MANAGEMENT SCIENCE
Vol. 52, No. 7, July 2006, pp. 1116–1127
ISSN 0025-1909 | EISSN 1526-5501 | 06 | 5207 | 1116

DOI 10.1287/mnsc.1060.0546
© 2006 INFORMS

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

Carliss Y. Baldwin
Harvard Business School, Baker Library, Boston, Massachusetts 02163, cbaldwin@hbs.edu

Kim B. Clark
Brigham Young University–Idaho, Rexburg, Idaho 83460, clarkk@byui.edu

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 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; nonrival goods; free riding; open source; software development; prisoners' dilemma game; institutional economics; organizational economics
History: Accepted by Eric von Hippel and Georg von Krogh, special issue editors; received August 30, 2004. This paper was with the authors 5 months for 3 revisions.

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 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 this paper is organized as follows. In §2, we discuss code architecture and define modularity and option value. In §§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. In §6, we consider developers' incentives to voluntarily reveal code they have already created. Section 7 connects our work to other theories and 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,

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

(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 (see, for example, Shaw and Garlan 1996 and Parnas 2001) and in the fact that codebases performing the same tasks can have dramatically different architectures (see, for example, Sullivan et al. 2001 and MacCormack et al. 2006).

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.[2] 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. 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, and the device drivers (*drivers*) and architecture (*arch*) subdirectories contain most of the modules (Godfrey and Tu 2000, Rusnak 2005). In the Eclipse open source project, the application programming interface (APIs) and related code constitute the codebase's platform, and 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] 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, Chapter 13). In this analysis, we are interested in what the developers see when they write code; hence, our focus is on modularity in design.

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 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 (Baldwin and Clark 2000, Chapter 3).

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.[3] With regard to options, each time a new function is added (for example, a new device driver in Linux) or an existing function is 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; therefore, if one knows the modular structure, then it is possible to identify the options and even formally estimate the option values inherent in the codebase (Sullivan et al. 2001, Lopes and Bajracharya 2005).

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 by the initial architect(s), for

[3] One useful tool is the so-called design structure matrix mapping method. See, for example, MacCormack et al. (2006) and the citations therein.

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 being 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 being equal, option values will be low. Among open source codebases, the Apache Web server is highly modular, and the GnuMeric spreadsheet program is relatively monolithic (Rusnak 2005).

We will return to modularity and option values in §§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 "nonrival" 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 nonrival 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 does not.[4] Moreover, in a "Robinson Crusoe" environment, in which the developers are isolated and do not communicate, both developers will write code.

Figure 1 **Normal Form of a Simple Game of Involuntary Altruism**

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

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 nonrival property at work.) What's more, neither developer 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 does not. These equilibria are *efficient*, in the sense that minimum effort is expended for maximum total return. However, the equilibria are also *inequitable* because the nonworker "free rides" on the effort of the worker.

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

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. There is always one fully mixed-strategy equilibrium.[5] 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

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

[5] 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	$v-c$

attractive is the game? Table 1 compares per-person payoffs in the Robinson Crusoe environment and the pure- and mixed-strategy equilibria.

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) specified 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 showed 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 that those who work do not benefit from the collective effort can be problematic. Ex ante, developers who think that 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 nonmodular system described above.[6] However, the platform plus either module is a functional system: It is not necessary to complete all modules 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 Normal Form of a Game of Involuntary Altruism with Perfect Information: Two Developers and Two Modules

	Developer 2		
Developer 1	Don't work	Work on A	Work on B
Don't work	0,0	$0.5v$, $0.5(v-c)$	$0.5v$, $0.5(v-c)$
Work on A	$0.5(v-c)$, $0.5v$	$0.5(v-c)$, $0.5(v-c)$	$v-0.5c$, $v-0.5c$
Work on B	$0.5(v-c)$, $0.5v$	$v-0.5c$, $v-0.5c$	$0.5(v-c)$, $0.5(v-c)$

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 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.[7] Furthermore, in equilibrium, both developers do better by joining and staying in this game than by coding in isolation. (They obtain $v-0.5c$ in the game versus $v-c$ on their own.)

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,

[6] Throughout this paper, we 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; all appendices are available on the *Management Science* website at http://mansci.pubs.informs.org/ecompanion.html. We are grateful to Joachim Henkel for pointing out the spectrum of rivalry and the possibility of complementarity.

[7] 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.

we assume that the design rules and the platform already exist.

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

There are $(Nj)!/(Nj - j)!$ ways of allocating work such that all modules get coded once.[8] In a simultaneous-move game in which 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 equilibria, 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 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 on the short-term value of code versus the developers' discount rates. Developers with high discount rates will code early, whereas 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.[9] 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.[10]

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

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

[10] 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.

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), 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 nonworkers 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.[11] 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.

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 §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.[12] Formally,

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

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, $\Delta Q(X; k)$ is positive and decreasing in k (Aoki and Takizawa 2002).

5.1. A Nonmodular 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 nonmodular 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 $\Delta Q(X; 2) > c$, then $\langle \text{Work}, \text{Work} \rangle$ is the unique Nash equilibrium of the game, and there is no free riding. The case

[11] 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.

[12] 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.

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) 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 being 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 (Merton 1990, Theorem 8.7, pp. 265–266). 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.[13] 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 architecture that "integrates" those components into one module.*

Proof. See Appendix E.

[13] In other words, components are primitive elements that may not be spread out across modules. Modules, however, may contain several components.

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 that this is a general result. But we will prove it in the context of normal probability distributions.[14] 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.[15]

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:

$$V(k; j, \sigma) = \sigma j^{1/2} Q(k)$$

(Baldwin and Clark 2000, Chapter 10).

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 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 versus 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).[16]

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

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

[16] 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 2 The Number of Developers Working in Equilibrium as a Function of Option Value per Trial ($Q(1)/c$) and the Number of Modules (j)

	Option value of one trial/cost of one trial				
No. of modules	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

To illustrate the magnitude of the effects of modularity and option value on the effort of volunteers, Table 2 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.

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.*[17]

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 or has high option value or both.

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 that 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,[18] 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 who do not (the free riders).

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

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

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 or have high option values or both. Thus, our model predicts that workers will gravitate to collective development projects based on modular architectures with

[17] We are grateful to an anonymous reviewer for pointing out an error in our previous analysis and suggesting this proposition.

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

high option value because those projects offer "more bang for the buck" (the "buck" here is a unit of coding effort).

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 (NPV) in the presence of the collective process is

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

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 or the market size, N, is 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. 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 e-mail, 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 create 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 $0.5v$ denote the value of one module. Let $0.5r$ 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.[19]

The result is a one-shot prisoners' dilemma game, whose unique Nash equilibrium is $\langle$don't reveal, don't reveal$\rangle$. (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 that a benefit of $0.5f$ accrues to the publisher of each module. If $f > r$, then $\langle$reveal, reveal$\rangle$ becomes the equilibrium of the game.

The benefits subsumed under the symbol f can take many forms. They can have economic value, such as reputation (Lerner and Tirole 2002), or personal value, such as the knowledge that one has reciprocated or helped someone else (Fehr and Falk 2002, Benkler 2002). In a multiperiod 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. The ability to exchange different modules or access option values can bring 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.[20]

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 influenced our model.

The first comprehensive theory of the open source development process was put forward by Raymond

[19] We assume that $0.5r < 0.5v$, so that the cost of communication and integration is less than the value of the revealed code to the other party. In a two-person game, if $0.5r > 0.5v$, the developers are better off not communicating.

[20] 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, e-mail, ftp transfers, newsgroups, and the Worldwide Web. The costs of integration have been reduced by CVS trees, automated change logs, and bug management software.

(1999) in three essays first published on the Worldwide Web.[21] 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)" (pp. 150–153). Second, he maintains that good code is generated because developers are "scratching an itch," that is, they need or value the code itself (p. 32). 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 to achieve status within the community of "hackers" (pp. 99–118).

Our model formalizes and extends parts of Raymond's (1999) theory, using the tools of option valuation in combination with game theory. The fact that software is a nonrival good, hence subject to free riding, lies at the heart of our analysis. We show that free 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 prisoners' 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' " (p. 153). 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 (1998). In effect, our model formalizes the link suggested by Ghosh between systems of generalized exchange and a set of diverse, nonrival goods. Modules of code are diverse, nonrival 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; 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 suggested, have to do, not with motivation, but with the "leadership, organization, and governance" 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 §3.[22]

Harhoff et al. (2003) and Henkel (2004) both focus on the problem of "free revealing." The fundamental tension in these models hinges on a trade-off 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 nonrival 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 show that as software ranges from full rivalry to nonrivalry, 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.

Bessen (2002) asked 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 argued that complex codebases create a combinatorial explosion of "use products," and, thus, a commercial firm may 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 supply of effort and mitigate free riding in a collective process.

[21] The essays, "The Cathedral and the Bazaar," "Homesteading the Noosphere," and "The Magic Cauldron," may be found in Raymond (1999).

[22] Johnson (2002) 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.

Finally, von Hippel and von Krogh (2003) went beyond the aforementioned formal models and argued that the open source development process is "a promising new mode of organization" (p. 213). 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—affects 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 "superadditive" in their effects: More of one increases the impact of the other. 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 nonrival 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, 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 2006). 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. 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 nonrival 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 interlibrary 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 are 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; 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.

An electronic companion to this paper is available on the *Management Science* website at http://mansci.pubs.informs.org/ecompanion.html.

Acknowledgments

The authors thank 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, and Sonali Shah for comments that enhanced this paper in many ways. They especially thank two anonymous reviewers and the editors of this volume for extremely careful reviews and thoughtful suggestions that allowed them to increase the rigor and scope of the analysis. Finally, they 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. The authors alone are responsible for errors, oversights, and faulty reasoning.

References

Aoki, Masahiko, Hirokazu Takizawa. 2002. Incentives and option value in the Silicon-Valley tournament game. *J. Comparative Econom.* **30** 759–786.

Axelrod, Robert F. 1984. *The Evolution of Cooperation*. Basic Books, New York.

Baldwin, Carliss Y., Kim B. Clark. 2000. *Design Rules*, Vol. 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 J.* **112**(3) 369–446.

Bessen, James. 2002. Open source software: Free provision of complex public goods. Retrieved June 7, 2005, http://www.researchoninnovation.org/opensrc.pdf.

Fehr, Ernst, Armin Falk. 2002. Psychological foundations of incentives. *Eur. Econom. Rev.* **46** 697–724.

Franke, Nikolaus, Sonali Shah. 2003. How communities support innovative activities: An exploration of assistance and sharing among end-users. *Res. 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). Retrieved May 22, 2003, http://www.firstmonday.dk/issues/issue3_3/ghosh/index.html.

Godfrey, Michael W., Quiang Tu. 2000. Evolution in open source software: A case study. *ICSM 2000, Proc. Internat. Conf. Software Maintenance, San Jose, CA, October 11–14.* IEEE Computer Society, Los Alamitos, CA, 131–142.

Harhoff, Dietmar, Joachim Henkel, Eric von Hippel. 2003. Profiting from voluntary information spillovers: How users benefit by freely revealing their innovations. *Res. Policy* **32** 1753–1769.

Henkel, Joachim. 2004. The jukebox mode of innovation—A model of commercial open source development. CEPR Discussion Paper 4507, Centre for Economic Policy Research, London, UK. Retrieved September 27, 2005, http://www.cepr.org/pubs/dps/DP4507.asp.

Hienerth, Christoph. 2006. The commercialization of user innovations: The development of the rodeo kayak industry. *R&D Management*. Forthcoming.

Johnson, Justin Pappas. 2002. Open source software: Private provision of a public good. *J. Econom. Management Strategy* **2**(4) 637–662.

Lerner, Josh, Jean Tirole. 2002. Some simple economics of open source. *J. Indust. Econom.* **52** 197–234.

Lopes, Cristina V., Sushil K. Bajracharya. 2005. An analysis of modularity in aspect-oriented design. *AOSD '05: Proc. 4th Internat. Conf. Aspect-Oriented Software Development, Chicago, IL, March 14–18*. ACM Press, New York, 15–26.

MacCormack, Alan, John Rusnak, Carliss Baldwin. 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Sci.* **52**(7) 1015–1030.

Merton, Robert C. 1973. Theory of rational option pricing. *Bell J. Econom. Management Sci.* **4**(Spring) 141–183. Reprinted 1990. *Continuous Time Finance*. Basil Blackwell, Oxford, UK.

Milgrom, Paul, John Roberts. 1990. The economics of modern manufacturing: Technology, strategy and organization. *Amer. Econom. Rev.* **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, Stanford, CA.

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

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. Ph.D. dissertation, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA.

Shah, Sonali. 2006. Motivation, governance, and the viability of hybrid forms in open source software development. *Management Sci.* **52**(7) 1000–1014.

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

Sullivan, Kevin, William G. Griswold, Yuanfang Cai, Ben Hallen. 2001. The structure and value of modularity in software design. *SIGSOFT Software Engrg. Notes* **26**(5) 99–108.

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

von Hippel, Eric, Georg von Krogh. 2003. Open source software and the "private-collective" innovation model: Issues for organization science. *Organ. Sci.* **14**(2) 209–223.