modeled in two ways, corresponding to the following two cases:

> **Case 1.** Work on what we have called a *minimal system* may be split among several developers. By definition, all parts of the minimal system will be needed to achieve any value. In this case, work is divisible, but value is not.
>
> **Case 2.** An already-functional minimal system may have *modules*. In this case, work on one module can proceed independently of work on other modules. Because the minimal system already exists, failure to supply a module does not eradicate the value of effort directed at other modules. Hence not only is the work able to be divided, but *the value of each developer's effort is also distributed over identifiable pieces*.

Whether work-splitting is possible and how value is distributed over the targets of effort depends ultimately on the architecture of the codebase. Modular systems are designed to make possible *both* work-splitting and independent, incremental additions to value. Thus, Case 2 is descriptive of modular codebases, whereas Case 1 is not. These may seem like subtle and unimportant differences: technological hairsplitting in other words. We shall see, however, that they can have important effects on the equilibria of the involuntary altruism game.[16]

As should be evident from these definitions, codebases must undergo an evolution through time. While the minimal system is being coded, the codebase is technically not modular, although its architecture (observable to developers) may portend future modularity. We emphasize (again) that for a given codebase, the size of the minimal system, and the size and number of the modules, hence the time needed to code each part, and the value of the different pieces, are fundamental architectural decisions. They are, moreover, decisions that occur very early in the development process. Thus even when codebases are preliminary and incomplete, they can and do differ in observable ways on these dimensions.

4.1 Modularity in the Game of Involuntary Altruism with Perfect Information

To fix ideas, let us assume that the codebase described in Section 3 has been modularized with no change in its aggregate value. The modularized codebase has distinct parts: in particular, the architecture specifies a minimal system plus j modules. At the time of the game, we assume that the minimal system has been coded. Without loss of generality, we assign that part of the system a value and cost of zero.[17]

Under the architectural design rules, each module can be worked on separately, and they are incrementally value-additive. If developed, we assume each will add value v/j to the system at a cost of c/j. Thus a "total system" comprising the minimal system plus j modules has the same value and overall cost as the non-modular system described in the previous section. (These allocations of value and cost

[16] Formal models of collective action problems and the supply of public goods often assume (explicitly or implicitly) that effort is divisible, but value cannot be assigned to different pieces of the good in question. (A generic model of this type is Gintis's "cooperative fishing" game: See Gintis, 2000, pp. 152-153 and 446-447.) In these models, the opportunity to free-ride causes effort to be undersupplied. This is not true of modular systems, as we show below.

[17] We could assign the minimal system a value and cost equal to any fraction of v and c. This would complicate the notation without affecting the basic results.

have been chosen for ease of comparison and exposition.)

We need to make additional, important assumptions regarding the time it takes to complete a module, and the frequency and cost of the developers' communications. Specifically, we assume that (1) each developer works on one and only one module at a time; and (2) all developers publish their code as soon as it is available. We will discuss the significance of these assumptions below.

To gain insight we will first derive the pure-strategy equilibria for two modules and two developers, and then vary the number of modules and the number of developers. Figure 2 shows the normal form of the game of involuntary altruism played by two developers when the codebase has two modules, A and B. Each developer may choose: (1) not to work; (2) to work on Module A; or (3) to work on Module B.

Figure 2
Normal Form of a Game of Involuntary Altruism with Perfect Information: Two Developers and Two Modules

Developer 1: \ Developer 2:	Don't Work	Work on A	Work on B
Don't Work	0 , 0	.5v, .5(v–c)	.5v, .5(v–c)
Work on A	.5(v–c), .5v	.5(v-c), .5(v-c)	v–.5c, v–.5c
Work on B	.5(v–c), .5v	v–.5c, v–.5c	.5(v-c), .5(v-c)

The game now has two Nash equilibria indicated by the shaded, off-diagonal cells. In each of these equilibria, both developers work, but on different modules. The full system gets built as efficiently as the original case, but the work is now shared equitably between the two developers. There are no free-riders.

The existence of these new equilibria depends on a certain configuration of values and costs. Not surprisingly, each module's incremental value must exceed the incremental cost of creating it. This is a

stricter requirement than saying that the total system value must exceed total system cost. Among other things, it means that the minimal system's value must be low relative to the incremental value obtained by building out the modules. If modules do not "pay their own way," they will simply not be built, even if they are "included" or "envisioned" in the original architecture.

The results also depend on our assumptions about the timing of the developers' communications. If each module takes half the time to build as the whole system, then, implicitly, the developers must communicate twice as often. What happens if there is a time lag between communications? Then each developer has a fourth alternative: to work on both Modules A and B. In that case, there are two additional pure-strategy equilibria which involve free-riding. In these new equilibria, one developer builds the whole system, and the other does not work. Thus communication technology that is slow relative to the time needed to complete a module's worth of work brings inequitable outcomes back into the set of pure-strategy equilibria.

What if we increase the number of potential developers, N, and modules, j? Suppose, first, that the number of developers is greater than the number of modules, $N > j$. Then, assuming rapid communication, in equilibrium (1) all modules whose value-added exceeds their cost will get built; (2) j developers will build the modules; and (3) the balance of the developers (N–j) will free-ride. All the pure-strategy equilibria are equally efficient. Their inequity increases as N–j increases.

Now suppose that the number of modules is greater than the number of developers, $j > N$. We continue to assume that the developers communicate as soon as they complete each module, and so the building out of the system will proceed in multiple rounds. Let $I_{j/N}$ and $R_{j/N}$ be respectively the integer part and remainder of the ratio j/N. For the first $I_{j/N}$ rounds of work, effort will be shared equitably among developers; then in the last round, $R_{j/N}$ developers will work, and the rest will free-ride.

The per-person payoff of a game with j modules and N (potential) developers depends on whether the person in question works or free-rides. Each free-rider gets the value of the system "for free." Each worker gets the value of the system minus the cost of coding one module, which is inversely proportional to the number of modules. Finally, a Robinson Crusoe developer gets the value of the system minus the cost of coding all of it:

Free-rider:	v
Worker:	v–c/j
Robinson Crusoe:	v–c

If the system has two or more modules, then the payoffs to the workers are higher than payoffs to Robinson Crusoes. *Thus participating in a collective development effort within a modular architecture beats coding in isolation.* Intuitively, a worker on one module benefits from the effort of the workers on the other modules. Each gets the benefit of the whole system for a fraction of the total cost of creating it herself.

Clearly these incentives work only if the target of effort is a non-rival good. The payoff structure depends critically on the assumption that one developer's use of the system does not interfere with another's. Ironically, the non-rival nature of the good is one of the things that makes free-riding possible in the first place.[18] *But a modular architecture for a non-rival good permits something beyond free-riding: it creates the possibility of a particular type of exchange among working developers.* In the idealized setting of this game, each worker contributes one module to the collective process and gains the use of some number of other modules.[19]

Interestingly, in this simple model, the presence or absence of free-riders is immaterial to a worker's decision as to whether to join a collective development process. The factors that tilt a worker's calculations in favor of joining are (1) the presence of other workers; and (2) the existence of a modular architecture that will support both a division of labor and the attribution of incremental value to specific modules.

---

[18] Non-excludability is the other thing that makes free-riding possible.

[19] In effect, a modular architecture introduces variety into the supply of the non-rival good. This in turn makes possible the "exchange" of differentiated items, each of which has a separate and visible value. See our discussion of generalized exchange in the context of Ghosh's "cooking-pot" model in section 7 below.

Although these results are suggestive, the situation we have discussed so far involves a relatively high level of coordination in the pre-play phase of the game. Each developer must know what all the others will do, including what module they will work on, with certainty, before choosing his or her action. There is no room for conjectural mistakes, experimentation, or groping towards a joint outcome. This assumption seems overly strict, especially if, as in the open source development process, the developers are numerous, widely dispersed, not hierarchically organized, and not centrally co-ordinated. Thus in the next section, we will look at equilibrium behavior in a modular system when each developer *does not know* exactly what the others are doing.

4.3 Modularity in a Game with Imperfect Information

In a game of imperfect information with j modules and N potential developers, standard symmetric payoff matrices, such as we have been using, become multi-dimensional and difficult to understand. However, we can work around this expositional difficulty by reformulating the developers' decisions in terms of a sequential game. Consider a system with j modules, each of which has value v/j and cost c/j. Developers arrive in a sequence and, based on a calculation of expected value (described below), decide whether to work or not. For ease of analysis, here and throughout the paper we assume that developers are risk neutral with respect to their payoffs.

If a developer decides to work, he or she randomly selects one of the j modules and develops it. Developers know how many other developers are working on the system, but do not know which modules the others are working on. Thus there is the potential for redundancy in their efforts.

Consider a system of j modules in which n of the N developers *are working*. The expected value of that system, denoted W(j, n), equals the expectation of the number of *distinct* modules selected by the n developers times the value per module:

$$W(j, n) = [E(\text{No. Distinct Modules} \mid j, n)]\ (v/j) \quad .$$

The expected number of distinct modules is a complicated function of n and j. Luckily, we shall see that it does not directly enter a developer's calculation as to whether to work or not.

Suppose that n developers are already working on the system. A new developer arrives: we label him the "index developer." He can decide (1) to free-ride or (2) to work on a randomly selected module. The payoffs to these alternatives are:

Free-ride: $W(j, n)$;

Work: $(v–c)/j + W(j, n) – [\text{Prob(Duplicated Effort)}](v/j)$.

If the developer free-rides, he gets the expected value of the system. If he works, he gets (1) the value minus cost of the module he develops; (2) plus the expected value of the system (created by the other developers); (3) *minus the probability that his module is being duplicated by someone else times the value of the module*. In other words, if the developer works, the expected value of the rest of the system to him is less than if he free-rides, because the system may turn out to include the module he worked on.

Under our assumption of random selection, the probability that none of the n developers has selected the same module as the index developer is $[(j–1)/j]^n$. It follows that the probability that someone else *has* selected the same module, ie., the probability of duplicated effort, is $\{1 – [(j–1)/j]^n\}$. Substituting this expression into the payoff formula, we obtain a decision rule for the index developer. He should work if:

$$W(j, n) \leq (v–c)/j + W(j, n) – \{1 – [(j–1)/j]^n\} (v/j) ;$$

and otherwise free-ride.[20]

The expectation of the value of the system, W(j, n), appears on both sides of this inequality, and thus drops out of the index developer's calculus. Rearranging terms, we can restate the decision rule as follows. The developer should work if and only if:

$$c/v \leq [(j–1)/j]^n . \qquad (1)$$

[20] In the event of a tie, we assume that the developer will not work.

To induce the index developer to work, the cost of developing a module relative to its value must be less than (or equal to) the probability that no other developer is working on that particular module. Sensibly, as the cost of developing a module increases, the index developer is more inclined to free-ride. Similarly, as the number of developers already working, n, increases, free-riding also becomes more attractive. However, as the number of modules, j, increases, the right-hand side of expression (1) increases, and the developer is more inclined to work. In this sense, even with imperfect information and the potential of duplicated effort, *modularity operates to reduce free-riding in a game of involuntary altruism.*

One technical issue arises. The decision rule changes as n changes, ie., as new workers enter the system. However, as long as the developers are symmetric, if the index developer works, none of the developers already working will stop. Thus the decision rule of *the last developer to work* defines an equilibrium for the system as a whole.

We can use the "last developer's decision rule" to calculate how many developers will actively work on a project as a function of its value, cost and degree of modularity. Consider a system with technological parameters v, c and j. If $v-c > 0$, then at least one developer will enter the process and work. Additional developers will enter and work as long as the decision rule summarized in expression (1) indicates that they should do so. As n increases, however, the right-hand side of expression (1) declines monotonically, while the left-hand side is constant. Therefore, for any v, c and j, there exists an $n^{\dagger}$ (not necessarily an integer) such that: (1) if $n < n^{\dagger}$, newly arriving developers will choose to work, but (2) if $n \geq n^{\dagger}$, additional developers will choose not to work. The smallest integer larger than or equal to than $n^{\dagger}$ (otherwise known as the ceiling of $n^{\dagger}$) thus equals the *equilibrium number of developers who will work on the project (over a single time interval)*. We label this number n*.

We can solve for n* by first converting expression (1) into an equality, inverting the exponent, and rearranging terms:

$$n^{\dagger}(v, c, j) = \frac{\ln (c/v)}{\ln [(j-1)/j]} \quad .$$

The smallest integer greater than (or equal to) this number equals the equilibrium number of working developers, n*. The rest of the developers, N–n*, will free-ride. Implicitly, as new developers arrive, the number of workers will grow until it reaches n*, and then stop growing. Hence the characteristics of the

sequential equilibrium do not depend on the *total* number of developers, N, as long as that number is greater than the equilibrium number of workers, n*.

Table 2 shows n* for a range of cost ratios (c/v) and degrees of modularity (j). The table shows that if j=1, the equilibrium number of working developers is never more than one: it never pays to have a second developer work on a non-modular project. (This is consistent with the pure-strategy equilibrium of a non-modular system, derived in section 3 above.) The table also shows that for any cost ratio less than 100%, the equilibrium number of working developers tends to increase as the system is split into more modules. And the impact of modularity is greater as the cost ratio goes down: a system that is less costly relative to its value supports (or calls for) more redundancy.

Table 2
The Equilibrium Number of Working Developers in a Game of Involuntary Altruism as a Function of Cost-to-Value Ratio, c/v, and the Number of Modules, j

No. of Modules	Cost/Value per Module									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
1	1	1	1	1	1	1	1	1	1	0
2	4	3	2	2	1	1	1	1	1	0
3	6	4	3	3	2	2	1	1	1	0
4	9	6	5	4	3	2	2	1	1	0
5	11	8	6	5	4	3	2	1	1	0
6	13	9	7	6	4	3	2	2	1	0
7	15	11	8	6	5	4	3	2	1	0
8	18	13	10	7	6	4	3	2	1	0
9	20	14	11	8	6	5	4	2	1	0
10	22	16	12	9	7	5	4	3	1	0
11	25	17	13	10	8	6	4	3	2	0
12	27	19	14	11	8	6	5	3	2	0
13	29	21	16	12	9	7	5	3	2	0
14	32	22	17	13	10	7	5	4	2	0
15	34	24	18	14	11	8	6	4	2	0
16	36	25	19	15	11	8	6	4	2	0
17	38	27	20	16	12	9	6	4	2	0
18	41	29	22	17	13	9	7	4	2	0
19	43	30	23	17	13	10	7	5	2	0
20	45	32	24	18	14	10	7	5	3	0

It is worth noting that the equilibrium number of working developers can be *higher or lower* than the number of modules. Systems with high cost-to-value ratios will have more modules than developers. In such cases, the modules of the system must be built out in rounds. Assuming that the total number of

modules stays constant, each round will have fewer modules left to be developed than the one before, and thus the number of developers actively working on the system will diminish over time.

Conversely, systems with low cost-to-value ratios will have more developers than modules. These systems may have quite a lot of redundant effort, although (by assumption) the location of the redundancies will only be known at the end of the coding interval.[21] This means that, in equilibrium, modular systems that allocate effort randomly may be quite inefficient. Their efficiency can be improved by creating *ex ante* coordination mechanisms that induce developers to disclose what they intend to work on. (We will return to the question of redundancy when we talk about modular option values below.)

4.4 Comparison to the Robinson Crusoe Alternative

In earlier sections, we worried about whether otherwise isolated developers would voluntarily join this game, that is, would they find it more attractive than the alternative of staying isolated and constructing the entire codebase from scratch. The answer is "it depends." First, consider those processes for which the equilibrium number of developers is one. (These processes are in the upper right corner of Table 2.) In these cases there is no benefit to the first developer from opening up the process by publishing the code. Equilibrium analysis predicts that no other developer will join the effort, hence the only thing that would be accomplished by such a move is a gift of the codebase to free-riders.[22]

In contrast, consider those processes for which the equilibrium number of developers is two or greater. (These are in the lower left of Table 2.) In these cases, the first developer does benefit from opening up the process to others. The calculus of payoffs predicts that at least one other developer will *join and work*. As long as there is a chance that the additional developers will work on different modules from the first, the first developer will get an expected benefit from their efforts.

All systems for which n* is two or greater have at least two modules. *Thus, under our assumptions, only codebases with modular architectures can "support" a game of involuntary altruism.* A modular architecture in effect creates the opportunity for working developers to contribute code for one module and gain

[21] Even with redundancy, some modules may be "missed" on the first pass, hence these systems will generally get built out in rounds as well.

[22] We do not rule out such altruism, but we want to clearly distinguish between pure gifts and generalized exchanges of effort directed at different modules. This allows our argument to be distinguished quite sharply from that of Benkler (2002). (See the discussion of his theory in section 7 below.)

access to code written for other modules. That prospect, if it is an equilibrium outcome, will dominate a developer's fallback alternative, which is to code in isolation.

In the context of this simple model, the per-person value of the system depends crucially on the number of workers and modules, but does not depend on the number of free-riders. Implicitly the workers do not gain (or lose) anything from the presence of free-riders, hence they will (rationally) be indifferent to the number of free-riders "hovering" about.

In addition, as the number of modules grows, the per-person equilibrium payoffs to the workers and the free-riders become closer and closer. To see this, note that in equilibrium, the payoffs to workers and free-riders are as follows:

Free-ride: $W(j, n^*)$;

Work: $W(j, n^*) - c/j$.

Each free-rider gets the value of a system with n* developers and pays no cost. Each worker gets the value of the same system minus the cost of coding her own module. The difference is:

$$\Delta P_{W-F} = - c/j \quad .$$

The workers are worse off than the free-riders by the cost of one module's effort, and the cost of coding one module is, by assumption, in inverse proportion to the number of modules. As a result, the value of the system to the workers and free-riders converges as the number of modules goes up. (In fact, although we derived this result for a specific case, this is a general result that applies to all pure-strategy equilibria in games of this type. It will reappear when we look at games involving option value. In mixed-strategy and evolutionarily stable equilibria, payoffs to workers and free-riders must be equal, hence $\Delta P_{W-F} = 0$.)

In a game with simultaneous (as opposed to sequential) moves, given N developers, there are a very large number of pure-strategy Nash equilibria.[23] All the pure-strategy Nash equilibria have the property that n* of the developers will work and N–n* free-ride. Thus virtually all the results derived above carry over to this new context. However, the Nash convention does not provide a means of selecting among these equilibria. For that, one must impose additional structure on the pre-play phase of the game. For example, when we introduced a sequential arrival process and assumed that the first n* developers would work and the remaining N–n* would free-ride, we implicitly selected one of the pure-strategy Nash equilibria from the very large set of candidates.

In addition to the pure-strategy equilibria of the simultaneous game, there is a single mixed-strategy equilibrium, which is evolutionarily stable. We can solve for the equilibrium "mixing probability," but doing so requires computational methods. However, earlier in this section we showed that as long as the number of workers in equilibrium was two or greater ($n^* \geq 2$), the payoff to a worker from joining a collective development process would be greater than the same worker's payoff from coding in isolation. A similar result holds for the mixed-strategy equilibrium: as long as the expected number of workers in equilibrium is greater than one, the expected payoff to a developer from joining the collective process dominates the Robinson Crusoe alternative of coding in isolation.

Table 3 summarizes the payoffs (per person and per coding interval) in a game of involuntary altruism for a system with an arbitrary number of modules and developers. To simplify the presentation of the table, we have assumed that the number of developers, N, is greater than the number of modules, j. The italicized entries indicated cases for which the expected payoff to a worker or the average player is greater than the Robinson Crusoe payoff. In those cases, developers will rationally prefer joining a collective effort over the alternative of coding in isolation.

[23] The number of pure-strategy equilibria equals the number of distinct combinations of size n* that can be formed in a population of size N. Thus:

$$\text{Number of Pure-Strategy Nash Equilibria} = \frac{N!}{[n^*!\,(N-n^*)!]}\ .$$

Table 3
Comparison of Payoffs in the Involuntary Altruism Game for a System of j Modules and N Developers (j < N)

	Per-person, per-period Expected Payoff
Robinson Crusoe Equilibrium	(v–c)/j
Pure-Strategy Equilibria with Perfect Information (j work, N–j free-ride)	
Free-rider	v
Worker	*v–c/j*
Pure-Strategy Equilibria with Imperfect Information (n* work, N–n* free-ride)	
n* = 1, Free-rider	v/j
n* = 1, Worker	(v–c)/j
n* ≥ 2, Free-rider	W(j, n*)
n ≥ 2, Worker*	*W(j, n*) – c/j*
Mixed-Strategy, Evolutionary Stable Equilibrium (Fraction Workers = □)*	*W(j, □*N)*

5 Option Value in a Game of Involuntary Altruism

In the analysis of the game up to this point we assumed that the values of the system and the individual modules were known *with certainty* at the beginning of play. However, as we noted in section 2, the values of to-be-completed designs are almost never certain. In this section, we will explore the effects of uncertainty and option value on the players' behavior and resulting equilibria in a game of involuntary altruism.

To make our analysis more precise, we will use probability distributions, mathematical expectations and variances to represent developers' perceptions of payoffs and risk. But we must be cautious in using these mathematical constructs and interpreting the results. The seemingly precise mathematical expressions are only analogues of much-less-precise mental objects in the minds of developers. Developers usually have a perception of the expected future utility of a design conditional on

its completion. They also often have a sense of dispersion around that expectation—a sense of things that might turn out better or worse than expected. However, these perceptions are not generally not as precise as mathematical expectations or variances: indeed they may not even conform to the laws of probability. Such perceptions do, however, affect the developers' calculations of future outcomes and their resulting actions. Our hope (and maintained assumption) is that the mathematical models are close enough to the developers' mental models for our analysis to be useful and to have some degree of verisimilitude.

5.1 Option Value Notation

We begin this section by introducing some new notation. We assume that the total value of the system in the eyes of a user can be expressed as the sum of the minimal system value, S_0, plus the incremental value added by the performance of each of j modules:

$$\text{Performance of the System} = S_0 + \sum_{i=1}^{j} X_j^i \quad ; \qquad (2)$$

At the start of the game, we assume that the minimal system has been coded and its value is known. As before, without loss of generality, we normalize its value to zero.

At the same time, the modules have not been coded, hence their eventual values (the X_j^i s in the expression) are uncertain. We assume that at the start of the game, module values can be modeled as random variables. Throughout this section, we will use superscripts to denote realizations of random variables, and subscripts to denote random variables with different distributions. Hence the performance of the system, given in expression (2), is a sum of realizations, one for each module. In this case, all the modules have the same distribution, which is associated with the random variable, X_j. The notation is burdensome, but necessary to avoid ambiguity.

We further assume that the outcome of a development effort, which corresponds to the realization of a random variable X, can have positive or negative value. Code with negative value is not worth incorporating into the system: it subtracts more functionality than it adds. At the end of a coding interval, the developers can observe the realization, X^i, for each module that was coded, and compare that

value to zero. If the new code has positive value ($X^i > 0$), it can be added to the system, and the system's value will increase by that amount. If the new code has zero or negative value ($X^i \leq 0$), it can be discarded, and the developers can try again. Hence the developers can add modules to the system, and mix and match module code as it becomes available.

For simplicity, we assume that developers are risk-neutral expected-value maximizers, and that coding intervals are short enough that we can ignore their time preferences. Let $V(\underline{X}_j; j, k)$ denote the value of a system comprising j symmetric modules and k development efforts per module. The payoffs to the modules are determined by a vector of independent, identically distributed random variables, $\underline{X}_j$. There is one element in this vector for each of j modules. In this notational system, $V(\underline{X}_1 ; 1, 1)$ denotes the value of *one development effort* (k=1) undertaken for a *non-modular system* (j=1) whose payoff is the random variable X_1.

In a system of j symmetric modules, let lower-case $v(X_j; k)$ denote *the value of a single module* if k development efforts are targeted at it. X_j in this case is not a vector; it is a single random variable corresponding to the payoff of the module in question. Again the outcomes of the individual development efforts are assumed to be independent draws from the probability distribution of this random variable.

Under these definitions, in a non-modular system with $S_0=0$, $V(\underline{X}_1; 1, 1)$ and $v(X_1; 1)$ are related as follows:

$$V(\underline{X}_1; 1, 1) = v(X_1; 1) = \text{Emax}(X_1^1, 0) \quad ,$$

where "Emax $(X_1^1, 0)$" denotes *the expected value of the maximum of one realization of the random variable X_1 and zero*.[24]

We have said that, when outcomes are uncertain, "duplication of effort," in the sense of

---

[24]More formally, Emax(X,0) is calculated by weighing each potential outcome of X that is greater and zero by its probability and summing over outcomes. For a continuous distribution, one integrates the expectation of the random variable over the positive part of its support:

$$\text{Emax}(X, 0) = \int_0^\infty y\, f_X(y)\, dy \quad ;$$

where $f_X(y)$ is the density function of the probability distribution of the random variable, X.

mounting several design experiments aimed at the same target, may be desirable. Thus we need a symbol to denote the expected value of the best of several design experiments. Let Q(X ; k) denote the "expectation of the highest value of k trial designs," as long as the highest value is greater than zero. Each trial design in this case is assumed to be drawn independently from the probability distribution of the random variable X.[25] Formally:

$$Q(X; k) = \text{Emax}(X^1, \ldots X^k, 0) \quad ;$$

where $X^1, \ldots X^k$ are the realizations of the individual trial designs.

It will also be convenient to be able to denote the difference in the value of a module caused by increasing k by increments of one. Thus let $\Delta v(X; i)$ indicate the increase in v(X; .), caused by increasing the number of design experiments from i–1 to i:

$$\Delta v(X; i) = v(X; i) - v(X; i-1) \quad .$$

It follows by substitution that, for a non-modular system with $S_0 = 0$, all of the following are true:

$$\begin{aligned} V(\underline{X}_1;1, k) &= v(X_1; k) \\ &= Q(X_1, k) \\ &= Q(X_1, 1) + [Q(X_1, 2) - Q(X_1, 1)] + \ldots + [Q(X_1, k) - Q(X_1, k-1)] \\ &= v(X_1; 1) + \Delta v(X_1; 2) + \ldots + \Delta v(X_1; k). \end{aligned}$$

The above relationships hold for general probability distributions. However, it will sometimes be useful to specialize the distribution and assume that X has a normal distribution with mean zero and variance σ^2: $X \sim N(0, \sigma^2)$. Conveniently, Q(X; k) for such a distribution is simply $\sigma Q(k)$, where σ is the standard deviation of the normally distributed random variable, and Q(k) denotes the expected value of

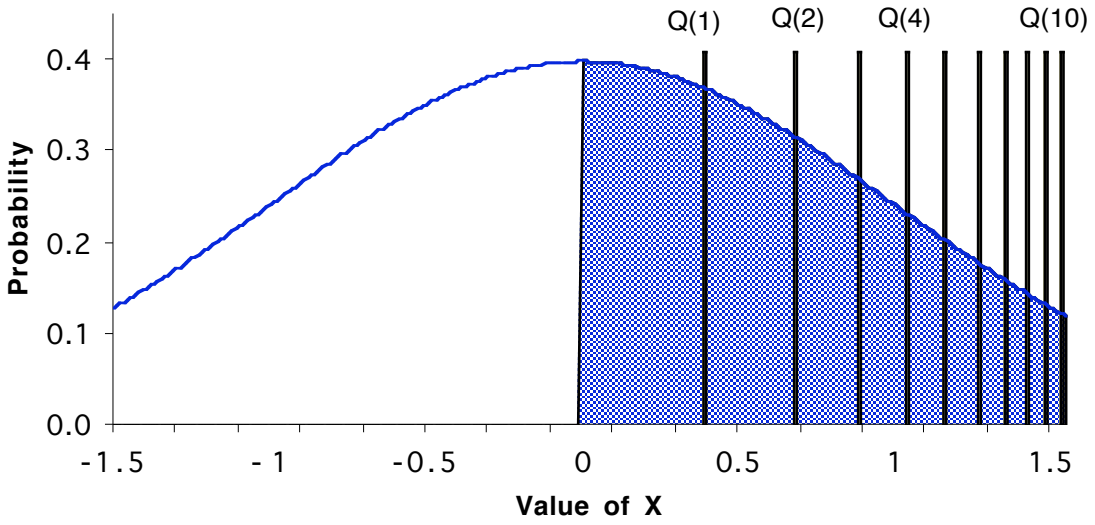
[25] The highest of k draws from a given distribution is well known in statistics: it is the "maximum order statistic of a sample of size k." Our definition differs slightly from the one found in statistics textbooks in that we require the "highest of k" to be also greater than zero. See, for example, Lindgren (1968), Chapter 8.

the highest of k independent draws from a so-called "standard" normal distribution with mean zero and variance one.[26]

For any normal distribution, the standard deviation, σ, is a measure of dispersion or risk. As we discussed in section 2, for any option, more risk creates more upside potential, and thus increases value: this can be seen from the fact that, holding k constant, increasing σ increases the expected payoff $\sigma Q(k)$. Thus for normal distributions, the standard deviation, σ, is a scalar measure of the "technical potential" of the design process.

Figure 3 graphs Q(k) for values of k ranging from 1 to 10. Note that Q(k) increases in k, but at a decreasing rate. In fact, this property holds true not only for the normal distribution but for any distribution.[27]

Figure 3
Q(k) - The Expected Value of the Best of k Independent Draws from a Standard Normal Distribution



[26] Formally

$$Q(k) = k \int_0^{\infty} z\,[N(z)]^{k-1}\, n(z)\, dz$$

where N(z) and n(z) are respectively the standard normal distribution and density functions (see Lindgren, 1968, for details).

[27] Aoki and Takizawa (2002).

5.2 Option Values in a Non-Modular Architecture with Two Developers

We are now ready to look at the effect of option value on the simplest version of the game of involuntary altruism, one with two developers and one module. To facilitate comparisons, we assume that the *expected value of one design experiment in this system* equals the value of the non-modular system in the previous sections:

$V(\underline{X}_1; 1, 1) = v(X_1; 1) = v$ (from the previous sections) .

As before, each developer can work or free-ride. If both free-ride, neither gets any value. If one works and the other free-rides, each gets the expected value of *one design experiment*, $v(X_1; 1) = v$, and the worker pays the cost c. However, if both work, they each have the option to take whichever codebase turns out to be more valuable after the fact. Thus their efforts are no longer strictly redundant: each gains in expectation from the work done by the other. In the payoff matrix, the expected value of the second worker's effort is denoted $\Delta v(X_1; 2)$, and the expected payoff to each worker, net of his or her cost, is $v + \Delta v(X_1; 2) - c$.

The normal form of this game is shown in figure 4. In three of the four cells, the payoffs are the same as in the simple game shown in figure 1, while in the fourth <Work, Work> cell, the payoffs are higher. The higher value in the <Work, Work> cell arises because the ability to select the better of the two designs creates an option value, equal to $\Delta v(X_1; 2)$.

Figure 4
Normal Form of a Game of Involuntary Altruism: One Module with Option Value and Two Developers

	Developer 2: Free-ride	Developer 2: Work
Developer 1: Free-ride	0, 0	v, v–c
Developer 1: Work	v–c, v	$v + \Delta v(X_1; 2) - c$, $v + \Delta v(X_1; 2) - c$

The nature of equilibrium in this new game depends on the magnitude of the option value, $\Delta v(X_1; 2)$ in relation to the developer's cost of effort, c. If $\Delta v(X_1; 2) > c$, <Work, Work> will be the unique Nash equilibrium of the game. However, if $\Delta v(X_1; 2) \leq c$, the nature of the equilibrium will revert to that of the original game: There will be two pure-strategy Nash equilibria in which one developer works and the other does not and a unique mixed-strategy equilibrium in which each developer works with probability π^*. As in the initial game, the mixed-strategy equilibrium will correspond to an evolutionarily stable strategy for pairs of developers randomly selected from a population. We can solve for this endogenous probability using standard methods:

$$\pi^* = \frac{v-c}{v-\Delta v(X_1; 2)} > \frac{v-c}{v} = \pi^* .$$

Here π^* denotes the equilibrium fraction of workers in the game without option value. Because $\Delta v(X_1; 2)$ is greater than zero, π^* greater than π^* for any probability distribution of payoffs.

How do the expected payoffs of this game compare to the expected payoffs in a Robinson Crusoe environment? A developer coding in isolation obtains expected payoff v–c. This is the same as the payoff to the worker in the pure-strategy equilibria. Thus there is no incentive for an isolated developer to join a

collective effort in which he or she works with probability one. However, in the mixed-strategy equilibrium, each developer works with probability π^*, which is less than one. In this case, the expected payoff to workers and free-riders alike is:

$$\text{Payoff to Workers and Free-riders in Evolutionarily Stable Strategy with Option Value} = \frac{v(v-c)}{v - \pi v(X_1; 2)} .$$

Because $\pi v(X_1; 2) > 0$, this payoff is higher than the expected payoff to Robinson Crusoe developers. Thus in contrast to our previous finding for non-modular architectures, *in the presence of option value, there is incentive for a developer to leave an isolated environment and join a collective development effort, even if there is some amount of free-riding in the collective process.* The reason it makes sense to join a collective effort is that, with option value, parallel work is not necessarily redundant. A Robinson Crusoe developer loses out on the possibility that another worker's finished codebase might be superior to her own. This possibility in turn tips the balance of payoffs in favor of joining a collective process.

Finally, if both developers work, then there is no free-riding in equilibrium, and each developer gains access to the other's code with probability one. In that case, because the other's code has option value, participating in the collective effort dominates coding in isolation.

5.3 Option Values in a Modular Architecture

What happens if modularity and option values are combined? From the fact that modularity and experimentation are economic complements (more of one makes the other more valuable), intuitively it would seem that modularity combined with option values should make a collective development effort even more attractive than modularity alone or option value alone. This section shows that this intuition is correct.

Suppose that a system is partitioned into j modules. As above, we assume that the value of each module is independent of any other module's value, and the whole system's value equals the sum of the individual module's values. We also assume that the distribution of the system's value is unchanged by a modularization of any degree. Thus for any number of modules, j:

$$X_1 \overset{\text{dist.}}{=} X_j + \ldots + X_j \quad . \qquad (3)$$

The Value of a Non-modular System (a random variable)		The Value of a System of j Symmetric Modules (the sum of j i.i.d. random variables)

Under these distribution-conserving assumptions, a modular architecture will always yield a higher total value than the corresponding non-modular architecture. Intuitively, the designers of the modular system could tie their own hands and commit to take *all* or *none* of the realizations of the individual module designs. The value of a modular system under this constraint would then have the same distribution as the value of the non-modular system, as shown in equation (3).

This observation is a special case of a general result in option theory, proved by Robert Merton in 1973.[28] The result is usually stated as "a portfolio of options is worth more than an option on a portfolio." In this instance, the non-modular architecture gives developers one large option: to take the whole new system or reject it and fall back on the minimal system. This is the "option on a portfolio." By contrast, the modular architecture gives developers many smaller options, each of which can be implemented or rejected. This is the "portfolio of options." Regardless of the underlying distributions, as long as equation (3) holds, a modular architecture will dominate the corresponding non-modular architecture because of the option values implicit in the ability to mix and match module realizations. Furthermore, more modules will yield more more option value, as long as the distributional identity holds.

We can gain further insight by specializing the distributional assumptions. For example, suppose that in a system of j modules, the payoff to one development effort on one module is normally distributed with mean zero and variance, σ^2/j: $X_j \sim N(0, \sigma^2/j)$. Then, for all positive integers, j, the distribution of the sum of j realizations of the random variable X_j has the same distribution as a single realization of the random variable X_1:

$$X_1 \overset{\text{dist.}}{=} X_j + \ldots + X_j \sim N(0, \sigma^2), \quad \text{for all j.}$$

Under these distributional assumptions, we have said, the value of k design experiments targeted

[28] Merton (1990), Chapter 8.

at a single module equals the standard deviation of the module's payoff times the expected value of the highest of k draws from a standard normal distribution:

$$v(X_j; k) = (\sigma/j^{1/2})\, Q(k) \quad .$$

And the value of the whole system is simply j times this expression:

$$V(\underline{X}_j; j, k) = j\, v(X_j; k) = \sigma\, j^{1/2}\, Q(k) \quad .$$

Now, in the context of a game of involuntary altruism, suppose that an arbitrary number of developers, N, can work. How many will voluntarily choose to work, and what will the value of the system then be? For simplicity, as in section 4.3 (the game with imperfect information), we assume that developers enter the system sequentially. However, in contrast to that analysis, here we assume that the developers have *perfect information*: each new index developer knows how many other developers are currently working and which modules they are working on. On the basis of that information, the index developer decides to work or free-ride. If she works, she also selects a module to be the focus of her effort. As before, the cost of a development effort for one module is c/j.

Suppose k developers are already working *on each module* of the system.[29] The payoffs to the next index developer, conditioned on her actions, are:

Free-ride: $V(\underline{X}_j; j, k)$;

Work: $V(\underline{X}_j; j, k) + \sigma v(X_j; k+1) - c/j$.

If the index developer free-rides, she gets the value of the system "for free". If she works, she gets the value of the system plus *the incremental value of adding her effort to one module*, $\sigma v(X_j; k+1)$, minus the cost of her effort, c/j.

[29] Because the modules are symmetric, with perfect information, developers will distribute their efforts evenly across the modules. Thus each cohort of j developers will behave in the same way.

From these payoffs, as in the previous section, we can calculate the index developer's optimal decision rule. She should work only if her incremental contribution to the system equals or exceeds the cost of her effort:

$$c/j \leq (\alpha/j^{1/2})\,[Q(k+1) - Q(k)] \quad ;$$

or, equivalently:

$$c \leq \alpha\, j^{1/2}\,[Q(k+1) - Q(k)] \quad . \qquad (4)$$

The index developer's decision rule has the same form for both modular (j≥2) and non-modular systems (j=1). However for given c, α, and k, higher j makes it easier to satisfy the inequality, hence easier to justify working.[30] From this it follows that more modular systems (higher j) generally support more parallel development efforts (higher k), other things equal.[31]

For fixed cost, c, technical potential, α, and degree of modularity, j, let us now define k_j^* (an integer) as the lowest number of developers that reverses the inequality in expression (4). This number corresponds to *the number of developers who will work on each module in equilibrium*. The *total* number of developers working on the system , denoted n_j^*, is simply the number of developers per module times the number of modules:

$$n_j^* = k_j^*\, j \quad . \qquad (5)$$

The fact that more modular systems support more parallel effort implies that k_j^* in equation (5) is weakly increasing in the number of modules. Necessarily, then, n_j^* is *strictly increasing and weakly convex in j*. Thus

[30] This follows from the fact that the right-hand side of expression 11 is multiplied by $j^{1/2}$, which is greater or equal to one and increasing in the degree of modularity, j.

[31] Formally, if $c \leq \alpha\, j^{1/2}\,[Q(k+1) - Q(k)]$ for some k and j= j*, then the inequality holds *a fortiori* for all j > j*. Thus increasing modularity never induces less parallel development effort. Conversely, suppose we reduce modularity. *As j declines, it is possible that the inequality will reverse.* In that case, to make the developers' actions consistent with their decision rule, the number of developers working on each module, k, must go down. Hence a less modular architecture may induce less parallel effort than a more modular architecture. QED

as the number of modules goes up, the equilibrium number of working developers in a system with option values goes up at a (weakly) increasing rate. This result is demonstrated in table 4, which computes n_j^* for a range of cost-to-technical-potential ratios and modularity.

Table 4
The Number of Developers Working in Equilibrium, n_j^*, as a Function of the Cost-to-Technical-Potential Ratio, c/□, and the Number of Modules, j

No. of Modules	Cost/Technical Potential				
	25%	50%	75%	100%	150%
1	2	0	0	0	0
2	6	2	0	0	0
3	12	3	0	0	0
4	16	8	4	0	0
5	25	10	5	0	0
6	30	18	6	0	0
7	42	21	7	7	0
8	48	24	16	8	0
9	54	27	18	9	0
10	70	30	20	10	0
11	77	44	22	11	0
12	84	48	24	12	0
13	104	52	26	26	0
14	112	56	42	28	0
15	120	60	45	30	15
16	128	64	48	32	16
17	153	85	51	34	17
18	162	90	54	36	18
19	171	95	57	38	19
20	180	100	60	40	20
21	189	105	63	42	21
22	220	110	66	44	22
23	230	115	92	46	23
24	240	120	96	72	24
25	250	150	100	75	25

Sensibly, the table shows that as the cost-to-technical-potential ratio increases, the equilibrium number of working developers goes down: indeed, in many cases the equilibrium number is zero. However, holding the cost-to-technical-potential ratio fixed, the equilibrium number of working developers tends to increase as the system is split into more modules. For example, if the cost-to-technical-potential equals the 100%, a non-modular system will not justify any development effort at all.

(In such a system, the expected value of one effort is only .3989 and the cost of that effort is one.) But a system with the same total cost and technical potential, which is split into 7 mix-and-matchable modules will attract one developer per module in equilibrium. And a system of 25 modules will attract three developers per module, for a total of 75 working on the system.

Although it is tempting to compare tables 2 and 4, in fact they represent the results of quite different thought experiments. In table 2, we assumed that developers had *imperfect* information about which modules other developers were working on, but knew the *ex post* value of any module with *certainty*. In that case, efforts directed at the same module were purely redundant: the developers would have eliminated the redundancies if they could have done so. In table 4, by contrast, we assumed that the developers had *perfect* information about what others were working on, but the value of any module was *uncertain*. In this case, as we have seen, option value can justify parallel efforts directed at the same module. The real world, of course, is characterized by both imperfect information and uncertainty about *ex post* module values. However, an analytic model incorporating both of these features is too complicated to offer much insight.

6 Will Developers Voluntarily Reveal Their Code?

Up to this point in our analysis we have assumed that, if a developer worked on any module of a codebase, his or her work product would be automatically revealed to the other developers (plus any free-riders) at the end of the coding interval. Under this assumption, we showed that a cooperative development effort can be sustained if the system is "modular enough" or has enough option value relative to the cost of building modules. However, as we indicated, the assumption of automatic revelation is counterfactual: in reality, those who write code do not have to reveal it. Moreover, there is always some cost to a developer from posting a block or a snippet of code. The cost may be as low as the cost of composing and sending an email, but it is there nonetheless.

Voluntary revelation and costly communication together create an interesting problem for the collective development process. This can be seen most starkly if we go back to the two-developer, two-module case with perfect information. For the sake of argument, let us assume that the two developers have shown up, gone to work *on different modules*, and completed their coding tasks. Each has a finished

module in hand, and is looking to gain from the work of the other. The costs of their coding efforts are sunk at this point, hence can be ignored for purposes of this analysis.

Each developer must choose whether to reveal the contents of his or her finished module to the other. Let v denote the value to each developer of the whole system; let .5v denote the value of one module; and let .5r denote the cost of communicating the code for one module. The normal form of this game is shown in figure 5. It is straighforward to verify that this is a one-shot Prisoners' Dilemma (PD) game,[32] whose unique Nash equilibrium is <Don't Reveal, Don't Reveal>.

Figure 5
Normal Form of a Game of Voluntary Disclosure: Two Modules and Two Developers

	Developer 2: Don't Reveal	Developer 2: Reveal
Developer 1: Don't Reveal	.5v, .5v	v, .5v–.5r
Developer 1: Reveal	.5v–.5r, v	v–.5r, v–.5r

Unlike the game of involuntary altruism analyzed above, the equilibrium of this game is not affected by changes in the number of modules nor by increased option value. Regardless of the codebase's architecture, a Prisoners' Dilemma lurks at the point where a developer must voluntarily reveal his code and pay the cost of communication. Fortunately, however, there are many well-known routes around a Prisoners' Dilemma. We will review them in the sections below, paying particular attention to how they are affected by modularity or option value in the code architecture.

[32] Technically, as long as r < v, the game conforms to the definition of a Prisoners' Dilemma.

6.1 Inducing Revelation in a One-shot Game

In a one-shot game, "fixing" a Prisoners' Dilemma involves adding benefits or reducing the costs of cooperation to the point where the equilibrium shifts to one of mutual cooperation. In the context of the collective development process, cooperation means voluntarily revealing code. Thus if the development process can provide *benefits to revealing* that offset the cost of revealing, the cooperative outcome <Reveal, Reveal> becomes the equilibrium.

Let f (signifying fame and fortune) denote the benefit of publishing code for the whole system. Consistent with our assumptions above, let f/j be the benefit of publishing code for one module in a system of j modules: for example, .5f would then be the benefit of publishing code for one module in a system of two. It follows that, for any j:

If $f > r$; <Reveal, Reveal> is the equilibrium .

According to this view of the open source development process, "fame and fortune" benefits only have to compensate developers for the costs of communication. As the model is constructed, the perceived value of the codebase plus the ability to "exchange" effort are what bring developers into the collective process. Within the process, the perceived values of individual modules are what elicit their coding effort. Thus only one step in the overall process—communication—requires compensation other than the simple right to use the codebase.

In the real world of open source development, there are several mechanisms whose effect is to reduce the cost of revelation, and several others that increase rewards to those who publish their code, report bugs, or submit patches. Many of the cost-reducing mechanisms are technological: the Internet, email, and organized, easy-to-find newsgroups. With a minimal number of keystrokes, a developer can send a missive that quickly reaches all interested parties.

In this connection, it is worth noting that a *laissez faire* stance toward free-riders substantially reduces the costs of communication by removing the need to police the boundaries of the group. At the same time, open source software licenses serve to protect the group's work product for use of its members. The protection is not against free-riding, which does not obstruct the creators' use of the

codebase, but against the conversion of the codebase into someone else's intellectual property.[33]

Rewards for communication may be private or public. Privately, some people are loath to free-ride: if they use a good, they feel obliged to give back something of value. For these individuals, the positive feeling associated with discharging a debt can offset the cost of revelation, especially if that cost is small.[34] Private feelings of altruism or the wish to be affiliated with a group can also offset small costs of revelation.[35]

Many observers of the open source movement contend that public recognition of authorship is key to the functioning of the open source development process. Those developers who make the effort to communicate may benefit from public recognition in three ways. First, their reputation may be enhanced leading to higher pay in the labor market.[36] Second, they may enjoy higher status and be recognized by members of their own group or "tribe".[37] Finally, contributors may obtain affirmation of the value of their work and the meaningfulness of their effort.[38] These three potential benefits are not mutually exclusive: any or all may matter to an individual developer.[39]

A number of empirical researchers are presently trying to disentangle the motives listed in the previous two paragraphs, in order to understand better how open source communities function and what is essential to their vitality. In our model, however, a particular configuration of motives is not essential to the success of the collective process. If *any* of these benefits matters to *even a subset* of developers, they can serve to "tip" the communication subgame into a cooperative equilibrium.

How does code architecture affect the costs and benefits of communication? First of all, a modular architecture substantially reduces the effort associated with a minimal *communication*, just as it

---

[33] By his own account, the desire to protect a user's rights to use, copy, and modify code is what caused Richard Stallman to invent the so-called GNU General Public License (GPL) (Stallman, 1999). As of today, the GPL is the dominant license used to protect open source software. In a survey of 39,000 open source software projects conducted in May 2002, Lerner and Tirole (2002b) found that the GPL and its variant, the "Lesser GPL", applied to 70 – 80% of the code. O'Mahony (2003a) argues forcibly and convincingly that, far from "giving away" their work, open source contributors take care to define ownership rights and to invest in governance mechanisms that precisely meet their needs. Those needs include, most importantly, guaranteed future access to and responsible stewardship of the codebase.

[34] Lakhani and Hippel (2003); Shah (2003b).

[35] Benkler (2002).

[36] Lerner and Tirole (2002a).

[37] Raymond (1999).

[38] Shah (2003b).

[39] Shah (2003a).

reduces the effort associated with a minimal *contribution*. Emails and postings can be shorter if the code is modular. This reduction in the cost of communication may make it easier to elicit contributions based on feelings of reciprocity, altruism or a wish to be affiliated.[40]

A modular architecture also allows many contributors to be recognized for their work on different parts of the system. Nevertheless, the "fame and fortune" value of a single contribution undoubtedly goes down as the number of modules goes up, and as a result, developers who are "hungry for stardom" might prefer to code in isolation. Offsetting this last effect, however, is the fact that option values, which justify multiple efforts directed at the same target, implicitly create tournaments in which developers can compete to provide the best design.[41] Indeed, a modular architecture with substantial option value in the modules creates many such tournaments, involving different areas of specialization and different levels of difficulty. Also, by reducing the scale of a typical tournament, modularity increases the frequency and specificity of competitive interaction and feedback. Finally, an open source community provides would-be competitors with ready-made opponents, judges, and an audience. Thus "stardom" may be more easily achieved within an active community than by working alone.[42]

6.2 Inducing Revelation in a Multi-stage Game

The other well-known way to induce cooperation in a Prisoner's Dilemma game is to convert a one-shot encounter into repeated interactions. Within the game-theoretic formalism, repetition makes possible contingent strategies. For example, "tit for tat," which is essentially the strategy "I'll cooperate as long as you do, and stop if you stop," is a repeated-game strategy that is robust against a wide range of other candidates.[43] In an n-person game, symmetric "tit for tat" induces cooperation among rational players as long as the game has no definite endpoint, and the probability of continuation at each point in time is high relative to the discount rates of the players. However, many other contingent strategies also

[40] Benkler (2002).

[41] Aoki (2001); Aoki and Takizawa (2002a, b).

[42] Or as Eric Raymond acerbically notes: "[S]oftware that nobody but the author understands or has a need for is a non-starter in the reputation game." (Raymond, 1999, p. 112.)

[43] Axelrod (1984); Baker, Gibbons and Murphy (2002).

support cooperation in multi-stage settings of the Prisoners' Dilemma game.[44]

Code architecture affects both the repetitive structure and the nature of the horizon in a collective development process. In the first place, a modular architecture coupled with rapid communication technology can convert a one-shot (or short-term) game into a multi-stage game. In the second place, option values in the modules can convert a game with a definite endpoint into one with an indefinite time horizon.

To see how architecture can convert a one-shot game into a repeated game, first consider a codebase with modules but no option value, such as we discussed in section 4. In theory, with perfect information, if the number of developers, N, was greater than the number of modules, j, all the modules of the codebase would be built in one coding interval. But that analysis did not take into account the barriers to revelation. Understanding that fellow workers might not reveal their code after the fact, some developers might reasonably doubt the effectiveness of the collective process, and continue to code alone. Thus the number of developers willing to participate in the process might be substantially less than j.

For the sake of argument, let us say that the number of participants was only two. Two developers could initiate a mutual "tit for tat" strategy, with each one publishing in response to the other. The iterated game between the two developers could then continue for j/2 rounds until all the modules had been coded. However, those developers would still face a definite horizon, and thus have incentives to defect in the last round, the next to the last round, and so on.[45]

Design uncertainty changes the nature of the horizon. If the outcome of each module design effort is uncertain, then the endpoint of the game cannot be predicted with certainty. Rationally, the developers will continue coding until every module reaches a level of performance such that the expected value of one more trial design is less than the cost of generating the design.[46] Whether that point has been reached can only be known after the fact, hence the end of the development process necessarily comes as a surprise.

---

[44] Axelrod (1984) Appendix B.

[45] Gintis (2000) pp. 103-110, 439.

[46] Loch, Terwiesch and Thomke (2001).

At the same time, however, design uncertainty means that some design efforts are bound to fail. This in turn complicates the developers' inferences and their strategies. For example, if one developer submits a poor design (or no design) on one round, did he put in a good faith effort and fail, or did he reduce his effort in hopes of free-riding on the other party? At what point should the punishment phase of "tit for tat" be invoked, and how long should it continue? And when should a developer who has not benefited from the other party's contributions stop paying the cost of communication and go back to coding alone?

Interestingly, these issues, which may make the cooperative equilibrium of a two-person repeated Prisoners' Dilemma game somewhat fragile, can be addressed by increasing the number of working developers. For example, suppose n developers are engaged writing code, and each uses the following "tit for tat"-type of rule for communication: "Publish unless there is no other publication of value on the last round, then stop publishing, but continue to code." If developers are pursuing independent design efforts, the probability that no one will have anything valuable to post goes down exponentially with the number who are participating. *Thus a large number of developers can form a mutually self-enforcing cooperative group that overcomes the Prisoners' Dilemma inherent in voluntary communication.*

To summarize, higher levels of modularity and option value reinforce one another and increase the number of worthwhile design tasks. Thus a modular, option-laden code architecture makes possible: (1) many rounds of development (a repeated game); (2) an uncertain time horizon (supporting "tit for tat"-type strategies of cooperation); and (3) a relatively large number of participants on each round (reducing the fragility of the tit-for-tat strategies).

6.4 Getting a Multi-stage Game Started

Higher levels of modularity also reduce the maximum unreciprocated exposure of any working developer. This in turn can help a collective development effort to get started. In effect, if the unit of contribution is small, a developer can "test" a potential collective environment at low cost in terms of his time and effort. But the aggregation of these individual "tests" is what will create, or at least initiate, the collective process. Therefore a context that is easier for individual developers to "test" is also more likely to give rise to an active open source project and community.

To be more precise, suppose an architect wanted to have the use of a particular codebase, and knew that a collective development process was a potential alternative to coding the whole system alone. A reasonable strategy for this architect is (1) to create an architecture and perhaps even a minimal system;[47] and (2) incur the cost of publishing his preliminary work in order to "ping" the environment and see if others will join the effort. If no one responds, the architect will have lost only the cost of his communication, and he can revert to coding alone. But, if the architecture of the codebase appears attractive to other developers, they might respond by contributing new code. Their purposes would be, first, to begin the sequence of "tit for tat" with the architect (thereby keeping the architect involved and publishing), and second, to make the collective process itself more vital and attractive to still other developers who might join. The point is that for an incomplete codebase with lots of work still to be done, the value to all users of initiating a collective development process is very high. This "shadow of the future," as Axelrod calls it, can overcome the cost of communication, particularly if that cost is low.

It almost goes without saying that the most attractive architectures for all concerned are those with higher degrees of modularity and high option values. In a multi-stage game, higher levels of modularity (higher j) reduce the cost of effort per developer per coding interval. And more option value makes cooperation in the repeated game more self-enforcing in three ways: (1) by increasing the number of developers who are likely to be coding; (2) by creating uncertainty as to the potential value of others' work; and (3) by making the time horizon of the repeated interactions indeterminate.

7 Other Theories and Models of the Open Source Development Process

The purpose of this section is to describe other models and theories that explain the open source development process and show how our theory is consistent or inconsistent with these others. The

[47] Eric Raymond described the codebase requirements for starting a new open-source project as follows: "When you start community-building, what you need to be able to present is a *plausible promise* Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future." (Raymond, 1999, p. 58). "Can be evolved into something really neat" expresses the *option values* inherent in the architecture, as perceived by the early developers.

literature on this topic is vast, and thus any review must necessarily be incomplete.[48] We have chosen to highlight works that are representative and that influenced our own models.

For purposes of comparing theories, it is useful to break down the open source development process into several stages or steps. Different theories and models apply to different steps, and in this sense are complementary. The steps in an open source development process are as follows: (1) the design of the architecture of the codebase and creation of a (possibly incomplete) minimal system; (2) the contribution of the initial codebase under some form of license to the open source community at large; (3) the writing of new code by developers in the community; (4) the posting of new code by developers; (5) its integration into the codebase; (6) the testing of the codebase; (7) the posting of bug reports; (8) the revision (patching) of the codebase to eliminate bugs. Broadly speaking, as a result of steps 1-2, a codebase comes into existence in the open source software community; as a result of steps 3-5, the codebase grows and gains in functionality; as a result of steps 6-8, the functional gains are solidified and the robustness and reliability of the codebase improves.

Figure 6 shows these steps in sequence. However, it is wrong to think of the overall process as one in which each step follows the one before in a linear fashion until the process is complete. In a real open source development process, many activities can and do go on in parallel in different parts of the system. Furthermore, steps 3 to 8 may be repeated over and over again until the developers see nothing worthwhile left to do.

Figure 6
Stages in the Open Source Development Process

1	2	3	4	5	6	7	8
Design	Contribute	Code	Post	Integrate	Test	Report Bugs	Patch

As we indicated at the beginning of this paper, our model applies to the early stages of the development process: the design of the initial codbase and developers' decisions to write and publish code. We have argued that, in these stages, the core of the development process can be thought of as two linked games played within a codebase architecture. The first game involves the implicit exchange of

[48] A whole library of related work may be found at http://opensource.mit.edu/online_papers.php .

effort directed at the modules and option values of a codebase; the second is a Prisoners' Dilemma game triggered by the irreducible costs of communicating. The implicit exchange of effort among developers is made possible by the the non-rivalrous nature of the codebase and by the modularity and option values of the codebase's architecture. This exchange creates value for all participants. In contrast, the Prisoners' Dilemma is a problem that must be surmounted if the exchanges are to take place. It can be addressed through a combination of reducing the costs of communication, providing rewards, and encouraging repeated interactions. Finally, the initial design and "opening up" of a codebase can be seen as a rational move by an architect who is seeking to "ping" the environment in hopes of initiating exchanges of effort with other developers.

The most detailed and comprehensive theory of the open source development process has been put forward by Eric Raymond in three essays first published on the worldwide web.[49] In characterizing "hacker culture," Raymond builds a self-consistent picture of a society of developers (a "tribe"), with individually complex motives and agendas, held together by an intricate structure of customs and social norms. It is impossible to do justice to Raymond's theory in a few sentences, but its most salient points are as follows. First, he suggests that good code is generated because developers are "scratching an itch,"that is, because they need, value or want the code itself.[50] Second, he conjectures that "given enough eyballs, all bugs are shallow." This is a shorthand phrase that captures the idea that a massively parallel, fast-cycle process of coding, releasing, peer review, and debugging, can in some cases outperform commercial software coding practices.[51] Famously, he suggests that open source developers participate in an evolved "gift culture" or "reputation game," that is, they publish code and compete to improve code in order to achieve status within the community or "tribe" of "hackers."[52] Finally, almost as an aside, he mentions that "the economic problems of open-source … are free-rider (underprovision) rather than congested-public-goods (overuse)," and connects these problems to the costs of communication:

> The real free-rider problems in open-source software are more function of friction costs in submitting patches than anything else. A potential contributor with little stake in the reputation game, … may … think, "It's not worth submitting this fix because I'll have to clean up the patch,

[49] The essays, "The Cathedral and the Bazaar," Homesteading the Noosphere," and "The Magic Cauldron," may be found in Raymond (1999) or at http://armedndangerous.blogspot.com/ .

[50] Raymond (1999) (CatB) p. 32.

[51] ibid. (CatB) pp. 36-44.

[52] ibid. (Homesteading) pp. 99-118.

write a ChangeLog entry, and sign the FSF assignment papers."[53]

By design, our linked-games-in-an-architecture model formalizes and amplifies parts of Raymond's theory, using the economic tools of design and option valuation in combination with game theory. Taking the last point first, the fact that software is a non-rival good, hence subject to free-riding, lies at the heart of our analysis. Indeed, we showed that free-riding can be reduced, and the underprovision problem eliminated (at least theoretically) by a modular, option-laden architecture. Thus a modular structure and/or option value justifies the formation of a collective development process amongst a set of Robinson Crusoe developers.[54] And the more modular and option-laden the architecture is, the more attractive and valuable the collective process will be to those developers.

Beyond these results, the notion of "scratching an itch" appears in our assumption that developers are users who will choose to code (a module) if and only if the target of their effort is personally appealing. The concept of massively parallel, fast-cycle coding is captured by our notion of highly modular codebases with communication speeds matched to the coding interval. (However, we do not model the peer review and debugging aspects of the process in a substantive way.) Finally Raymond's "reputation game" appears as one way (but not the only way) of addressing the Prisoner's Dilemma of communication.

Our models were also greatly influenced by an idea advanced by Rishab Ghosh: the so-called "cooking-pot model," of generalized exchange on the Internet.[55] Crucial for us was the link Ghosh made between generalized exchange and non-rival goods. In the first place, he claimed:

> People "put it" to the Internet because they realize that they "take out" from it. Although the connection between giving and taking seems tenuous at best, it is in fact crucial. …[W]hatever resources there are on the Net for you to take out, without payment, were all put in by others without payment. … So the economy of the Net begins to look like a vast cooking-pot, … [which] keeps boiling because people keep putting in things as they themselves, and others, take things out.[56]

---

[53] ibid. (Magic Cauldron) pp. 150-153.

[54] In other words, open source development is a collective process designed for libertarians.

[55] Ghosh (1997). Takahashi (2000) gives an excellent overview of generalized exchange and some of the puzzles it poses for game-theoretic analysis.

[56] ibid.

But then he observed:

> The Internet cooking-pots … are quite different [from real cooking pots.]… They take in whatever is produced, and *give out their entire contents to whoever wants to consume. The digital cooking pot … dish[es] out not single morsels but clones of the entire pot. …*
>
> The key here is the value placed on *diversity*.[57] [Emphasis added.]

In effect, our model of the open source development process formalizes the link suggested by Ghosh, between a viable system of generalized exchange and a set of diverse non-rival goods. We show that a system based on implicit exchanges of diverse non-rival goods is both economically valuable and potentially self-enforcing, as long as the Prisoners' Dilemma of communication can be solved. In addition, we show that "diversity"—in our case, the diversity created by modules and options—is indeed necessary in order to draw volunteers into this system.

Josh Lerner and Jean Tirole's paper "The Simple Economics of Open Source," (2002a) was one of the first attempts to explain the open source development process in economic terms. Taking the perspective of labor economics, Lerner and Tirole focused especially on the apparent puzzle of open source developers working "for free." They argued that after a full accounting of costs and benefits, most programmers' decisions to supply open source code were fully consistent with a rational choice model. They also pointed out that, a good reputation obtained via visible open source contributions might be a valuable signal, hence an asset, in the labor market. The *real* puzzles of the open source development process, they suggested, have to do not with the motivations of developers, but with the "leadership, organization and goverance" of successful projects. We agree with this assessment, but would add "what constitutes a workable code architecture?" to their list of puzzles.

Turning to formal models, Justin Pappas Johnson (2002) constructed one of the earliest game-theoretic models of the open source development process. He was the first to connect open source software development with the so-called "private provision of public goods", and to focus on both free-riding and redundant effort in that context. His model served as the point of departure for our own model of involuntary altruism (discussed in sections 3 – 5 above), although we simplified his framework in some ways and expanded it in others. (Specifically, Johnson allows heterogeneity in the developers'

[57] ibid.

valuations and costs of effort, while we do not; we explicitly model both the modular structure and the option values of the codebase, while he does not.) An important difference in our approaches is that Johnson does not explicitly consider what we have called the "Robinson Crusoe alternative," that is, the developers' right to code in isolation. Thus the equilibria he derives have the property that all developers will do as well staying out of the game as if they join. As indicated above, we think this feature is problematic in a process that must rely solely on volunteers.[58]

Johnson and others writing in the public goods tradition assume that work products must be automatically contributed to the collective effort. In contrast, Harhoff, Henkel and von Hippel (2000) focus on the problem of "voluntary revelation" of a new design. To this end they construct a game-theoretic model in which two users must simultaneously decide whether to innovate, and if so, whether to reveal their innovation to an intermediary. The intermediary may improve the design, but will also reveal it to the other user, who may be a competitor. Hence this model combines the "code" and "publish" stages of a development process into one game. The fundamental tension of the model hinges on the tradeoff between the benefit due to the intermediary's improvement of the design, and the damage due to the competitor's use of the design. Equilibrium behavior depends on the balance of these factors. In contrast, we effectively ruled out competition among developers by assuming that the codebase is a non-rival good.

All the formal models discussed above, including our own, address the early stages of the open source development process in a game-theoretic way. In contrast, James Bessen (2002) has constructed a decision-theoretic model of the *late* stages of the process. Bessen asks the question, how can open source software compete with commercially developed software, given the well-known undersupply and free-ridership problems associated with public goods? He argues that the resolution of this apparent anomaly lies in the inherent complexity of software. More precisely, when a codebase embodies a large number of functionalities, there is a combinatorial explosion of "use products." A commercial firm cannot debug all the use products and remain profitable: it must perforce focus on the largest groups of users. At the same

[58] The omission of the Robinson Crusoe alternative accounts for the difference in Johnson's and our own appraisal of the impact of modularity on the development process. Johnson argues that "a modular structure for development need not enhance the level of contributions." However, under his assumptions, developers are not free to leave the collective process, even though it provides them individually with no more benefit that does coding alone. In contrast, we argue that, with a modular codebase, even *working* developers benefit by joining a collective development process, precisely because they gain access to different modules coded by other developers.

time, to protect their intellectual property, most firms do not reveal their source code, and thus users with non-standard needs cannot easily modify proprietary code in order to customize it. Such users might prefer to start with a codebase that they can modify at will, that is, a "free" or "open source" codebase.[59] Bessen goes on to show that open source and standardized software can coexist, because they occupy different "locations" in the overall software product market. However, he does not address the origins of open-source software, nor does he model the functioning of the open source development process itself. Open source codebases arise as a result of forces outside the model.

Also in Bessen's model, users' changes to the codebase are a one-way street: there is no provision in the model for users to provide feedback or to contribute modules or patches back to the original codebase, and thereby influence the evolution of the codebase over time. In contrast, Linus Torvalds, the designer of Linux and a premier architect of open source software, highlights the importance of user *feedback* to the architect and other developers of the codebase:

> There are a lot of advantages in a free system, the most obvious one being that it allows more developers to work on it and extend it. However, even more important that that is the fact that it in one fell swoop also gave me a lot of people who *used* it and thus both tested it for bugs and tested it for usability. …[A] single person (or even a group of persons sharing some technical goal) doesn't even think of all the uses a large user community would have for a general purpose system.
>
> So a large user-base has actually been a larger bonus than the developer base, although both are obviously needed to create the system that Linux is today. I simply had no *idea* what features people would want to have, and if I had continued to do Linux on my own it would have been a much less interesting and complete system.[60]

Consistent with Bessen's insight, Torvalds perceives the "use products" of a codebase to be far more diverse than the initial architect or developers can envisage. However, as Torvalds sees it, the codebase can and should evolve along the lines that users value most highly. In a sense, then, he is saying that the option values of particular features are revealed (or clarified) via user feedback as the development process goes on.[61] And clearly the architecture of codebase is one of the factors that makes such "user-driven" evolution possible.[62]

---

[59] The rights to view and to modify code are two of the "four freedoms" that are part of the basic definition of free or open source software. See http://www.gnu.org/philosophy/free-sw.html .

[60] Torvalds (1999).

[61] On the interplay of users' perceived needs and the evolution of complex designs generally, see Clark (1985).

[62] On the interaction of design architecture and design evolution, see Baldwin and Clark (2000).

Finally, a number of authors, both participants and observers, claim that the open source development process is a new—and perhaps better—way of organizing work. For example, Eric von Hippel and Georg von Krogh (2003) argue that the open source development process differs from two well-known production paradigms: classic "private goods," which will be produced voluntarily by private agents (usually firms) for sale in markets, and classic "public goods,"which are subject to free-riding, hence must rely for production on involuntary taxation or other forms of social coercion. In contrast, they argue, the open source development process produces a public good—the codebase—but at the same time is "self-rewarding," ie., developers choose what to code and thus get the code they most want. The result is "a promising new mode of organization that can indeed deliver the 'best of both worlds' to society under many conditions."[63] Indeed Siobhan O'Mahony (2003a, b) has investigated the structure of this "new mode of organization" in some detail. She has documented the emergence of key features like licenses, trademarks, brands, and non-profit foundations, and explained their purposes within the open source context.

In a similar vein, Benkler (2002) argues that "peer production", of which the open source development process is an example, differs from both traditional markets and traditional managerial hierarchies. He observes that all "collaborative production systems", that is, any non-Robinson-Crusoe-type economy, must solve an information problem: the question of what individual agents should do in order to be productive. Markets supply the relevant information via prices; firms do so by setting up a hierarchy of authority for the purpose of assigning tasks. However, both these devices are subject to transaction costs, which in turn cause production-relevant information to be lost and resource allocation to be "sticky" and sub-optimal. Benkler then argues that "commons-based peer production," that is, unpaid work by volunteers in service of a common goal, can in some instances overcome the disadvantages of both market-mediated and managerially organized production. He goes on to explore the circumstances in which "peer production" is likely to be the most efficient way of setting up a productive system.

In different ways, then, von Hippel and von Krogh, O'Mahony, and Benkler all consider the open source development process to be an example—indeed, the examplar—of a new method of organizing

[63] von Hippel and von Krogh (2003), p. 213.

work that is suitable for the production of certain kinds of goods. We will develop this idea and analyze both the boundaries and the competitive implications of this "new institutional form" in the final section of this paper.

8 Conclusion: Open Source Development as a Productive Institution

In his path-breaking work on comparative institutional analysis, Masahiko Aoki defined an *institution* in the following way:

> An institution is a self-sustaining system of shared beliefs about how the game is played. Its substance is a compressed representation of the salient, invariant features of an equilibrium path, perceived by almost all the agents in the domain as relevant to their own strategic choices. As such it governs the strategic interactions of agents in a self-enforcing manner and in turn is reproduced by their actual choices in a continually changing environment.[64]

In other words, an institution is a framework for social interaction that can come into existence and continue to exist based on mutually consistent beliefs, which are confirmed and reinforced by the actual behavior of individuals operating within the framework. The collective development process we have modeled is indeed an institution in this sense. It is also a way to organize human beings for the production of certain types of goods—basically, non-rivalrous goods. The next questions to arise are: First, under what external conditions is it possible for this type of "productive institution" to come into existence and survive? And second, what does the existence of this institutional form imply for other productive institutions in the economy, most especially, for-profit firms? These were the questions we asked at the beginning of this paper, which, on the basis of our theoretical analysis, we are now able to address.

8.1 Limits of the Institutional Form

The underpinnings of the collective development process as we have modeled it are twofold. First, users are developers. Second, the developers' work products are non-rivalrous goods. Absent these

[64] Aoki, 2001, p. 26.

characteristics, the institution we described cannot exist. Users who are not developers, cannot write code, and perforce cannot form a code-writing collective. And if work products are rivalrous goods, consumption by one user precludes consumption by another. In contrast, we have assumed throughout this paper that a developer can both donate her code to the collective, and continue to use it herself.[65]

The property of non-rivalry in use is a well-known and distinctive feature of non-physical objects, including ideas, designs, code, patterns, etc. In the telling words of Thomas Jefferson (as quoted by Lawrence Lessig):

> "[An idea's] peculiar character… is that no one possesses the less because every other possesses the whole of it. He who receives an idea from me, receives instruction himself without lessening mine; as he who lites his taper at mine, receives light without darkening me."[66]

In contrast, virtually all physical objects and physical spaces are subject to exclusion constraints, or at least congestion effects, that make them rivalrous in use.

In addition to having user-developers and non-rivalrous goods, a collective development process must be more appealing to developers than working alone. For this to occur, we have shown that the collective's work product must have a modular architecture or option value or both. A modular architecture and/or option value allows implicit "exchanges" of effort to take place among the working developers who participate in the collective process.

We must emphasize that *these exchanges are not classic transactions*. The objects being exchanged in the collective development process are non-rivalrous goods, thus in effect, each developer gets to have his cake and eat it too. Still the exchanges are welfare-enhancing because (1) different developers can target different modules; and (2) designs of the same module are uncertain, hence have option value. Thus, just as in the classic analysis of transactions in a free market, free exchanges of effort within modular, option-laden architecture are beneficial to all participants.

Beyond user-developers, non-rival goods, and a modular, option-laden architecture, the collective process we have described needs several other enabling features. These may be present in the initial environment or created by the participants or both. First, for its very existence, the collective

---

[65] In our formal analysis, the non-rivalrous nature of code was assumed to be absolute: code was a "pure" public good by Samuelson's (1954) definition. More generally, code might be subject to minor transactions costs and/or congestion effects, and most of the results would still go through. Nevertheless, non-rivalrous use lies at the very heart of the collective development process as we have modeled it.

[66] Jefferson to Isaac McPherson (August 13, 1813), quoted in Lessig (2001) p. 94.

development process requires *methods of communication that are scaled to the coding interval*. More modular architectures have shorter coding intervals (because the units of useful work are smaller), hence require faster and cheaper communication. Second, any decentralized process based on a modular architecture requires mechanisms for system integration and the testing of modules. In the models of this paper, we assumed that a modular codebase "assembled itself" automatically, but that was a gross idealization.[67] Real open source development projects require both a good code infrastructure, for example, CVS trees, and human agents (so-called committers and maintainers) in order to the manage system integration and testing in an orderly, efficient way.

Two other features, while not strictly necessary for the existence of a collective process, can make such processes much more efficient. In the first place, we have seen that developers can operate with perfect or imperfect knowledge about what others are doing. When a codebase has option value, multiple efforts targeted at the same module are not, strictly speaking, redundant. Nevertheless, for each module, there is an optimal number of trials to run, and with purely random targeting, it is possible to both undershoot and overshoot this optimum. Thus a collective process can be made more efficient by publicizing who is working on what. And, in point of fact, most real open source projects do make it easy for potential workers to find out which tasks are well-covered and which are going begging.

The second efficiency factor is related to system integration and testing. In a codebase with option value it is necessary to distinguish better designs from worse designs. Thus the developers need ways to communicate about their individual perceptions of quality, and ways to resolve disputes when they arise. In our models of the collective process, we made "finding the best" of several candidate designs appear automatic. In actuality, ranking candidate designs is both costly and potentially contentious. And in the context of the open source development process, unresolved conflicts are especially problematic, because dissatisfied parties can always choose to "fork" the codebase. Forking in turn creates two (or more) codebases where there only needs to be one, and thus undercuts the economies of the collective process.

Last but not least, to be credible, a collective development process must have ways to resolve the

[67] On the challenges and pitfalls of system integration and testing, see Brooks (1995); Zachary (1994); Torvalds (1999); Narduzzo and Rossi (2003); or almost any hardware or software engineering text. We are on record as having said, "Testing costs are the Achilles heel of modular design." (Baldwin and Clark, 2000, p. 272.)

Prisoners' Dilemma of revelation. Indeed, as we saw above, actual open source development processes seem to address this issue in several, complementary ways. First, thanks to the Internet, email and newsgroups, the cost of communication among developers is indeed very low. Second, those who choose to communicate receive recognition and status within their group, and may be rewarded in other ways as well. Finally, especially in the early stages of the building of a codebase, interactions among the developers are repeated and open-ended, and so "the shadow of the future" looms large.

8.2 Competitive Implications

What does the existence of a collective development process based on implicit exchanges of effort portend for other actors in an economic system? In particular, what does the presence of this institution signify for firms that might supply similar goods? In brief, a collective development process operating within a modular and option-laden architecture can be a strong competitor. It can diminish and potentially destroy the profitability of a commercial codebase.

To see how commercial codebases and firms fit into the picture, suppose we have a set of user-developers who value a particular type of codebase. However, contrary to our initial assumptions, suppose the value of the codebase to one developer is less than her cost of making it. In this case, no developer *acting alone* will have incentives to create the good in question. But if there are many developers, all of whom assign a value to the good, an entrepreneur may take advantage of the situation. He could hire as many developers as were needed to create the codebase, make it, and then sell it to all who wanted it. In fact, this is a classic justification for the existence of firms in an economic system: As institutions, they can aggregate demand and profitably supply products that are "too big" for any individual to construct.

However, a modular architecture for the good in question opens up a new and different pathway of production. Rather than aggregating demand, as firms do, the architecture can bring the scale of a useful contribution down to the level of one person's effort. Then, as we have seen, a collective development process fueled by the "exchange" of effort can be used to create a system that is potentially as good or better than the one a commercial firm might supply. It follows that developer-users who can "hook into" a collective process via a modular architecture do not need firms as production

intermediaries. In this fashion, a modular architecture can operate as a replacement for firms in the larger economic system.[68]

What does the existence of this new pathway of production imply for a firm that proposes to compete "in the same space" as the collective? In essence, the collective development effort directs a large amount of the producer's surplus into the hands of the user-developers. To make this point clear, suppose there are N potential user-developers, each of whom places a value V on a complete system. A for-profit firm aims to enter this market, but must stave off a potential competition from a collective development process. We assume the firm will hire its own developers (who are not users), create a functionally equivalent good, and sell that good to the N user-developers who value it. For simplicity, we assume that only one such firm will enter this market, hence its primary competition will come from the potential collective process, not from other firms. Finally we assume the firm will compete based on price.[69]

To be price-competitive with the collective process, *the firm must charge each user-developer less than her cost of effort if she worked on the collective product*. Under our assumptions, in a system of j modules that cost is c/j. If the firm charges more than this amount, a coalition of developers could band together and do better on their own. However, if the firm charges less than c/j, those same workers will be better off buying from the firm.

Under these assumptions, the *most revenue* the firm can hope to obtain is $Nc/j - \varepsilon$, where ε is a small number. This revenue cap arises because user-developers have the ability to replace the firm with a collective development process, and will have incentives to do so if the firm charges them more than their individual cost per module, c/j. Note that the firm's revenue depends inversely on the number of modules, and does not depend at all on the technical potential of the system. This means that any option value arising from the technical potential of the system will be captured by users, and not by the firm that produces the system.

---

[68] This argument follows the thinking of Lessig (1999): a technical protocol, in this case an architecture, becomes a substitute for system of social organization, in this case a firm.

[69] There are other strategies a firm might employ, but they involve co-opting a coalition of developers large enough to block the formation of a collective development process. Most such coalitions will be highly unstable.

Against this revenue must be set the cost of making a system that is equivalent to the product of the collective effort. Given the assumptions and results of previous sections, the cost of matching performance is either c, for a system with no option value, or k_j*c, for a system with option value. Thus the net present value (NPV) of the opportunity to a for-profit firm is:

$NPV = Nc/j - c - \Pi$ for a system with no option value; or

$NPV = Nc/j - k_j^*c - \Pi$ for a system with option value.

Code architecture enters into these NPVs in a powerful way. First, as indicated above, revenue goes down in inverse proportion to the number of modules. Second, in systems with option value, the cost of matching functionality goes up with the option value, because k_j^* is an increasing function of both Π and j (see Table 4). *Thus, ironically, systems with more option value are actually worth less as commercial opportunities*: a commercial firm must match the functionality of the option-laden system, but cannot charge any more for it. More modules mean more option value, hence more development effort, and the commercial firm must match that effort (with no reward) in order to have an equivalent system.

Indeed, in the presence of option value, it may be impossible for a commercial firm to compete with a collective process. Specifically, if the number of user-developers, N, is low relative to the number of modules and their option value, the firm's opportunity will have an *ex ante* negative NPV, and not be viable as a business proposition:

$$\text{If } N \leq k_j^* j, \text{ then } NPV \leq 0, \text{ and no commercial opportunity exists.} \quad (6)$$

In expression (6), the bound on N, $k_j^* j$, goes up faster than the number of modules. The bottom line is that it can be very difficult for a commercial firm to compete head-to-head with a collective development process that harnesses both modularity and option value in the architecture of its code.

Looking across different types of codebases, however, at any moment in time we would expect to see cross-sectional heterogeneity in their technical potential (Π), level of modularity (j), and number of user-developers (N). Then, expression (6) says, the two institutional forms, collective process and commercial firm, might "split up" the various codebase markets. In theory at least, codebases with many

users, few modules and low technical potential would be provided by firms, while those with fewer users, many modules, and high technical potential would become the targets of collective development. Thus in a large market with many groups of users and diverse codebases, both institutional forms may co-exist, serving different needs. But, especially when a new architecture arises, such peaceful co-existence may end, and the two institutional forms may wind up competing head-to-head in the same marketplace. Indeed, that is our reading of the situation today in the market for server operating systems.

References

Aoki, Masahiko (1999) "Information and Governance in the Silicon Valley Model," Stanford University, http://wwwecon.stanford.edu/faculty/workp/swp99028.html, viewed Jan 12, 2001.

Aoki , Masahiko (2001). *Towards a Comparative Institutional Analysis*, MIT Press, Cambridge, MA.

Aoki, Masahiko and Hirokazu Takizawa (2002) "Incentives and Option Value in the Silicon-Valley Tournament Game, *Journal of Comparative Economics*, forthcoming.

Aoki, Masahiko and Hirokazu Takizawa (2002) "Modularity: Its Relevance to Industrial Architecture," presented at the Saint-Gobain Centre for Economic Research 5th Conference, Paris, FR, November.

Axelrod, Robert F. (1984) *The Evolution of Cooperation*, Basic Books, NY.

Baker George, Robert Gibbons and Kevin J. Murphy (2002) "Relational Contracts and the Theory of the Firm," *The Quarterly Journal of Economics* 117:39-84.

Baldwin, Carliss Y. and Kim B. Clark (1992) "Modularity and Real Options: An Exploratory Analysis" Harvard Business School Working Paper #93-026, October.

Baldwin, Carliss Y. and Kim B. Clark (2000). *Design Rules, Volume 1, The Power of Modularity*, MIT Press, Cambridge MA.

Benkler, Yochai (2002) "Coase's Penguin, or Linux and the Nature of the Firm," *Yale Law Journal*, 112.

Bessen, James (2002) "Open Source Software: Free Provision of Complex Public Goods," http://www.researchoninnovation.org/opensrc.pdf viewed 5/22/03.

Brooks, Frederick P. (1995) The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition, Addison-Wesley, Reading, MA.

Chamberlin, John (1974), "Provision of Collective Goods as a Function of Group Size," *American Political Science Review*, 68(2):707-716.

Clark, Kim B. (1985) "The Interaction of Design Hierarchies and Market Concepts in Technological Evolution," *Research Policy*, 14(5):235-251.

Ghosh, Rishab Aiyer (1998) "Cooking Pot Markets: An Economic Model for the Free Trade of Goods and Services on the Internet," *First Monday*, 3(3) http://www.firstmonday.dk/issues/issue3_3/ghosh/index.html viewed 5/22/03.

Gintis, Herbert (2000) *Game Theory Evolving*, Princeton University Press, Princeton, NJ.

Harhoff, Dietmar, Joachim Henkel and Eric von Hippel (2000) "Profiting from Voluntary Information Spillovers: How Users Benefit by Freely Revealing Their Innovations," Sloan Working Paper, #4125, MIT, July.

Head, John G. (1962) "Public Goods and Public Policy," *Public Finance*, 17(3):197-219.

Henkel, Joachim and Eric von Hippel (2003) "Welfare Implications of User Innovation," draft, rec'd April.

Johnson, Justin Pappas (2002) "Open Source Software: Private Provision of a Public Good," *Journal of Economics and Management Strategy* 2(4):637-662.

Kuwabara, Ko (2000) "Linux: A Bazaar on the Edge of Chaos," *First Monday*, 5(3) http://www.firstmonday.dk/issues/issue5_3/kuwabara/index.html viewed 5/22/03

Lakhani, Karim and Eric von Hippel (2003) "How Open Source Software Works: 'Free' User-to-user Assistance," *Research Policy*, 32:923-943.

Lerner, Josh and Jean Tirole (2002a) "Some Simple Economics of Open Source," *Journal of Industrial Economics*, 52:197-234.

Lerner, Josh and Jean Tirole (2002b) "The Scope of Open Source Licensing," NBER Working Paper 9363, Cambridge, MA, December.

Lessig, Lawrence (1999) *Code and Other Laws of Cyberspace*, Basic Books, New York, NY.

Lessig, Lawrence (2001) The Future of Ideas: The Fate of the Commons in a Connected World, Random House, New York, NY.

Levy, Steven (1984, 2001) *Hackers: Heroes of the Computer Revolution*, Penguin Books, New York. NY.

Lindgren, Bernard W. (1968) *Statistical Theory*, Macmillan Publishing Co., New York, NY.

Loch, Christoph H., Christian Terwiesch and Stefan Thomke (2001) "Parallel and Sequential Testing of Design Alternatives," *Management Science*, 45(5):663-678.

Maynard Smith, John (1982). *Evolution and the Theory of Games*, Cambridge University Press, Cambridge, UK.

Merton, Robert C. (1973) "Theory of Rational Option Pricing," *Bell Journal of Economics and Management Science*, 4(Spring): 141-183; reprinted in *Continuous Time Finance*, Basil Blackwell, Oxford, UK, 1990.

Merton, R. C. and Zvi Bodie (1995). "A Conceptual Framework for Analyzing the Financial Environment," in The Global Financial System: A Functional Perspective, (ed. Crane et al) Harvard Business School Press, Boston, MA.

Milgrom, Paul and John Roberts (1990) "The Economics of Modern Manufacturing: Technology, Strategy and Organization," *American Economic Review*, 80:511-528.

Narduzzo, Alessandro and Alessandro Rossi (2003) "Modularity in Action: GNU/Linux and Free/Open Source Software Development Model Unleashed," draft, rec'd May.

Olson, Mancur (1971) *The Logic of Collective Action*, Harvard University Press, Cambridge, MA.

O'Mahony, Siobhan (2002) "The Emergence of a New Commercial Actor: Community Managed Software Projects," Ph.D dissertation, Department of Management Science and Engineering Management, Stanford University, June.

O'Mahony, Siobhan (2003a) "Guarding the Commons: How Community Managed Software Projects Protect Their Work," *Research Policy* 1615:1-20.

O'Mahony, Siobhan (2003b) "Non-Profit Foundations and their Role in Community-Firm Software Collaboration," *Making Sense of the Bazaar: Perspectives on Open Source and Free Software*, O'Reilly & Associates, Inc., Sebastopol, CA.

Palfrey, Thomas R. and Howard Rosenthal (1984) "Participation and the Provision of Discrete Public Goods: A Strategic Analysis," *Journal of Public Economics*, 24:171-193.

Raymond, Eric S. (1999) *The Cathedral and the Bazaar* O'Reilly & Associates, Inc., Sebastopol, CA.

Salus, Peter H. (1994) *A Quarter Century of Unix*, Addison-Wesley Publishing Company, Reading, MA.

Samuelson, Paul A. (1954) "The Pure Theory of Public Expenditure," *Review of Economics and Statistics*, 26:387-390.

Sanchez, Ron (1991) "Strategic Flexibility, Real Options and Product-based Strategy," Ph.D dissertation, Massachusetts Institute of Technology, Cambridge, MA.

Shah, Sonali (2003a) "Identifying and Distinguishing between Motives: A Framework for Studying the Choices Made by Software Developers When Allocating their Discretionary Time," draft, rec'd April.

Shah, Sonali (2003b) "Understanding the Nature of Participation and Coordination in Open and Gated Source Software Development Communities," draft, rec'd April.

Shapiro, Carl and Hal R. Varian (1999) *Information Rules: A Strategic Guide to the Network Economy*, Harvard Business School Press, Boston, MA.

Simon, Herbert A. (1962) "The Architecture of Complexity," *Proceedings of the American Philosophical Society* 106: 467-482, repinted in *idem*, (1999) *The Sciences of the Artificial*, *3rd ed.* MIT Press, Cambridge, MA, 183-216.

Simon, Herbert A. and Mie Augier (2002) "Commentary on 'The Architecture of Complexity'," in *Managing in the Modular Age: Architectures, Networks, and Organizations*, (R. Garud, A. Kumaraswamy, and R. N. Langlois, eds.) Blackwell, Oxford/Malden, MA.

Stallman, Richard M. (1983) "Initial Announcement of the GNU Project," http://www.gnu.org/gnu/initial-announcement.html viewed 5/30/03.

Stallman, Richard M. (1999) "The GNU Operating System and the Free Software Movement," in *Open Sources: Voices of the Open Source Revolution*, (C. DiBona, S. Ockman, and M. Stone, eds.) O'Reilly & Associates, Inc., Sebastopol, CA.

Sullivan, Kevin J., William G. Griswold, Yuanfang Cai and Ben Hallen, "The Structure and Value of Modularity in Software Design," University of Virginia Department of Computer Science Technical Report CS-2001-13, submitted for publication to ESEC/FSE 2001.

Takahashi, Nobuyuki (2000) "The Emergence of Generalized Exchange," *American Journal of Sociology*, 105(4):1105-1134.

Tanenbaum, Andrew S., Linus Torvalds et.al. (1999) "The Tanenbaum-Torvalds Debate," Appendix A in *Open Sources: Voices of the Open Source Revolution*, (C. DiBona, S. Ockman, and M. Stone, eds.) O'Reilly & Associates, Inc., Sebastopol, CA.

Thomke, Stefan (1998) "Managing Experimentation in the Design of New Products," *Management Science*, xxx:743-762.

Thomke, Stefan and David Bell (2001) "Sequential Testing in Product Development," *Management Science*, 47(2):xxx-yyy.

Topkis, Donald M. (1998) *Supermodularity and Complementarity*, Princeton University Press, Princeton, NJ.

Torvalds, Linus (1998) "What Motivates Software Developers?" Interview in *First Monday*, 3(3) http://www.firstmonday.dk/issues/issue3_3/torvalds/index.html viewed 5/22/03.

Torvalds, Linus (1999) "The Linux Edge," in *Open Sources: Voices of the Open Source Revolution*, (C. DiBona, S. Ockman, and M. Stone, eds.) O'Reilly & Associates, Inc., Sebastopol, CA.

Ulrich, Karl (1995) "The Role of Product Architecture in the Manufacturing Firm," *Research Policy*, 24:419-440, reprinted in *Managing in the Modular Age: Architectures, Networks, and Organizations*, (G. Raghu, A. Kumaraswamy, and R.N. Langlois, eds.) Blackwell, Oxford/Malden, MA.

Woodard, C. Jason (2001) "Approaches to Modeling Public Communication in Open Source Software Communities," draft, rec'd May.

Zachary, G. Pascal (1994) Showstopper: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft, Free Press, New York, NY.