The Architecture of Cooperation:  
Does Code Architecture Mitigate Free Riding in the Open Source Development Model?

Carliss Y. Baldwin
Kim B. Clark

Harvard Business School

This Draft:
August 30, 2004

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
The Architecture of Cooperation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model?

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. It can be addressed through a combination of reducing the costs of communication, providing rewards, and encouraging repeated interactions.

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

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.\(^1\) If 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.

In this paper we argue that the architecture of a codebase is a critical factor that lies at the heart of the open source development process.\(^2\) 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.

The rest of the paper is organized as follows. In section 2, we discuss code architecture and define modularity and option value. In sections 3, 4 and 5, we model critical aspects of the open source development process as a simple game and

\(^1\) 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 (Raymond, 1997, p. 84). Indeed, the belief that open source code is better than closed source code is common among participants in open source projects. (See numerous quotes in O’Mahony, 2002) However, the opposite view—that proprietary, closed source code is superior—is common, too. The fact that software developers disagree on this issue makes an investigation of the interaction of architecture and development process especially interesting.

\(^2\) This is a shortened and simplified but otherwise unrevised version of a working paper published in June 2003. Some sections of the working paper have been moved to appendices in a separate document.
investigate how changes in code architecture affect developers’ incentives to work in the context of this game. We will show that more 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, opportunities that do not exist in so-called “monolithic” codebases. 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 theories and models of the open source development process. Section 8 concludes.

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. Code architecture is not a matter of natural law: to a large degree, it is under the control of the initial designer or “architect” of the codebase. 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 tell how modular and how “option laden” the future, evolving codebase will be. Those judgments in turn affect the developers’ incentives to create new code and contribute it back to a collective development effort.

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.\(^3\) However, 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 modules in the system.\(^4\)

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.

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.

**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. Uncertainty in turn causes new designs to have “option-like” properties. According to modern finance theory, an option is “the right but not the obligation” to

---

\(^3\) This definition is taken from Rumelhart and McClelland (1995), as quoted in Baldwin and Clark (2000, p. 63). Although Herbert Simon did not use the term “modularity,” it is essentially what he called “near-decomposability.” (Simon, 1962; Simon and Augier, 2002).

\(^4\) Baldwin and Clark (2000).
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. First, when payoffs take the form of options, taking more risk creates more value.\(^5\) Second, seemingly redundant efforts may be value-increasing.\(^6\) 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. 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.\(^7\)

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

---

\(^5\) This is a basic property of options, proved for any distribution of payoffs in Merton (1973).

\(^6\) 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.

\(^7\) This is the definition of economic complementarity used by Milgrom and Roberts (1990). The complementarity of modularity and experimentation was rigorously demonstrated by Baldwin and Clark (1992; 2000).
write code. But first we need to define the developers’ work products and relationship to one another. We do so in the context of a simple game.

3 Games of “Involuntary Altruism”

Software code is a “non-rival” good in the sense that the use of the code by one developer does not prevent its use by another or many others. In this section, we model a work environment in which non-rival goods are produced. The environment 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. The result is a well-known game form, which in economics is sometimes labelled “the private provision of public goods.” Johnson (2002) applied this game form to the open source development process.8

In the interest of clarity, we will begin by laying out a symmetric, two-person, one-shot game with no uncertainty as to payoffs. We will then discuss variations of the basic scenario, including multi-person games, games with payoff uncertainty, and a game of “voluntary revelation” in which developers must decide to publish their code and pay a price for doing so.

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

---

8 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.” See, for example, Samuelson (1954) and Olsen (1971). Lessig (2001, p. 99-97) explains how these concepts apply to ideas, designs, software and the Internet.
> c, thus either party has an incentive to write code if the other doesn’t. 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, 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. 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. 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 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.

---

9 These values and costs can be interpreted as dollar equivalents of private benefits and costs.

10 Modularity implies divisibility of effort, however, a codebase can be divisible in terms of effort, but still not modular. See, for example, Gintis (2000, p. xx).
Figure 1
Normal Form of a Simple Game of Involuntary Altruism

There is also a mixed-strategy equilibrium of this game wherein each developer works with probability $\alpha^*$. We can find $\alpha^*$ by setting the expected payoff of working and not working equal to one another and solving the resulting expression:

$$\alpha^* = \frac{v - c}{v}.$$  

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.\(^\text{11}\)

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. For a given number of developers, there is always one mixed-strategy equilibrium. Equilibrium $\alpha^*$ decreases as the number of developers increases, but the probability that a given developer encounters at least one worker is always $(v-c)/v$; and the expected payoff per developer is always $v-c$.

---

\(^\text{11}\) 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. An evolutionarily stable strategy (ESS) is a distribution of strategy-types that cannot be “invaded.” (Maynard Smith, 1982.) For example, imagine that the two developers in our game are drawn randomly from a population consisting of $\alpha$ Workers and $1-\alpha$ Free-riders. If $\alpha = 1$, 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 $\alpha = 0$, 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. However, if the fraction of Workers and Free-riders equals the mixed-strategy equilibrium fraction $\alpha^*$ derived above, then a new “mutant” Worker or Free-rider will not have a higher expected payoff than the average member of the population.
3.2 The Attractiveness of the Game to Outside Developers. For an individual developer, the game we have specified is an alternative to coding in isolation. How attractive is it relative to this alternative? Table 1 compares 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.

Table 1
Comparison of Equilibrium Payoffs in the Involuntary Altruism Game

<table>
<thead>
<tr>
<th></th>
<th>Per Person Expected Payoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robinson Crusoe Environment</td>
<td>v–c</td>
</tr>
<tr>
<td>Pure-Strategy Equilibria</td>
<td></td>
</tr>
<tr>
<td>(one works, the others do not)</td>
<td></td>
</tr>
<tr>
<td>Free-rider</td>
<td>v</td>
</tr>
<tr>
<td>Worker</td>
<td>v–c</td>
</tr>
<tr>
<td>Mixed-Strategy,</td>
<td></td>
</tr>
<tr>
<td>Evolutionarily Stable</td>
<td>v–c</td>
</tr>
<tr>
<td>Equilibrium</td>
<td></td>
</tr>
</tbody>
</table>

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 in a way that increases 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. 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.

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. Now assume that the codebase consists of a minimal system plus j modules. At the time of the game, we assume that the minimal system has been coded, hence its costs are sunk.

Given the architectural design rules, each module can be worked on separately. 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. We also 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.12

12 In the presence of modularity, mixed modes of work and communication lags make the developers’ strategies potentially very complex. A simple game form does not then suffice to capture the situation.
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. The game now has two Nash equilibria as indicated by the shaded, off-diagonal cells. In each equilibrium, 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.\(^\text{13}\)

**Figure 2**

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

<table>
<thead>
<tr>
<th>Developer 1: Don't Work</th>
<th>Developer 2: Don't Work</th>
<th>Work on A</th>
<th>Work on B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 0</td>
<td>.5(v), .5((v-c))</td>
<td>.5(v), .5((v-c))</td>
<td></td>
</tr>
<tr>
<td>.5((v-c)), .5(v)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.5((v-c)), .5(v)</td>
<td></td>
<td>.5((v-c)), .5(v)</td>
<td>(v-.5c), (v-.5c)</td>
</tr>
<tr>
<td>.5((v-c)), .5(v)</td>
<td>.5(v-.5c), .5(v-.5c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.5((v-c)), .5(v)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. Conversely, if the number of modules is greater than the number of developers, \(j > N\), the building out of the system will

\(^{13}\) In the new equilibria, 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. If specific modules do not “pay their own way,” they will not be built, even if they were “envisioned” by the original architect.
proceed in multiple rounds. Let m be the integer part of the ratio j/N. For the first m rounds of work, all developers will work; in the last round, j–mN developers will work, and the rest will free-ride.

Table 2 shows that as long as the system has two or more modules, equilibrium payoffs to workers in the collective process are higher than payoffs to Robinson Crusoes. Thus participating in a collective development effort within a modular architecture beats coding in isolation. Intuitively, as long as someone else does some of the work, a developer is better off joining the collective than coding alone.

<table>
<thead>
<tr>
<th>Free-rider</th>
<th>j &gt; N</th>
<th>j ≤ N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker</td>
<td>v – mc/j</td>
<td>v</td>
</tr>
<tr>
<td>Robinson Crusoe</td>
<td>v – (m+1)c/j</td>
<td>v – c/j</td>
</tr>
<tr>
<td></td>
<td>v – c</td>
<td>v – c</td>
</tr>
</tbody>
</table>

Clearly these incentives work only if the target is a non-rival good. The payoff structure depends critically on the assumption that one developer’s use of a module does not interfere with another’s. Interestingly, however, 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 supports the allocation of individual effort to different modules.

Although these results are suggestive, the situation we have discussed so far involves a relatively high level of common knowledge in the game. Strictly speaking, to obtain the payoffs shown, each developer must know which modules the others will
work on, with certainty, before choosing his or her action. This assumption is, however, unnecessarily strict. In Appendix A, we show that the basic results go through in a system of imperfect information where each developer does not know what others are doing and thus risks duplicating their efforts.

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 never certain. In this section, we explore the effects of uncertainty and option value on the players’ behavior and resulting equilibria.

We begin by introducing some new notation. We assume that the value of a system can be modeled as a random variable, X. We also assume that developers are risk-neutral expected-value maximizers, and that coding intervals are short enough so that we can ignore their time preferences.

We have said that, when outcomes are uncertain, “duplication of effort,” in the sense of 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.\(^{15}\) Formally:

\[ Q(X ; k) = \text{the expected value of the highest of k draws from } X, \text{ if } Q(X ; k) > 0. \]

\(^{14}\) Formally for \( j \leq N \), this is obvious. For \( j > N \), it follows from the fact that \((m+1)/j\) is strictly less than one.

\(^{15}\) 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.
Q(X; k) = Emax (X^1, … X^k, 0) ;
where X^1, … X^k are the realizations of the individual trials. Let ΔQ(X; k) denote the difference between Q(X; k) and Q(X; k−1). For any distribution, Q(X; k) is increasing and concave; hence ΔQ(X; k) is positive and decreasing in k. 16

5.1 A Non-Modular Architecture with Two Developers. We are now ready to look at the effect of option value on a game with two developers and one module. 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: Q(X;1) = v. 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, Q(X;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 payoff to each worker if both work is: Q(X;1) + ΔQ(X;2) – c. The second term in the expression is the option value created by the second development effort. It is strictly positive.

The nature of equilibrium in this new game depends on the magnitude of the option value in relation to the developer’s cost of effort, c. If ΔQ(X;2) > c, <Work, Work> will be the unique Nash equilibrium of the game. There is then 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. The case is a little more complicated if ΔQ(X;2) ≤ c,

16 Aoki and Takizawa (2002).
but the results are basically the same. 17

Thus 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 the code architecture is not modular and there is some amount of free-riding in the collective process. With option value, parallel work is not necessarily redundant. An isolated developer loses out on the possibility that another worker’s codebase might be superior to her own. This possibility in turn tips the balance of payoffs in favor of joining a collective process.

5.2 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 intuition is correct.

In fact, in the context of our model, two things happen. First, Robert Merton (1973) showed that for any distribution, “a portfolio of options is always worth more

\[ \beta^* = \frac{v-c}{v-\Delta Q(X;2)} \]

A developer coding in isolation obtains expected payoff \( v-c \), which 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 she works with probability one. However, in the mixed-strategy equilibrium, each developer works with probability \( \beta^* \), and the expected payoff to workers and free-riders alike is:

\[ \text{Payoff to Workers and Free-riders} = \frac{v(v-c)}{v-\Delta Q(X;2)} \]

17 There will be two pure-strategy Nash equilibria in which one developer works and the other does not and a unique mixed-strategy and evolutionarily stable equilibrium in which each developer works with probability \( \beta^* \). Solving for \( \beta^* \) and substituting \( v \) for \( Q(X;1) \) yields:

\[ \beta^* = \frac{v-c}{v-\Delta Q(X;2)} \]

\( \Delta Q(X;2) > 0 \), hence this payoff is higher than the expected payoff to Robinson Crusoe developers.
than an option on a portfolio.” Thus, holding the distribution of the sum of module outcomes fixed, creating more modules increases option value, hence the value of the whole. But in the context of our game, there is another effect: creating more module with option value also increases the number of developers who will work in equilibrium. The latter effect is most easily seen if we assume that, in a system of j modules, module-option payoffs are symmetric and normally distributed with mean zero and variance, $\sigma^2/j$: $X_i \sim N(0, \sigma^2/j)$. The value of k design experiments targeted at a single module then equals $(\sigma^2j^{1/2}) Q(k)$, where $Q(k)$ denotes the expected value of the highest of k independent draws from a “standard” normal distribution with mean zero and variance one. The value of the whole system in this case is simply $j$ times this expression: $\sigma^2j^{1/2} Q(k)$. Note that this expression is increasing in $\sigma$, $j$ and $k$.

Now, in the context of our game, 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, we assume that developers enter the system sequentially that each new 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. The payoffs to the next index developer, conditioned on her actions, are:

---

19 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.
Free-ride: \( \sigma^{1/2} Q(k) \);  
Work: \( \sigma^{1/2} Q(k) + \sigma^{-1/2} \Delta Q(k+1) - c/j \).

In other words, 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, \( \sigma^{-1/2} \Delta Q(k+1) \), minus the cost of her effort, \( c/j \). She should then work if and only if:

\[
c \leq \sigma^{1/2} \Delta Q(k+1)
\]

The righthand side of this expression is increasing in both \( \sigma \) and \( j \), although decreasing in \( k \). Thus having more modules makes it easier to satisfy the inequality, that is, increases the developer’s incentives to work. From this it follows that more modular systems (higher \( j \)) generally support more parallel development efforts (higher \( k \)), other things equal.

Table 3 computes the number of developers who will work in equilibrium for different degrees of option value and modularity. Not surprisingly, for low levels of option value and modularity, the number of working developers tends to be small. In many case, it is zero: for example, in a non-modular system (\( j=1 \)), the option value per trial must be greater than 100% to justify any effort at all. However, holding the option value per trial fixed, the number of working developers tends to increase as the system is split into more modules. With an option value per trial of 100%, a system of ten modules will have 40 developers working in equilibrium (four per module); a system of 25 modules will attract 175 developers (seven per module). Essentially, the dispersion of module-outcomes combined with the ability (inherent in modular systems) to mix-and-match “the best with the best,” increases every developer’s incentives to work on a module. Given a fixed pool of developers, this in turn reduces the fraction of the pool
who will free-ride. Thus, within the confines of our model, we can say unequivocally that code architecture, specifically an option rich and modular architecture does mitigate free-riding in a voluntary, collective process with rational participants that is modeled after open source development processes.

Table 4
The Number of Developers Working in Equilibrium as a Function of Option Value per Trial, Q(1)/c, and the Number of Modules, j

<table>
<thead>
<tr>
<th>No. of Modules</th>
<th>Option Value of One Trial/ Cost of One Trial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25%</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

At the same time, such architectures increase the advantages to outside developers that come from working within a collective development process vs. coding in isolation. Indeed, table 4 shows that projects that are not worth undertaking by a single developer can in theory attract hundreds of developers if the architecture is sufficiently modular.
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 code 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 was 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.

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. 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.

Each developer must now 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 3. As long as r < v, this is a one-shot Prisoners’ Dilemma (PD) game, whose unique Nash equilibrium is <Don’t Reveal, Don’t Reveal>. Moreover, the equilibrium of this game is not affected by changes in the number of modules nor by increased option value.
Figure 3
Normal Form of a Game of Voluntary Disclosure: Two Modules and Two Developers

Thus regardless of 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 one of them below and discuss several others in Appendix B. As it turns out, although the basic Prisoners’ Dilemma is not affected by code architecture, the remedies are generally cheaper or easier to implement if the codebase is modular, has option value, or both.

As Robert Axelrod (1984) has shown, “fixing” a Prisoners’ Dilemma game involves adding benefits or reducing the costs of cooperation to the point where the equilibrium shifts to one of mutual cooperation. Thus if the development process can provide benefits to revealing that offset the costs, the cooperative outcome <Reveal, Reveal> becomes the equilibrium.

For example, 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. Then, 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. Earlier we showed that the value of an individual module to its creator would elicit coding effort. And the ability to exchange modules would bring developers into the collective process. 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 communication, 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 licenses protect the group, not against free-riding (which is tolerated), but against the conversion of the codebase into someone else’s intellectual property.20

On the benefits side, many observers of the open source movement contend that public recognition of authorship holds the key to the open source development process. Those developers who make the effort to communicate benefit from public recognition.

---

20 By his own account, the desire to protect developers’ 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. Lerner and Tirole (2002b) found that the GPL and its variant, the “Lesser GPL”, applied to 70 – 80% of the codebases they surveyed.
in three ways.\textsuperscript{21} First, their reputation may be enhanced leading to higher pay in the labor market. Second, they may enjoy higher status and be recognized by members of their own group or “tribe”. Finally, contributors may obtain affirmation of the value of their work and the meaningfulness of their effort. These three potential benefits are not mutually exclusive: any or all may matter to an individual developer. A number of empirical researchers are presently trying to disentangle the motives of open source developers. 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 reduces the effort associated with a minimal contribution. Emails and postings can be shorter if the code is modular.

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 many developers can compete to provide the best design.\textsuperscript{22} Indeed, an option rich, modular architecture creates many such tournaments, involving different areas of the

\textsuperscript{21} For discussions of these benefits, see Raymond (1999), Lerner and Tirole (2002a), and Shah (2003a,b).
codebase 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 the would-be competitors with ready-made opponents, judges, and an audience. As a result, for some, “stardom” may be more easily achieved within an active community than by working alone.23

7 Other Theories and Models of the Open Source Development Process

In this section, we relate our theory and models to others that have been put forward to explain the open source development process. The literature on this topic is vast, thus we will focus on works that are representative and that influenced our own models.

The most detailed and comprehensive theory of the open source development process was put forward by Eric Raymond in three essays first published on the worldwide web.24 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 or value the code itself.25 Second, he conjectures that “given enough eyballs, all bugs are shallow.” This is a shorthand phrase that captures the idea that an open, massively parallel, fast-cycle process of coding, releasing, peer review, and debugging, can in some cases outperform

22 Aoki (2001); Aoki and Takizawa (2002a, b).
23 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.)
25 Raymond (1999) (CatB) p. 32.
proprietary, closed source coding practices.\textsuperscript{26} Third, 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.”\textsuperscript{27} Finally, 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.\textsuperscript{28}

Our linked-games-in-an-architecture model formalizes and amplifies parts of Raymond’s theory, using the economic tools of 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 show that free-riding can be reduced, and the underprovision problem eliminated (at least theoretically) by an option rich modular design architecture. The more modular and option-laden the architecture is, the more attractive and valuable the collective process will be to developers. 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. 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 influenced by an idea advanced by Rishab Ghosh (1997):

\textsuperscript{26} ibid. (CatB) pp. 36-44.
\textsuperscript{27} ibid. (Homesteading) pp. 99-118.
\textsuperscript{28} Quoting Raymond: “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, write a ChangeLog entry, and sign the FSF assignment papers.’” ibid. (Magic Cauldron) pp. 150-153.
the so-called “cooking-pot model,” of generalized exchange. 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.

Josh Lerner and Jean Tirole (2002a) argued that a good reputation obtained via open source contributions can be an asset in the labor market, and as a result, developers’ contributions to open source projects may be fully consistent with economic models of rational choice. The real puzzles of the open source development process, they suggested, have to do not with motivation, but with the “leadership, organization and goverance” of successful projects. We would add “what constitutes an appropriate code architecture?” to their list of puzzles.

Justin Pappas Johnson (2002) was the first to model open source software development using the “private provision of public goods” framework. His model served as the point of departure for our own Base Case, although we simplified his framework in some ways and expanded it in others. Johnson, however, does not explicitly consider what we have called the “Robinson Crusoe alternative.” As a result, the equilibria he derives all have the property that developers will do as well by staying out of the game as by joining. This seems problematic in a setting without any form of coercion, in which the process must rely on strictly voluntary contributions.

---

29 Specifically, Johnson allows heterogeneity in the developers’ 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.

30 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
Johnson and others writing in the public goods tradition assume that work products are automatically revealed. In contrast, Harhoff, Henkel and von Hippel (2000) focus on the problem of “free revealing” of a new design. 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 fundamental tension of their 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. 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 contrast, James Bessen (2002) has constructed a 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 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. Bessen goes on to show that open source and standardized software can coexist, because they occupy different “locations” in the overall software product market.

Finally, a number of authors 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 both classic “private goods,” which are produced voluntarily by private agents for sale in markets, and classic “public goods,” which must rely for production on taxation or other forms of coercion. In contrast, they argue, the open source development process produces a public good—the codebase—that is “self-rewarding,” i.e., 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.”

Siobhan O’Mahony (2003a, b) has investigated the structure of this “new mode of organization” in some detail. 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, and goes on to explore the circumstances in which it 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 the examplar of a new method of organizing work that is suitable for the production of certain kinds of goods. Thus one contribution of this paper is to address the question, in a world of rational developers, what does the open source development process need to work? We summarize our findings in the concluding section of this paper. (Appendix C discusses what the open source development process may portend for commercial firms with proprietary codebases that are in head-to-head competition with open source codebases.)

---

8 Conclusion: What Does the Open Source Development Process Need to Work?

The underpinnings of the open source development process as we have modeled it are twofold. First, users are developers. Second, the developers’ work products are non-rivalrous goods. Absent these characteristics, the institution, as we have modeled it, 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.

In addition to having user-developers and non-rivalrous goods, we stipulated that a collective development process must be more appealing to developers than working alone. We showed that, for this to occur, the work product must have a modular architecture, option value, or, better still, 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. However, these exchanges are not classic transactions. The exchanges are welfare-enhancing, not because transactors have different marginal rates of substitution for the same good (as in classic economic theory), but because participants get to use the goods of others as well as their own. In effect, each developer gets to have his cake and eat it too.

Beyond user-developers, non-rival goods, and an option rich modular design

---

32 We are here using Masahiko Aoki’s (2001) definition of an institution: a stable form of social interaction supported by self-confirming beliefs.

33 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.
architecture, the collective process we have described needs several other enabling features. First, it needs methods of communication that are scaled to the coding interval. Modular architectures have shorter coding intervals (because the units of useful work are smaller), hence require faster and cheaper communication. Second, the process requires mechanisms for system integration and the testing of modules. In this paper, we assumed that a modular codebase “assembled itself” automatically, but that is a gross oversimplification. Real open source development projects require both good code infrastructure tools, for example, CVS trees, and human agents (committers and maintainers) to manage system integration and testing in an orderly, efficient way.

Two other features, while not strictly necessary, can make collective processes much more efficient. In the first place, developers need to know who is working on what. 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. Secondly, 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 over quality when they arise. In our models, we made “finding the best” of several candidate designs appear automatic. In reality, efficient methods of ranking need to be part of the institutional structure.

Last but not least, we saw that a collective development process must have ways to resolve the Prisoners’ Dilemma of revelation. Actual open source development processes appear to address this issue in several ways. First, thanks to the Internet,

---

34 On the challenges and pitfalls of system integration and testing, see Brooks (1995); Torvalds (1999); Narduzzo and Rossi (2003); or almost any hardware or software engineering text.
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. Finally, as discussed in Appendix B, 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.

References


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


