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

Carliss Y. Baldwin*
Kim B. Clark†

Harvard Business School

Final Version:
October 3, 2005

*Harvard Business School
†Brigham Young University, Idaho

Our thanks to Jason Woodard, John Rusnak, Alan MacCormack, Eric Raymond, Ben Hyde, Paul David, Jim Bessen, Joachim Henkel, Masahiko Aoki, Nobuo Ikeda, Eric von Hippel, Daniel Frye, Karim Lakhani, Siobhan O'Mahony, Sonali Shah, for comments that enhanced this paper in many ways. We especially thank two anonymous reviewers and the Editors of this volume for extremely careful reviews and thoughtful suggestions that allowed us to increase the rigor and scope of the analysis. Finally we thank participants in the HBS-MIT Conference on Free/Open Source Software and the LMU-MIT International Workshop on User Innovation and Open Source Software, and members of the Negotiations, Organizations and Markets group at Harvard Business School for generously sharing their insights on many occasions. 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 Participation: 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. We define two observable properties of an architecture: (1) modularity and (2) option value. Developers can often make informed judgments about modularity and option value from early, partially implemented code releases. We show that codebases that are more modular or have more option value (1) increase developers' incentives to join and to remain involved in an open source development effort; 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 codebases that are not modular or have no option value.

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 from code developed using proprietary development processes.[1] This paper explores the interactions between the open source development process and the design and structure of codebases from a theoretical perspective. We 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 define two observable properties of an architecture: (1) modularity and (2) option value. Developers can often make informed judgments about modularity and option value from early, partially implemented code releases. We will show that codebases that are more modular or have more option value (1) increase developers' incentives to join and to remain involved in an open source development effort; 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 codebases that are not modular or have no option value.

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 investigate how changes in architecture affect developers' incentives to work in the context of this game. Section 6 considers developers' incentives to voluntarily reveal code they have already created. Section 7 connects our work to other theories 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, 1999, 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

models of the open source development process. Section 8 concludes.

2 Code Architecture: What It Is and Why It Matters

According to Shaw and Garlan (1996, p.1), software architecture specifies (1) the elements of a codebase; (2) how they interact; (3) patterns of composition; and (4) constraints on those patterns. The architecture of a codebase is not a matter of natural law, but is to a large degree under the control of the initial designers of the system. Evidence of this fact may be found in the many texts that teach software engineers how to design good architectures,[2] and in the fact that codebases performing the same tasks can have dramatically different architectures.[3]

In this paper, we will show that a code architecture can be well-suited or ill-suited to the open source development process. We begin by introducing two important properties of an architecture: (1) its *modular structure*; and (2) the *option values embedded in the modules*.

2.1 Modularity. A complex system is said to exhibit *modularity in design* if its parts can be designed independently, but will work together to support the whole.[4] Systems can have different modular structures and different degrees of interdependence between their respective components. However, the different parts of a modular system must be compatible. Compatibility is ensured by *architectural design rules* that developers obey and can expect others to obey.

It is useful to divide a codebase and its architecture into (1) a platform; and (2) a set of modules. The *platform* supports the modules and is essential to the system.

---

makes an investigation of the interaction of architecture and development process especially interesting.

[2] See, for example, Shaw and Garlan (1996) and Parnas (2001).

[3] See, for example, Sullivan *et. al.* (2001) and MacCormack *et. al.* (in this volume).

[4] The modular structure of the *design* of a codebase is not necessarily the same as the "runtime" modularity of the compiled code (Baldwin and Clark, 2000). In this analysis, we are interested in what the developers see when they

Modules are distinct parts of the larger system, which can be designed and implemented independently as long as they obey the design rules. For example, in the Linux operating system, the kernel (*kernel*) and file sharing (*fs*) subdirectories are part of the platform, while the device drivers (*drivers*) and architecture (*arch*) subdirectories contain most of the modules.[5] In the Eclipse open source project, the application programming interface (APIs) and related code constitute the codebase's platform, while the plug-ins constitute the modules.

We emphasize that, for our purposes, "platform" and "module" are simply definitions that allow us to characterize different structural parts of a codebase. We are not excluding any architecture from consideration. In the extreme, for example, a "monolithic" codebase can be thought of as a platform and module integrated into a single unit.

2.2 Option Value. Software development is a design process. A fundamental property of designs is that at the start of the 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 choose a course of action and obtain an associated payoff. A new design creates the ability but not the necessity—the right but not the obligation—to do something in a new way. *But the new design does not have to be adopted:* indeed (as long as the designers are rational) it will be adopted only if it is better than the old. In this sense a new design is an option.

Options interact with modularity in a powerful way. By definition, a modular

---

write code, hence our focus is on modularity in design.
[5] Godfrey and Tu (2000); Rusnak (2005).

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.[6]

2.3 Discerning Modules and Options. The modular structure and options of a codebase are observable. *Ex post* modular structure can be detected by mapping the dependencies of a codebase using a variety of tools.[7] With regard to options, each time a new function is added (for example a new device driver in Linux) or an existing function upgraded, a design option has been exercised. Thus the change logs of a codebase track the exercise of options.

We are interested, however, in developers' *ex ante* perceptions of modules and options, because these perceptions (we will show) influence their decisions to code. On the one hand, if the architecture of a codebase has been formally specified in terms of design rules, then it is straightforward to determine its modular structure. Options are attached to modules, and thus if one knows the modular structure, it is possible to identify the options and even formally estimate the option values inherent in the codebase.[8]

On the other hand, many open source projects do not formally specify their architectural design rules. Notwithstanding this fact, developers can still discern the modular structure of a to-be-written codebase and make informed judgments about the location and magnitude of the options. Their appraisals may be based on commitments

[6] Baldwin and Clark (2000, Chapter 3).
[7] One useful tool is the so-called "Design Structure Matrix" mapping method. See, for example, MacCormack *et.al.* (in this volume) and the citations therein.
[8] Sullivan *et. al.* (2001) and Lopes and Bajracharya (2005).

by the initial architect(s), for example, "the codebase will have a layered architecture," or "everything is a plug-in." Or the new codebase may be based on a well-known architectural model, as Linux was based on Unix.

Finally, developers can learn about modular structure and option values by working on a codebase directly. If changes can be made cleanly by contributing small chunks of code, the codebase architecture is—manifestly—modular, and other things equal, the option values will be high. If changes tend to ramify throughout the system and require deep knowledge of the whole, then the codebase architecture is "monolithic," and other things equal, option values will be low. Among open source codebases, the Apache web server is highly modular, while the GnuMeric spreadsheet program is relatively monolithic (Rusnak, 2005).

We will return to modularity and option values in sections 4 and 5 when we analyze the effects of architecture on open source developers' incentives to write code. But first we need to define the developers' work and relationship to one another. We do so in the context of a simple game.

3 Games of "Involuntary Altruism"

Software is a "non-rival" good meaning that the use of the code by one developer does not prevent its use by another. In this section, we model a work environment in which non-rival goods are produced. The environment is 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 others, and the benefit occurs whether the first player wants to help the others or not. The result is a well-known game form, sometimes labeled "the private provision of public goods." Johnson (2002) was the first to apply this game form to the open source development

process and we follow his lead.

3.1 The Basic Game of Involuntary Altruism. In the interest of clarity, we begin by laying out a symmetric, two-person, one-shot game with no uncertainty as to payoffs. Consider two developers, each of whom needs a specific codebase 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.[9] 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. (This is the non-rival property at work.) 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. The latter 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 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-

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

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 α^*. We can find α^* by setting the expected payoffs of working and not working equal to one another and solving the resulting expression:

$$\alpha^* = 1 - c/v \quad .$$

As the number of players in the game increases, the equilibria change in predictable ways. For a given number of developers, N, there are N symmetric pure-strategy equilibria, which have the property that one developer works and the others all free-ride. And there is always one fully mixed-strategy equilibrium.[10] In a mixed-strategy equilibrium with N developers, the probability that any one developer works is $\alpha_N^* = 1 - (c/v)^{1/N-1}$. This number decreases as N increases, but the expected payoff per developer is always $v - c$.

3.2 The Attractiveness of the Game. For an individual developer, the game we have specified is an alternative to coding in isolation. How attractive is the game? Table 1 compares per-person payoffs in the Robinson Crusoe environment and the pure and mixed-strategy equilibria.

[10] There are also equilibria in which some players play pure strategies and some play mixed strategies.

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 Equilibria	$v - c$

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. Similarly, in the mixed-strategy equilibria, the expected payoff per person is the same as in the Robinson Crusoe environment. In expectation, a developer does not benefit by joining the collective effort.

The problem *in expectation* can be addressed in various ways. For example Johnson (2002) specifies a game in which developers have different values and costs drawn randomly from a distribution. They learn their values and costs only after "joining" the collective effort and they have no option to withdraw. Johnson shows that in a mixed-strategy equilibrium, developers with v/c above a certain threshold will work and the others will free-ride. *Ex ante*, before learning their values and costs, every developer gains in expectation from joining the collective effort. *Ex post*, however, *the workers are no better off than if they were isolated*.

In the real world, the fact *those who work* do not benefit from the collective effort can be problematic. *Ex ante*, developers who think they will have a high value or low

cost relative to others might not bother to join. *Ex post*, workers who bear all the costs and obtain no incremental benefit will be indifferent between staying in and dropping out.

In the next section we show that a modular architecture can address the asymmetry of payoffs between workers and free-riders. Intuitively, the developer of one module can benefit from code written for other modules. In this fashion a modular architecture creates opportunities for developers to exchange their work on different parts of the system.

4 Bringing Architecture into the Game

To see how a modular architecture changes the game, consider a codebase consisting of a platform plus two modules. At the time of the game, the platform exists, hence its costs are sunk. By definition, each module can be worked on separately: if developed, each will add value $v/2$ to the system at a cost of $c/2$. Thus a "total system" comprising the platform plus two modules has the same value and overall cost (excluding the platform) as the non-modular system described above.[11] However, the platform plus either module is a functional system: it is not necessary to complete all modules in order to realize value from the system. Finally, 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.

Figure 2 shows the normal form of a one-shot, simultaneous-move game played by two developers when the codebase has two modules, A and B. Each developer may

[11] Throughout this paper we will assume that modules have additive value. If modules are complements, i.e., the presence of one makes another more valuable (to all participants), then all of our results go through *a fortiori*. If modules are substitutes, then they are, by definition, rivalrous. Full and partial rivalry are considered in Appendix B. We are grateful to Joachim Henkel for pointing out the spectrum of rivalry and the possibility of complementarity.

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 but the work is now shared equitably between the two developers, and there are no free-riders.[12] Furthermore, in equilibrium both developers do better by joining *and staying in* this game than by coding in isolation. (They obtain $v-.5c$ in the game vs. $v-c$ on their own.)

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

	Developer 2: Don't Work	Work on A	Work on B
Developer 1: 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)

4.1 N Developers and j Modules. We now generalize these results to deal with an arbitrary number of modules and developers. Let j be the number of modules in a codebase architecture and N be the number of potential developers. As before, we assume the design rules and the platform already exist.

Consider an N-person game that unfolds over j discrete time periods, $t = 1, 2, ...j,$. Let T denote the time needed for one person to code the entire codebase:

[12] Note that under a modular architecture, each module's value must exceed its cost. 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.

T/j is the time needed to code one module. The developers have identical utility functions of the form: $u_i = (mv - m_i c)/j$, where m denotes the total number of modules coded by all developers, and m_i denotes the number coded by the i^{th} developer. We assume as before that $v > c$. In each time period, each developer chooses to work on a specific module or not to work.

Let the action taken by the i^{th} developer in period t be denoted s_i^t. The entire strategy space, S is then:

$$S = \prod_{i=1}^{N} \prod_{t=1}^{j} s_i^t .$$

A work allocation or strategy profile, $s = (s_1^1, \ldots s_1^j, \ldots, s_N^1, \ldots s_N^j)$ is a point in this space. We assume that the actions of each player are observable by every other player, and the coded results are available in the period in which the work is done. We also assume that there is no discounting of v or c: developers are indifferent to when they expend effort and obtain code, as long as that occurs before the horizon T.

Consider first a game of complete information and pure strategies. There are potentially a very large number of equilibria in this game, but all have the property that all modules will be coded and none will be coded more than once. This result is critical to what follows, so we state it formally as a proposition:

Proposition 1. In a game of complete information and pure strategies, all modules will be coded once and only once.

Proof. Consider a strategy profile in which some module is not coded. Then, holding the strategies of all other players fixed, one developer could improve her utility by coding that module because $(v - c)/j > 0$. Thus the strategy profile is not an

equilibrium. Conversely, consider a strategy profile in which some module is coded more than once. One of the developers coding that module could improve his utility by not coding the module. Hence this strategy profile is also not an equilibrium. *QED*.

There are $(Nj)!/(Nj - j)!$ ways of allocating work such that all modules get coded once.[13] In a simultaneous-move game where all players commit to their strategies once and for all, each of these allocations is a Nash equilibrium in which no player can unilaterally improve his payoff. However, only a few of these Nash equilibria rely on "self-enforcing" or "time-consistent" strategies. These are the so-called "subgame perfect" equilbria, in which the strategy profiles at each point in time constitute an equilibrium of the corresponding subgame. *Subgame perfect equilibria do not rely on promises that will be hard for players to carry out.* For this reason, in multiperiod settings, subgame perfect equilibria may be more descriptive of reality than equilibria that depend on players carrying out threats or promises that are harmful to their own interests.

We now partition the set of Nash equilibria into two subsets: *equitable* and *inequitable*. In an *equitable equilibrium*, work is spread out as evenly as possible, hence the difference in workload between any two developers is at most one module. All other equilibria are *inequitable*. We have:

Proposition 2. The inequitable equilibria are not subgame perfect.

Proof. The full proof is in Appendix A. A sketch of the proof is as follows. In an inequitable equilibrium, a "high-workload" developer can improve her lot by (1) waiting until the last period(s); and then (2) coding one module. In the last period, all

[13] This is the total number of ways to assign j modules to Nj time blocks, with one and only one item per block.

remaining modules will be coded: in effect, threats by other developers not to work in the last period are not consistent with equilibrium in the last-period subgame as long as some modules remain to be coded. All high-workload developers are unambiguously better off playing the "wait-then-code" strategy. Thus the initial inequitable allocation is not subgame perfect.

A potentially unrealistic feature of the equitable equilibria identified by Proposition 2 is that, given automatic revelation (as we have assumed), much work gets deferred until just before the horizon. In fact, this is an artifact of our assumptions about time preference—that developers assign equal value to all events before the horizon, T, and zero value to events after the horizon. If we assume that "normal" geometric discounting prevails, then the timing of work will depend the short-term value of code vs. the developers' discount rates. Developers with high discount rates will code early, while those with low discount rates will wait. Thus if there is one developer with a very high discount rate and all others have low discount rates, the first developer will do all the work (early) and the resulting equilibrium will be inequitable.[14] However, *if two or more developers have high discount rates, then they will (all) go to work right away and split the work equitably among themselves.*[15]

To summarize, in a j-period game of pure strategies involving N developers with similar discount rates, all subgame perfect equilibria involve equitable allocations of work among developers. Given a modular code architecture ($j > 1$), this means that the work will be divided as evenly as possible. If $N > j$ (more developers than modules)

[14] Anyone who has washed the dishes rather than letting them sit in the sink has experienced this equilibrium firsthand.

[15] This result points to another potential advantage of collective development under a modular architecture: the codebase can be completed more quickly by many people working in parallel than by one person coding in isolation.

then j developers will work, and the number of free-riders will be commensurately reduced. If $N \leq j$ (more modules than developers) then all developers will work, and there will be no free-riders.

With a modular architecture, an equitable allocation of work also means that the payoffs to "high-workload" developers in the collective process will be higher than payoffs to Robinson Crusoes. With a monolithic architecture, as we saw, *for those who work,* participating in the collective effort only weakly dominates coding in isolation. In contrast, with a modular architecture, joining and staying in the collective development effort *strictly* dominates coding in isolation for workers and non-workers alike.

Interestingly, in this stylized model, the presence or absence of free-riders is immaterial to the worker's incentives to be part of a collective development process. The factors that tilt a worker's calculations in favor of joining and staying in are (1) the presence of other *workers* (e.g., developers with similar discount rates); and (2) the existence of a modular architecture that allows developers to work on different modules and then bring their work together into a functioning whole system.

Appendix B describes the impact of changing some of the assumptions of the model. Specifically, we look at what happens when code is (1) not perfectly modular; or (2) partially rivalrous. Appendix C analyzes the mixed strategy equilibria of this game.[16] In all cases, although the details change, the broad results remain the same: in equilibrium, a modular architecture decreases the number of free-riders and increases expected payoffs to the developers who work.

[16] The results for the mixed-strategy equilibria depend in interesting ways on how the mixtures are set up. For example, do all developers "roll the dice" on each module or do they form specialized subgroups focused on particular modules? As it turns out, specialized groups both diminish free-riding and increase the probability that all modules will be coded. See Appendix C for details.

5 Option Value in a Game of Involuntary Altruism

Up to this point we have assumed that the value of the system and the individual modules were known *with certainty* at the beginning of the game. 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 assume that developers are risk-neutral expected-value maximizers, and that coding intervals are short enough so that we can ignore their time preferences.

Let the value of a system be modeled as a random variable, X. When outcomes are uncertain, it is well-known that "duplication of effort," in the sense of mounting several design experiments aimed at the same target, may be desirable. Thus 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.[17] Formally:

$$Q(X;k) = E\max(X^1, \ldots, X^k, 0) \quad ;$$

where $X^1, \ldots, X^k$ are the realizations of the individual trials. Here "0" denotes the normalized value of the best pre-existing design (if any). We assume that the support of X ranges both above and below zero, thus a new design may be better or worse than its predecessor. To complete our notation, let $\Delta Q(X;k)$ denote the difference between $Q(X;k)$ and $Q(X;k-1)$. For any distribution, $Q(X;k)$ is increasing and concave; hence

[17] 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.

$\Delta Q(X;k)$ is positive and decreasing in k.[18]

5.1 A Non-Modular Architecture with Two Developers. We first look at the effect of option value on the basic game with two developers and one module. To facilitate comparison with results in the previous section, we assume that the expected value of one design experiment equals the value of the non-modular system: $Q(X;1) = v$. 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, each has the option to take whichever codebase turns out to be more valuable after the fact. The expected payoff to each is then: $Q(X;1) + \Delta Q(X;2) - c$. The second term in this expression is the option value created by the second development effort, and is strictly positive. (The normal form of this game is shown in Appendix D.)

The nature of equilibrium in this new game depends on the magnitude of the option value in relation to a developer's cost of effort. If $\Delta Q(X;2) > c$, <Work, Work> is the unique Nash equilibrium of the game, and there is no free-riding. The case is a little more complicated if $\Delta Q(X;2) \leq c$. In this case, there will be two pure-strategy Nash equilibria in which one developer works and the other does not and a single mixed-strategy equilibrium in which each developer works with probability β^*. In the mixed-strategy equilibrium, however, the probability that either developer works is higher than in the corresponding game with no option value, hence the probability of free-riding is lower. (See Appendix D for details.)

What this means is that, even without modularity, in the presence of (enough)

[18] Aoki and Takizawa (2002).

option value, developers have strictly positive incentives to join and stay in a collective development effort. The reason is that when design outcomes are uncertain, parallel work is not necessarily redundant. An isolated developer loses out on the possibility that another developer's code might be superior to her own. This possibility in turn can tip the balance of incentives in favor of joining and staying in a collective process.

What happens if modularity and option values are combined? First, we shall show that, other things equal, "more modular" architectures increase option value. Second, in the context of our game, creating more modules or more option value increases the number of developers who will work in equilibrium.

5.2 Modularity Increases Option Value. The first of these results is based on a theorem proved by Robert Merton in 1973.[19] The theorem, loosely stated, says "a portfolio of options is worth more than an option on a portfolio." However, Merton's theorem must be reframed to apply to design processes. To do this, we make three assumptions. First, we assume that the value of a codebase can be expressed as the sum of the realized values of a set of underlying components. Second, the component value distributions are the same under all architectures in which modules are combinations of discrete components.[20] Third, component outcomes are not perfectly correlated. We now have:

Proposition 3. Under the above-stated assumptions, for any set of component probability distributions, a modular architecture has a higher expected value than the corresponding monolithic architecture. Furthermore, any architecture that "modularizes" two components has a higher expected value than the corresponding

[19] Merton (1990, Theorem 8.7, pp. 265-6).
[20] In other words, components are primitive elements that may not be spread out across modules. Modules, however,

architecture that "integrates" those components into one module.

Proof. See Appendix E.

5.3 More Modules and More Option Value Attract More Developers. In the game we have described, as the number of modules and the option values embedded in the system increase, more developers will work in equilibrium. Because Proposition 3 does not depend on distributional assumptions, we believe this is a general result. But we will prove it in the context of normal probability distributions.[21] Thus in a system of j modules, we assume that module payoffs are symmetric and normally distributed with mean zero and variance, σ^2/j: $X_j \sim N(0, \sigma^2/j)$. In this expression, σ, called "technical potential," is a measure of the overall option value embedded in the system.[22]

Under these assumptions, the value of the whole system, if there are k developers working on each module, can be written as a function of k, j, and σ:

$$V(k; j, \sigma) = \sigma j^{1/2} Q(k) \quad .$$[23]

Now consider a one-shot, simultaneous-move game of complete information involving N developers working on a codebase of j modules with technical potential σ. We assume that $N > j$, thus developers can either work or free-ride. If a developer works, she selects a module to be the focus of her effort and incurs a cost, c/j. Then:

Proposition 4. In the equilibrium of a game of involuntary altruism, the number of developers who voluntarily choose to work is increasing in both the number of

---

may contain several components.

[21] Sums of normal random variables are normally distributed, hence normal distributions make it easy to compare different modular architectures.

[22] It is analogous to "volatility" in financial option theory.

[23] Baldwin and Clark (2000, Chapter 10).

modules, j, and the technical potential, σ, of the system.

Proof. See Appendix F.

Proposition 4 states that higher option value (measured by σ) combined with the ability inherent in modular systems to mix-and-match "the best with the best," *increases every developer's incentives to work*. Given a fixed pool of developers, these incentives reduce the fraction of developers that will free-ride. Thus, within the confines of our model, we conclude that modular code architectures with high option value do mitigate free-riding in a voluntary, collective development process. Such architectures also increase the advantages *to every developer* of working within the collective process vs. coding in isolation.

An immediate corollary of Proposition 4 is that modularity and option value are economic complements: more of one makes more of the other more valuable (Milgrom and Roberts, 1990).[24]

To illustrate the magnitude of the effects of modularity and option value on the effort of volunteers, table 3 computes the number of developers who will work on a codebase in equilibrium for different degrees of option value and modularity. It shows that in the presence of option value, projects not worth undertaking under a monolithic architecture may attract tens or even hundreds of self-interested developers under a sufficiently modular architecture.

[24] This follows from the value function of the system: $V(k^*; j, \sigma) = \sigma\, j^{1/2}\, Q(k^*)$, and the fact that (by Proposition 4) equilibrium k^* is increasing in both j and σ.

Table 3
The Number of Developers Working in Equilibrium as a Function of Option Value per Trial ($Q(1)/c$) and the Number of Modules (j)

No. of Modules	Option Value of One Trial/ Cost of One Trial				
	25%	50%	100%	150%	200%
1	0	0	0	2	3
5	0	5	15	25	30
10	0	20	40	60	80
15	0	45	75	120	150
20	20	60	120	180	220
25	25	75	175	250	300

5.4 Competition between a Firm and a Collective Development Process. The products of a proprietary firm and a collective development process may compete with one another. For example, IBM's and DEC's networking businesses competed with the early Internet (they failed in that competition and were disbanded). Microsoft Windows™ today competes with Linux in the server operating system market. How does code architecture affect such competition?

Firms and collectives will exploit the same architecture differently. In general, a firm, which can aggregate revenues across consumers, will invest more to exploit the option value of an architecture than a collective, whose members get only the use value of the modules they code. Thus we have:

Proposition 5. *Ceteris paribus*, a proprietary firm with a "large enough" customer base will carry out more design searches and supply a higher-quality system (in expectation) than a collective development process.

Proof. See Appendix G.

However, the value of a collective development process does not have to come from using the system. Value can come in the form of *price discipline* on a proprietary firm resulting from competition between two systems. Our model indicates that the

value of price discipline is greatest when the underlying architecture is modular and/or has high option value.

To see this, consider an architecture and a system with j modules, σ technical potential, and N users. For simplicity, we assume that the users have identical preferences: in expectation each is willing to pay $\sigma\, j^{1/2}\, Q(k)$ for a system that has had k development efforts per module. Consider a proprietary firm that is a monopolist in this product market: let k^p denote the number of developers working on each module at the firm under its optimal strategy. After the development work is done, the value of the system (in expectation) is N times each user's willingness to pay: $\sigma\, j^{1/2}\, Q(k^p)\, N$. (Note that once the coding is done, development costs are sunk.)

Now consider a collective development process based on the same architecture that yields a competitive system. In equilibrium, the collective will have k^* developers working on each module of the system, where $k^* \leq k^p$ (by Proposition 5). After the development work is done, the value of *this* system to users (again in expectation) is $\sigma\, j^{1/2}\, Q(k^*)\, N$, and *its cost to each user is free*. (Remember development costs are sunk once coding is complete.) Faced with such competition, the proprietary firm, must drop its price to $\sigma\, j^{1/2}\, [Q(k^p) - Q(k^*)]$ to capture any sales. The revenue loss to the proprietary firm, which is a value gain to consumers,[25] is:

$$\text{Value Gain to Consumers} = \sigma\, j^{1/2}\, Q(k^*)\, N$$

This value gain in turn is split between those who work (by coding a module) and those

[25] In more complex settings, the revenue loss to the proprietary firm would be a value gain to consumers *and complementors*.

who don't (the free-riders).

Moving backward in time, consider the expected value gain *to workers* from joining the collective effort. For them:

$$\text{Value Gain to Workers} = \sigma j^{1/2} Q(k^*) N - c/j \quad .$$

From Proposition 4, we know that k^*, hence $Q(k^*)$ increases with j and σ. It follows immediately that the value gain to workers from the price discipline exerted by the collectively developed codebase on the proprietary codebase is increasing in both modularity and option value. This is true *even if the collectively developed codebase is inferior to the proprietary codebase and the proprietary codebase ends up accounting for all sales in the actual market*.

What this means is that, according to our model, self-interested developers may join collective development efforts whose main effect is to exert price pressure on proprietary firms. Those collective efforts in turn will have the greatest value *to workers* when the underlying architectures are highly modular and/or have high option values. Thus our model predicts that workers will gravitate to collective development projects based on modular architectures with high option value because those projects offer "more bang for the buck."[26]

If the proprietary firm anticipates the formation of a competitive collective development project in its product market, will it still enter? It depends. The proprietary firm's *ex ante* net present value in the presence of the collective process is:

$$\text{NPV} = \sigma j^{1/2} [Q(k^p) - Q(k^*)] N - c k^p \quad .$$

[26] The "buck" here is a unit of coding effort.

Here $\sigma j^{1/2} [Q(k^p) - Q(k^*)]$ represents the (expected) quality difference between the proprietary and the collectively developed systems. Clearly if the expected quality difference and/or the market size, N, are large enough (relative to the cost of coding), the proprietary firm should enter even if it knows it will face competition from a collective. However, the NPV *can* be negative, in which case, given rational expectations, the proprietary firm should not enter, and the collective will end up being the sole supplier of the system.

6 But Will Developers Voluntarily Reveal Their Code?

Up to this point we have assumed that, if a developer works on any module of a codebase, his code is automatically revealed to the other developers (including 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 or has enough option value relative to the cost of coding 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 of communication: the cost might be as small as the effort of composing an email, but it is there. Furthermore, contributed code must be integrated into a central codebase before it can be useful to others. Part of this cost of integration is often borne by the original codewriter.

Voluntary revelation combined with costly communication and integration creates an interesting hurdle for the collective development process. The nature of the hurdle can be seen most starkly if we go back to the two-developer, two-module case. Accordingly, let us assume that two developers have joined a collective process, 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. As before, let v denote the value to each developer of the whole system and let $.5v$ denote the value of one module. Let $.5r$ denote the cost to one developer of revealing her code. Included in r are the costs of communicating the code and of integrating the modules into a system.[27]

The result is a one-shot Prisoners' Dilemma game, whose unique Nash equilibrium is <Don't Reveal, Don't Reveal>. (The normal form of the game is shown in Appendix H.) As a general rule, however, a prisoners' dilemma can be "fixed" by adding benefits and reducing the "incentives to defect", r, to the point where the equilibrium shifts to one of mutual cooperation. Thus, in a two-module architecture, suppose a benefit of $.5f$ accrues *to the publisher* of each module. If $f > r$, then <Reveal, Reveal> becomes the equilibrium of the game.

The benefits subsumed under the symbol f can take many forms. They can have economic value, like reputation (Lerner and Tirole, 2002a), or personal value, like the knowledge one has reciprocated or helped someone else (Fehr and Falk, 2002; Benkler, 2002). And in a multi-period setting, the future value of continuing to cooperate can create a benefit that overcomes the one-shot incentive to defect (Axelrod, 1984).

The key point to be gleaned from this analysis is that *the benefits included in f only have to compensate developers for the costs of communication and integration, not the cost of their coding effort*. The value-in-use of the codebase can "pay for" the coding effort. And the ability to exchange different modules or access option values can bring

[27] We assume that $.5r < .5v$, so that the cost of communication and integration is less than the value of the revealed

developers together in a collective development process. Thus only the last steps in the overall process—communication and integration—require compensation above and beyond the simple right to use the codebase.[28]

7 Other Theories and Models of the Open Source Development Process

In this section, we relate our theory and model to others that have been advanced 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 model.

The first comprehensive theory of the open source development process was put forward by Eric Raymond in three essays first published on the Worldwide Web.[29] For us, the most salient points of his theory are as follows. First, he argues that "the economic problems of open-source … are free-rider (underprovision) rather than congested-public-goods (overuse)."[30] Second, he maintains that good code is generated because developers are "scratching an itch," i.e., they need or value the code itself.[31] Third, he contends that open source developers participate in a "gift culture" or "reputation game," that is, they publish code and compete to improve code in order to achieve status within the community of "hackers."[32]

Our model formalizes and extends parts of Raymond's theory, using the tools of option valuation in combination with game theory. The fact that software is a non-rival good, hence subject to free-riding, lies at the heart of our analysis. We show that free-

---

code to the other party. In a two-person game, if $.5r > .5v$, the developers are better off not communicating.

[28] The costs of communication and integration in turn can be brought down significantly by technology. In the open source context, the Internet enabled low-cost communication via bulletin boards, email, *ftp* transfers, newsgroups, and the Worldwide Web. The costs of integration have been reduced by CVS trees, automated change logs, and bug management software.

[29] 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/ .

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

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

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

riding can be reduced, and the underprovision problem eliminated (at least theoretically) by creating an architecture with modules, option value or both. The more modular and option-laden the architecture, the more attractive the collective process will be to developers. The notion of "scratching an itch" appears in our assumption that developers code modules for their own use. Finally Raymond's "reputation game" is a way (but not the only way) of addressing the prisoner's dilemma of communication and integration. (Indeed Raymond precisely describes the dilemma we have modeled in the following quote: "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.' " In our framework, this is clearly a case where $f < r$.)

Our model was also influenced by the so-called "cooking-pot model" of Ghosh (1997). In effect, our model formalizes the link suggested by Ghosh between systems of generalized exchange and a set of diverse, non-rival goods. Modules of code are diverse, non-rival goods: we showed that a system based on implicit exchanges of such goods can be both economically viable and self-enforcing as long as the prisoners' dilemma of communication and integration has been addressed.

Lerner and Tirole (2002) argued that a good reputation obtained via open source contributions can be an asset in the labor market, and thus developers' contributions to open source projects are fully consistent with economic models of rational choice. The real puzzles of the open source development process, they suggest, 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.

We have already acknowledged our debt to Johnson (2002) who was the first to model open source software development using the "private provision of public goods" framework. The key points of difference between our models were discussed in section 3 above.[33]

Harhoff, *et. al.* (2000) and Henkel (2002) both focus on the problem of the "free revealing." The fundamental tension in these models hinges on a tradeoff between the benefits obtained by the exchange of complementary modules, or another party's improvement of the design, and the damage caused by a competitor's use of the design. In our model, we ruled out competition between developers *a priori,* by assuming that software is a non-rival good. To a first approximation this is true for the *users* of software, but producers can be fierce rivals. Within our model, the effects of such competition can be understood by locating goods along a "spectrum" of rivalry. In Appendix B we showed that as software ranges from full rivalry to non-rivalry, there is a critical threshold of rivalry. *Partially rival* goods that fall short of this threshold can be developed collectively, and our analysis of modularity and option value applies in these cases. Goods that exceed the threshold cannot be developed collectively.

James Bessen (2002) 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 complex codebases create a combinatorial explosion of "use products," and thus, a commercial firm may

[33] Johnson also argues that "nonmodularity will sometimes temper the free-riding present in an open source environment" (p. 660), a conclusion seems to run in the face of our analysis. The disagreement arises because of a subtle difference in our respective definitions of "free-riding." In a mixed strategy equilibrium, we define "free-riding" as the probability of not working at all (see Appendix C). In contrast, Johnson equates free-riding with the probability that a developer will not work *on a particular module.* Even if a developer's propensity to work on each module goes down, the probability that he will work (on some module) may still go up as the number of modules increases.

elect to focus on the largest groups of users. Users whose needs are not addressed may then take it upon themselves to develop and debug their own software. Our model suggests a different (though not contradictory) answer to Bessen's question. We argue that architecture, specifically the modularity and option value inherent in an architecture, can change developers' incentives in ways that increase the suppy of effort and mitigate free-riding in a collective process.

Finally Eric von Hippel and Georg von Krogh (2003) go beyond the above-mentioned formal models and argue that the open source development process is "a promising new mode of organization." Unlike classic private goods (produced for sale) or public goods (funded by taxation), the open source process produces a public good—the codebase—that is "self-rewarding" in the sense that developers choose what code to write and thus get the code they most want. Building on their argument, our model shows that modular code architectures with high option value are especially well-suited to a "self-rewarding" style of organization. In the context of a voluntary collective process, modular architectures with high option value will elicit higher levels of effort and participation in equilibrium than monolithic, low-option-value architectures.

8 Conclusion

In this paper we have used a simple and stylized model to make the case that the architecture of a codebase—specifically its modularity and option value—affect developers' incentives to work within the framework of the open source development process. Thus, we argue, code architecture can have a major impact on the sustainability and value of such processes.

Our analysis yields three predictions that are potentially testable in large

samples. First, our model predicts that, *ceteris paribus*, open source codebases that are more modular or have more option value will attract more voluntary contributions (effort) than codebases that are monolithic or have low option value. Second, modularity and option value are "super-additive" in their effects: more of one increases the impact of the other. And third, proprietary firms will be under more price discipline from open source development when the open source codebase is highly modular and has high option value.

Our model was motivated by the open source development process, but it applies to any non-rival good as well as to partially rival goods whose rivalry falls below a certain threshold. Thus it is appropriate to ask: where else might we see systems of work fueled by voluntary effort within architectures that are modular or have option value? To conclude this paper, we offer some conjectures as to where such "architectures of participation" may be found.

First, *all design processes have option value*. This means that user-innovators (who are not in competition) *always* have incentives to form voluntary collective groups ("communities") for the purpose of sharing and improving designs. Such communities are in fact quite common and are the focus of ongoing research (Franke and Shah, 2003; Hienerth, 2004; von Hippel, 2005; Shah, in this volume). Our model suggests that the more modular and option-rich the underlying designs, the larger and more active the user-innovator communities are likely to be. And, for high levels of modularity and option value, it may be difficult for proprietary firms to keep up with the users' innovations and remain profitable. Firms in such markets must then develop strategies that leverage the users' innovative effort. Providing "toolkits" to assist the users' in their design activities is one such strategy (von Hippel, 2005).

Second, many non-rival goods have a "natural" modular structure. Peer-to-peer file sharing networks are a case in point: each user can take care of her own files, but each also gains from having access to the files of others. This observation—essentially about the value of modularity—applies equally to Napster and the inter-library loan program among universities.

Third, the participants in a voluntary collective process can be firms as well as individuals. For example, trade shows are a quintessential collective enterprise in which the modules (booths and exhibits) are freely contributed by firms. The publicity and traffic generated by the show is a "public" good from which all firms in the show derive benefit. However, in an interesting parallel with open source development, the firms in a trade show also use their exhibits to compete in a reputation game.

Finally when firms are involved in a mutually beneficial, voluntary collective effort, the key is to control potential rivalry, so that each firm can be sure that it is gaining, not losing, by the arrangement (Henkel, 2004). The need to provide such assurances may in turn require changes in the methods by which goods are revealed and shared. For example, firms in the direct mail industry often share mailing lists, but no firm ever publishes its mailing list for all to see. Instead, mailing list transfers are structured as swaps (with tight restrictions).

In terms of our framework, the underlying architecture of mailing lists is modular and has option value. Thus, by our argument, it may be very profitable for direct-mail firms *that are not rivals* to engage in list sharing. But some direct-mail firms *are* rivals, and thus the potential cost of indiscrimate revelation is high—much higher than in open source software development. Reflecting this difference, in direct mail, the procedures for communicating and integrating mailing lists are much less open than

the corresponding procedures in open source development. Understanding why these procedural differences exist and how they might be addressed within a collective process is a promising avenue for future research.

References

Aoki, Masahiko and Hirokazu Takizawa (2002) "Incentives and Option Value in the Silicon-Valley Tournament Game, *Journal of Comparative Economics*, 30:759-786

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

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 6/7/05.

Fehr, Ernst and Armin Falk (2002) "Psychological Foundations of Incentives," European Economic Review, 46:697-724.

Franke, Nikolaus and Sonali Shah (2003). "How Communities Support Innovative Activities: An Exploration of Assistance and Sharing Among End-Users," *Research Policy* 32(1):157-178.

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.

Godfrey, Michael W. and Quiang Tu, (2000) "Evolution in Open Source Software: A Case Study," *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, 00:131-143.

Harhoff, Dietmar, Joachim Henkel and Eric von Hippel (2003) "Profiting from Voluntary Information Spillovers: How Users Benefit by Freely Revealing Their Innovations," Research Policy, 32:1753-1769.

Hienerth, Christoph (2004). "The Commercialization of User Innovations: Sixteen Cases in an Extreme Sporting Industry," in *Proceedings of the 26th R&D Management Conference*, Sesimbra, Portugal.

Henkel, Joachim (2004) "The Jukebox Model of Innovation," CEPR Discussion Paper 4507.

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

Lerner, Josh and Jean Tirole (2002) "Some Simple Economics of Open Source," *Journal of*

Industrial Economics, 52:197-234.

Lopes, Cristina V. and Sushil K. Bajracharya (2005) "An Analysis of Modularity in Aspect-oriented Design," in *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development*, ACM Press, pp. 15-26.

MacCormack, Alan, John Rusnak and Carliss Baldwin (2006). "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," (in this volume).

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.

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

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.

Parnas, David L. (2001) *Software Fundamentals: Collected Papers by David L. Parnas*, ed. D. M. Hoffman and D. M. Weiss, Boston MA: Addison-Wesley.

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

Rusnak, John (2005). "The Design Structure Analysis System: A Tool to Analyze Software Architecture," unpublished Ph.D thesis, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA.

Shah, Sonali (2006) "Motivation, Governance, and the Viability of Hybrid Forms," in this volume.

Shaw, Mary and David Garlan (1996). *Software Architecture: An Emerging Discipline*, Upper Saddle River, NJ: Prentice-Hall.

Sullivan, Kevin, William G. Griswold, Yuanfang Cai and Ben Hallen (2001). "The Structure and Value of Modularity in Software Design," *SIGSOFT Software Engineering Notes*, 26(5):99-108.

von Hippel, Eric (2005). *Democratizing Innovation*, Cambridge, MA: MIT Press.

von Hippel, Eric and Georg von Krogh (2003) "Open Source Software and the 'Private Collective' Innovation Model: Issues for Organization Science," *Organization Science*, 14(2):209-223.

Appendices

Appendix A
Proof of Proposition 2

Proposition 2. An inequitable equilibrium is not a subgame perfect.

Proof. Initially, let $N \geq j$, so that there are at least as many developers as modules. Let s be an inequitable equilibrium, and let Developer i have a high workload in this equilibrium, coding two or more modules. She can improve her payoff by (1) waiting until the last period; and then (2) coding one module. This strategy clearly reduces her cost. Furthermore, in the last period subgame, by the same reasoning used in Proposition 1, all *remaining* modules will be coded. (In effect, threats by other developers not to work in the last period are not credible if some modules still remain to be coded.) Developer i is unambiguously better off playing the alternate strategy, and thus the inequitable equilibrium is not subgame perfect.

If $N < j$, so that there are more modules than developers, the same reasoning applies, but developers cannot wait until the last period before coding. Let τ denote the integer part of j divided by N. In order to code all modules, all developers must work each period from $T - \tau$ forward. Developer i's alternate strategy is then to wait until there are $\tau + 1$ time intervals left to go, and code one module in each of those intervals. *QED.*

Appendix B
Extensions of the Basic Model: "Nearly Modular" Code; Partial Rivalry

"Nearly Modular" Code. In the model, modules are well-separated entities with clear boundaries. In reality, however, some components may not be cleanly separated from others. Such systems are "nearly modular," or in Herbert Simon's language "near-decomposable."

Near-modularity can be reflected in the costs of coding. Specifically the sum of the cost of coding j modules in a modular architecture may be *more* than the cost of coding the same functions in a monolithic architecture. However, as long as the cost of coding *one* module is less than the value of the *whole* system, all of the results in the text will go through, although the magnitude of the incentives to participate in the collective process will be somewhat reduced. That is, in equilibrium, all modules will be coded, and in a subgame perfect equilibrium, developers will split the work equitably. Furthermore all developers will strictly prefer participating in the collective process to coding in isolation.

However, in the case of near-modularity, the equitable work allocation will be *inefficient* relative to having one developer code the whole system and supply it to the others. A profit-seeking firm could "arbitrage" the cost difference by hiring one developer to code the whole system under a monolithic architecture and selling system to others for less than their perceived cost of coding. The difference between the cost under the modular architecture and the cost under the (more efficient) monolithic architecture represents the maximum profit that such a firm could earn.

Thus when a codebase is "nearly modular," implying that the cost difference between a modular and monolithic architectures is small, our model predicts that there will be competition between proprietary firms and voluntary collective arrangements like the open source development process.

Partial Rivalry. Our model depends on the fact that code is a non-rival good: one person's use does not prevent another's. Clearly, if code were completely rivalrous so that one and only one person could consume it (like a piece of chocolate), the collective enterprise would have no reason to exist. But under *partial rivalry*, one developer can

use another's code, but will not realize as much value as if she wrote it herself.

To capture this idea, let v_{ab} denote the value of a codebase or a module that is used by Developer a but written by a different Developer b. Non-rivalry then implies $v_{ab} = v_{aa}$; partial rivalry implies $0 < v_{ab} < v_{aa}$; and full rivalry implies $v_{ab} \leq 0$. (v_{ab} can be less than zero if a and b are in competition with one another, and coding by b causes a to lose sales.)

As we said, if $v_{ab} = 0$, developers have nothing to gain by coming together and will code in isolation. But under partial rivalry, the results depend on *how much worse* the other person's code is in relation to the cost of writing it oneself. Specifically, let c_a be Developer a's cost of coding. If $v_{aa} - v_{ab} < c_a$, then a will prefer to write his own code for that module. If this condition holds true for all modules, then the situation is the same as for full rivalry: all developers will code in isolation.

But if $v_{aa} - v_{ab} \geq c_a$, Developer a will be happy (or at least indifferent) to use code someone else has written. In this case, the results in the text go through, unchanged except for the magnitude of payoffs. Propositions 1 and 2 still hold: in any Nash equilibrium, all modules will be coded, and in any subgame perfect equilibrium, the work allocation will be equitable.

Appendix C
Equilibria in Mixed Strategies

The equilibria investigated in the text involve pure strategies. Here we investigate the mixed-strategy equilibria of the involuntary altruism game with N developers and j modules. All assumptions are the same as in section 4.1 of the text: the game unfolds over j time periods, and in each period, each developer chooses to work

on a module or not. We assume that each developer first selects a module and then works on that module with probability α. As it happens, the equilibrium depends in interesting ways on how the focal modules are selected.

Case 1: All Developers Work on Modules Seriatim. Let the modules be placed in some order and suppose all developers work on the same module in the same time period. That is, module 1 is the focus of effort in period 1; module 2 in period 2, etc. Each developer in each period sets the expected payoff of working and not working equal to one another:

$$[1-(1-\alpha)^{N-1}](v/j) = (v-c)/j \quad .$$

Equilibrium α^* is then:

$$\alpha_N{}^* = 1-(c/v)^{1/N-1} \quad .$$

The expected payoff of this gamble is $(v-c)/j$. *Ex post*, however, each developer either free-rides or works. As long as someone works, the free-riders are better off than the workers.

Modularity makes the work allocation more equitable by increasing the probability that each developer does some work. The probability of any one developer not working at all is:

$$(1-\alpha_N{}^*)^j = (c/v)^{j/(N-1)} \quad .$$

Clearly this number is decreasing in j.

A potentially problematic feature of the mixed strategy equilibrium is that there is a positive probability on each round that a module will not be coded. This probability is:

$$(1-\alpha^*)^N = (c/v)^{N/(N-1)} \quad ;$$

which is increasing in N. Intuitively, as the number of developers increases, the incentives to free-ride go up, and the effectiveness of the collective effort decline. This effect is independent of the codebase's architecture. However, with a modular architecture, a slight change in the work allocation pattern can diminish the effects of high N. As we show in the next section, one can reduce "effective N" by having the developers specialize on particular modules.

Case 2: Developers Specialize by Module. Let the number of developers be greater than the number of modules and let the set of developers be divided up so that N/j developers work on each module. For simplicity, assume that N/j is an integer. (Non-integer values do not change the basic results.) In the first time period, each subset of developers plays the game of mixed strategies on "their" module. Once their module is coded, they retire, otherwise they play again on the next round.

The equilibrium probability that each developer will code a module in any round is now:

$$\alpha_{N/j}^{*} = 1-(c/v)^{j/(N-j)} \quad .$$

The probability of any developer not working on the first round is:

$$1-\alpha_{N/j}^{*} = (c/v)^{j/(N-j)} \quad ,$$

which is decreasing in j. Thus, with a modular architecture in place, having developers specialize by module decreases the number of free-riders in each round relative to having all developers work on the modules seriatim. The more modules in the architecture, the higher the probability that any given developer will work.

The probability that a module will not get built on the first round is:

$$(1-\alpha_{N/j}^{*})^{N/j} = (c/v)^{N/(N-j)} \quad .$$

This number is also decreasing in the number of modules. Finally, the probability that a particular module will not get built at all is:

$$[(1-\alpha_{N/j}{}^*)^{N/j}]^j = (c/v)^{Nj/(N-j)} \quad .$$

This number is increasing in N, but more slowly than in Case 1. Moreover, for fixed N, the number is decreasing in j. Thus given specialization by module, the more modules in the architecture, the higher is the probability that each module will be coded.

Appendix D
Additional Results for the One Module, Two Developer Game with Option Value

Normal Form of the Game

	Developer 2: Free-ride	Developer 2: Work
Developer 1: Free-ride	0, 0	v, v–c
Developer 1: Work	v–c, v	v + ΔQ(X; 2) – c, v + ΔQ(X; 2) – c

The Mixed Strategy Equilibrium

There is a unique mixed strategy equilibrium for the above game, in which each developer works with probability β^*:

$$\beta^* = \frac{v-c}{v-\Delta Q(X;2)} \quad .$$

The expected payoff to workers and free-riders alike is:

$$\text{Expected Payoff} = \frac{v(v-c)}{v-\Delta Q(X;2)} \quad ;$$

while a developer coding in isolation obtains $v-c$. $\Delta Q(X;2)>0$, hence in expectation the payoff is higher from joining the collective effort than from coding in isolation. Also a working developer can gain from the efforts his peer, because there is a possibility that the other's code will be superior.

Appendix E
Proof of Proposition 3

Assumptions:

(1) The value of a codebase can be expressed as the sum of the realized values of a set of underlying components.

(2) The component value distributions are the same under all architectures in which modules are combinations of discrete components.

(3) Component outcomes are not perfectly correlated.

Proposition 3. Under the above-stated assumptions, for any set of component probability distributions, a modular architecture has a higher expected value than the corresponding monolithic architecture. Furthermore, any architecture that "modularizes" two components has a higher expected value than the corresponding architecture that "integrates" those components into one module.

Proof. Let there be j components indexed by c. For a monolithic architecture, A, by assumption (1), the value of the system before taking options into account, X_A, is:

$$X_A = \sum_{c=1}^{j} X_c^A \quad ;$$

where X_c^A is the value contribution of each component under architecture A.

Define $\lambda_c^A \equiv E(X_c^A)/E(X_A)$. Without loss of generality, we can re-express the X_A as a weighted sum of the normalized value of the components:

$$X_A = \sum_{c=1}^{j} \lambda_c^A x_c^A \quad ;$$

where:

$$x_c^A \equiv X_c^A / \lambda_c^A = X_c^A \frac{E(X_A)}{E(X_c^A)} \quad .$$

Let B be a modular architecture that splits out j modules corresponding to the j components of the system. By assumption (2), B is "distribution preserving" with respect to A. Thus for each component $x_c^A = x_c^B$ and $\lambda_c^A = \lambda_c^B$. (In what follows, we will drop the superscripts and simply write x_c and λ_c.)

Under a monolithic architecture, the j components of the system are designed *inter*dependently. Thus at the end of the design process, designers have one option: to take the whole system or reject it. The value of this single option at the end of the design process, denoted V_A, is:

$$V_A \equiv \max(\sum_{c=1}^{j} \lambda_c x'_c, 0) \quad ;$$

where x'_c denotes the *realization* of the design process for component c.

Under a modular architecture, components can be designed independently of one another. Thus under the modular architecture B, with j modules, at the end of the design process, designers have j options: they can take the best solution component by component. The value of this so-called "portfolio of options" at the end of the design

process, denoted V_B, is:

$$V_B \equiv \sum_{c=1}^{j} \max(\lambda_c x'_c, 0) = \sum_{c=1}^{j} \lambda_c \max(x'_c, 0) \quad .$$

It is well known that the function $f(y) = \max(y, 0)$ is convex. This implies that in all states of nature the value of the single option will be less than or equal to the value of the portfolio of options: $V_A \leq V_B$. Thus B weakly dominates A in outcomes. As a result, $E(V_A) \leq E(V_B)$. (The inequality is strict if, for some component c, there is a non-zero probability that the $x_c < 0$.)

This argument generalizes to any number of trial designs (any number of X's). The function $f_k(y^1, ..., y^k) = \max(y^1, ..., y^k, 0)$ is convex, thus for any joint realization of k design outcomes on each of j components, $(x'^1_1, ...x'^1_j, ..., x'^k_1, ...x'^k_j)$:

$$V_A \equiv \max\left(\sum_{c=1}^{j} \lambda_c x'^1_c, ..., \sum_{c=1}^{j} \lambda_c x'^k_c, 0\right) \leq \sum_{c=1}^{j} \lambda_c \max(x'^1_c, ..., x'^k_c, 0) \equiv V_B \quad .$$

Again, the modular architecture weakly dominates the monolithic architecture in outcomes, and thus $E(V_A) \leq E(V_B)$. This inequality in turn holds strictly if outcomes within each trial are not perfectly correlated as stipulated by assumption (3). (Intuitively, if component outcomes are perfectly correlated within each trial, then some trial will obtain the maximum value for each and every component. In that case, there is no benefit to the modular architecture, whose virtue is that it allows designers to "mix and match" component outcomes across trials.)

Now consider modular architecture C, which is like B except for the fact that two modules have been integrated into one. Again because of the convexity of the "max" function, for any joint realization of k design outcomes on each of j

components, $(x'^1_1,...x'^1_j, ..., x'^k_1,...x'^k_j)$:

$$V_C \equiv \max\left(\sum_{c=1}^{2}\lambda_c x'^1_c, ..., \sum_{c=1}^{2}\lambda_c x'^k_c, 0\right) + \sum_{c=3}^{j}\lambda_c \max(x'^1_c, ..., x'^k_c, 0)$$

$$\leq \sum_{c=1}^{2}\lambda_c \max(x'^1_c, ..., x'^k_c, 0) + \sum_{c=3}^{j}\lambda_c \max(x'^1_c, ..., x'^k_c, 0) \equiv V_B$$

Once again, the more modular architecture, B, weakly dominates the less modular C in outcomes. Therefore $E(V_C) \leq E(V_B)$, and the inequality holds strictly as long as component outcomes are not perfectly correlated. *QED*

Appendix F
Proof of Proposition 4

Assumptions:

(1) As in Appendix E.

(2) Module payoffs are symmetric and normally distributed with mean zero and variance, σ^2/j: $X_j \sim N(0, \sigma^2/j)$

Proposition 4. In the equilibrium of a game of involuntary altruism, with N developers, j modules, and option values proportional to a technical potential parameter, σ, the number of developers who voluntarily choose to work is increasing in both the number of modules, j, and the technical potential, σ, of the system.

Proof. Consider a strategy profile $s(k)$ in which $k < N$ developers work *on each*

module of the system.[34] Note that k can only take on integer values. The payoffs to the $k+1^{st}$ developer are:

Free-ride: $\sigma j^{1/2} Q(k)$;

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

If $c < \sigma j^{1/2} \Delta Q(k+1)$ then in equilibrium (at least) $k+1$ developers should work. If $c \geq \sigma j^{1/2} \Delta Q(k+1)$, then k or fewer developers should work. It follows that the equilibrium number of developers per module, k^*, is:

$$k^* = k \quad \ni \quad \sigma j^{1/2} \Delta Q(k+1) \quad \leq \quad c \quad < \quad \sigma j^{1/2} \Delta Q(k) \quad .$$

$\Delta Q(k)$ is strictly decreasing, thus equilibrium k^* is unique.

The function $\sigma j^{1/2} \Delta Q(k)$, which implicitly defines equilibrium k^* is increasing in both σ and j, but decreasing (as a step function) in k. Higher j or higher σ moves the function up relative to c, which in turn tends to increase the intercept k^*. (Because k can take on only integer values, small increases in j or σ may not move k^* to the next higher value. However, sufficiently large increases in j or σ will move k^* upward.) Thus more modules or higher option value will tend to support more development effort per module.

The total number of developers who voluntarily work in equilibrium is the product of the number of modules times the workers per module: jk. We have just shown that equilibrium k^* is increasing by steps in j and σ. Thus *a fortiori*, the total number of developers, jk^* is also increasing in j and σ. *QED*

---

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

Appendix G
Proof of Proposition 5

Assumptions:

As in Appendix F.

Proposition 5. *Ceteris paribus*, a proprietary firm with a "large enough" customer base will carry out more design searches and supply a higher-quality system (in expectation) than a collective development process.

Proof. In the course of proving Proposition 4, we showed that in a collective development process, the equilibrium number of developers per module, k^*, is:

$$k^* = k \quad \ni \quad \sigma j^{1/2} \Delta Q(k+1) \quad \leq \quad c \quad < \quad \sigma j^{1/2} \Delta Q(k) \quad .$$

$\Delta Q(k)$ is strictly decreasing, thus equilibrium k^* is unique.

For a proprietary firm, the incremental profit obtained by increasing development effort per module from k to $k+1$ is $\sigma j^{1/2} \Delta Q(k+1) N$ and the cost is c. We stipulate that the firm has more than one customer, thus $N > 1$. From this it follows that optimal k^p for a proprietary firm is:

$$k^p = k \quad \ni \quad \sigma j^{1/2} \Delta Q(k+1) N \quad \leq \quad c \quad < \quad \sigma j^{1/2} \Delta Q(k) N \quad .$$

Dividing through by N obtains:

$$k^p = k \quad \ni \quad \sigma j^{1/2} \Delta Q(k+1) \quad \leq \quad c/N \quad < \quad \sigma j^{1/2} \Delta Q(k) \quad .$$

$\Delta Q(k)$ is strictly declining in k and $c/N < c$ for $N > 1$. It follows immediately that $k^p \geq k^*$. Furthermore, for high enough N the inequality is strict. Thus a firm with N above some threshold will invest in more design searches per module than the comparable collective development process. In that case, and the expected value of the proprietary system will be strictly higher than the expected value of the collectively

developed system:

$$\sigma\, j^{1/2}\, Q(k^{p}) \;>\; \sigma\, j^{1/2}\, Q(k^{*}) \quad .$$

QED

Appendix H
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