

Open Source Software As Consumer Integration into Production

Jennifer Kuan

October 26, 2000

Abstract

Open source software is emerging as a potentially important competitive force in the software industry. Open source software, which is created collectively by individual volunteers, has captured the attention of venture capitalists and has Fortune 500 companies as customers. Yet very little is known about how such software will compete with established software firms. In this paper, I propose a model for understanding this new business phenomenon, in terms of consumer integration into production. This model predicts greater productivity and thus higher quality of open source software compared to more traditional closed source software. I test this prediction empirically and find that the prediction holds significantly in two of three cases.

1. Introduction

“We invested in Cygnus [a provider of Linux embedded-systems software development tools] because the company has a tremendous position in a very interesting space and an excellent management team,” says John Johnston of August Capital. “But I didn’t know in 1997, and I’m not sure I can even tell you now, exactly how the open source business model will work out.” (Red Herring Magazine, Feb. 1999)

Two interesting phenomena are illustrated by this quote. First, as sophisticated investors, venture capitalists have identified an increasingly important business model known as open source software. Formerly known as “freeware,” this business model involves free software that can be modified by users. Under

this business model, transactions take place over the Internet, and no money changes hands between the developers of the software and the users of the software (who may be one and the same). A growing number of software programs are thriving under this business model and are beginning to challenge well-established proprietary software packages. Indeed, the increasing prominence of these free software programs is getting the attention of large computer firms. Recent announcements of support or endorsement come from IBM, Sun Microsystems, Netscape and others.¹

However, mixed with the excitement over the emerging open source business model is another telling sentiment, confusion. Clearly, venture capitalists see that open source software is important, but they have very little understanding of how open source businesses will make money. If even the smart money is confused about this business model, it is highly likely that the industry executives who are rushing to support freeware products are in the dark as well.

In this paper, I present a way of understanding this puzzling, new business phenomenon, and to answer the question, that business practitioners and investors have, of how open source software will compete with closed source software. I look at open source software as the result of consumer integration into production; i.e., the result of consumers deciding to make rather than buy a good. By comparing open and closed source software in terms of a make-or-buy problem, trade-offs can be identified and testable predictions generated. Indeed, the model predicts that when users choose to make open source software, the software will be of higher quality than closed source software, *ceteris paribus*. I test this prediction empirically, using software bug data, and find that the prediction largely holds.

Finally, I discuss the theoretical implications of the model: because the open source software business model is related to a general model of nonprofit firms, the open source study can improve our understanding of the role of nonprofits in industrial organization (especially mixed industries, which are surprisingly pervasive, such as health care, insurance, education, and R&D), the role of intellectual property rights in innovation and investment, and how an understanding of open source software can help us better understand nonprofit and for-profit research and development (R&D).

¹For example, in April, 1999, IBM decided to bundle Apache with its WebSphere Internet commerce package. Also, Sun Microsystems, Oracle, Informix, Hewlett-Packard, and Computer Associates have committed to porting proprietary software to run on free Unix operating system Linux. Similarly, Netscape's Mozilla project makes the Netscape Navigator browser open source.

2. Background and Literature Review

Open source software is also known as “freeware” because it can be downloaded from the Internet free of charge, and also because the users who download the software are free to use the software however they like, which includes being able to modify the software to fit a particular need. However, to change software requires changing its source code, the human-readable version of software that gets compiled into machine readable object code. The term “open source,” is therefore more precise in distinguishing this type of software creation and distribution from the predominant method, in which consumers purchase a license to run the object code.

Note, also, that open source software is different from “share-ware,” software whose object code is downloadable free of charge. In some cases, individuals write programs and then post them on the Internet for others to use. Along with the program is often a request for \$5 or an amount equal to the consumer’s willingness to pay (e.g., McAfee Associates, see Shapiro and Varian, p. 90). Another common use of share-ware is as a marketing tactic. Software producers often put demo versions of their software on the Internet for free download, but the demo versions are limited in functionality. Allowing users to download this software gives prospective customers a chance to evaluate the software before buying (licensing) it. (Shapiro and Varian)

The open source software movement started in 1984 with GNU, a free version of Unix created by Richard Stallman. From the start, Stallman argued that copyrights restrict users’ freedom to use and modify software and, in protest, founded the Free Software Foundation. Under copyright law, source code can be protected as creative expression. “...Source and object code, which are the literal elements of a program, will almost always be found to be protectible expression.” The United States Copyright Office began to register computer programs in 1964, but did not explicitly define “computer program” until amendments to the 1976 Act in 1980. The definition added to section 101 of the Copyright Act of 1976 is “a set of statements or instructions to be used directly or indirectly in a computer program in order to bring about a certain result.” While in rare cases some disputed components are not protected by copyright law, e.g., *Apple v. Microsoft*, in most cases, the computer program is protected, e.g., *Apple v. Franklin*. A copyright is infringed if any part of source code containing creative expression is copied without permission. But, a copyright is not infringed if a program accomplishes a task using original, independently created source code.

“The main purpose or function of a program will always be an unprotectible idea.” Thus, a programmer can legally “copy” a copyrighted piece of software by independently recreating its functionality.

As an alternative to copyrighting, Stallman created the General Public License (GPL), in which he defines “copylefting.” Copylefted software may not restrict users’ ability to use, distribute, or modify software. The one restriction on users is that modifications to software be made publicly available. (The enforceability of this contract is discussed by Heffan). In practice, open source software is available for download from the Internet. Users download it, use it, and modify it if they like, and report bugs to an on-line bulletin board. Modified software gets re-posted to the Internet for others to download and improve further. In this way, improvements and innovations are accumulated in the latest version of the software for everyone to use.

Because the open source software phenomenon is so new, the scholarly literature on the subject is limited to unpublished papers, of which I discuss a short list briefly. Lakhani and von Hippel examine the case of Apache, an open source web server. They look at how the “mundane” task of software support is performed in a user-driven environment like open source software. Lerner and Tirole examine the possible incentives for contributors to open source projects and how firms might implement such incentives. Kogut and Turcanu view open source projects as an example of how distributed software development can be conducted, where contributors are geographically dispersed. Lee and Cole look at the Linux kernel mailing list to understand the organization behind open source development. They find that while there are thousands of voluntary contributors, there is also a hierarchical division of labor. Finally, Tuomi looks at the social institutions that support open source projects, since formal organization appears to be lacking, and finds that these include peer review, reputation, and status.

Each of these papers brings some definition to the phenomenon of open source software by examining different aspects of the phenomenon. And while I will draw upon much of this existing work, none of the papers mentioned above gets at the question of how open source relates to closed source software in the marketplace. I propose that this competitive question can only be addressed by comparing open source software with closed source software.

3. A Model of Closed Source Software

To compare open and closed source software, I first present a model of closed source software, which, unlike open source software, is an animal we know. In particular, I will lay out what consumers get (consumer surplus) when they buy software from a software company.

To model this, I simply use a canonical, contract theory model of a monopolistic, price-discriminating, profit-maximizing entrepreneur. I make a standard assumption that consumers can be divided into two types (high and low), according to their willingness to pay. Finally, since the good being produced is software, I make some assumptions about the production function. For example, I assume that when it comes to software, the more effort one puts into writing it, the fewer bugs and/or more features it will have (or in any case, getting rid of more bugs or adding more features will require putting in some more effort). Also, I assume that one only has to write the program once because the program can be copied as many times as one wants at almost no cost. Therefore, I assume that the good being produced is nonrival (i.e., it has a zero marginal cost of production), so only one unit of a good is produced. I use the standard interpretation of “quantity” when only one unit of a good is produced and consumed, as “quality” (fewer bugs, more features).

In this model, I use a canonical contract theory approach, defining consumers’ types with a taste parameter, θ . Consumers with a high value of θ are willing to pay a lot for quality. For simplicity, I consider only two types, high and low, that is, $\theta = \{\theta_L, \theta_H\}$, where $\theta_H > \theta_L$. The following parameters characterize the model:

$\theta = \{\theta_L, \theta_H\}$	taste parameter
n	number of high types, $\theta = \theta_H$
m	number of low types, $\theta = \theta_L$
x	“quality” of the good
t_H	amount high type buyer pays seller
t_L	amount low type buyer pays seller

The consumers’ utility is given by

$$u(t, x) = \theta x - t$$

where x is the “quality” of the software, and t is the transfer made by the consumer to the seller (the price paid to the seller).

The profit-maximizing seller knows there are two types as described above, and also that there are n high types and m low types. If we assume that the seller

cannot discriminate between consumers, he must either separate consumers by offering a menu of products or pool consumers with a single product and a single price.

The seller maximizes profit, which is given by

$$\Pi = \max\{(n + m)t - c(x), nt_H + mt_L - c(x_H) - c(x_L)\}$$

where t is the transfer he gets for the software and $c(x)$ is the cost of producing software, n is the number of high types and m is the number of low types.

Assume $c'(x) > 0$, so higher quality is more costly to produce, and $c''(x) > 0$, so there are decreasing scale economies in quality.

First, the pooling equilibrium. If the seller does not separate the two consumers, the seller's problem is

$$\begin{aligned} \max \Pi(t, x) &= (n + m)t - c(x) \text{ subject to} \\ u_H(t, x) &\geq 0 \\ u_L(t, x) &\geq 0 \end{aligned}$$

Since only the second constraint is binding, the seller's problem becomes

$$\Pi(x) = (n + m)\theta_L x - c(x)$$

So x_p solves the first-order condition

$$c'(x_p) = (n + m)\theta_L.$$

The seller's profit is

$$\Pi(x_p) = (n + m)\theta_L x_p - c(x_p)$$

The consumer surplus, which goes to the high types because the low type's participation constraint binds, is:

$$CS_p = \theta_H x_p - \theta_L x_p$$

and total surplus is seller's profit + consumer surplus:

$$TS(x_p) = n\theta_H x_p + m\theta_L x_p - c(x_p)$$

If the seller would like to separate the two consumers, his problem is

$$\begin{aligned}\Pi(x_L, x_H, t_L, t_H) &= mt_L - c(x_L) + nt_H - c(x_H) \text{ subject to} \\ u_L(t_L, x_L) &\geq 0 \\ u_H(t_H, x_H) &\geq 0 \\ u_H(t_H, x_H) &\geq u_H(t_L, x_L) \\ u_L(t_L, x_L) &\geq u_L(t_H, x_H)\end{aligned}$$

of which only the first and third constraints bind.

$$\begin{aligned}u_L(t_L, x_L) &= 0 \\ u_H(t_H, x_H) &= u_H(t_L, x_L)\end{aligned}$$

Substituting back into the firm's maximization problem gets

$$\Pi(x_L, x_H) = (n + m)\theta_L x_L - c(x_L) + n\theta_H(x_H - x_L) - c(x_H)$$

Maximizing Π gets the first-order conditions

$$\begin{aligned}c'(x_H^*) &= n\theta_H \\ c'(x_L^*) &= (n + m)\theta_L - n\theta_H\end{aligned}$$

So the seller's profit is

$$\Pi(x_L^*, x_H^*) = (n + m)\theta_L x_L^* - c(x_L^*) + n\theta_H(x_H^* - x_L^*) - c(x_H^*)$$

The consumer surplus again goes to the high type because the low type's participation constraint binds:

$$CS_s = \theta_H x_L^* - \theta_L x_L^*$$

and total surplus is $\Pi(x_L^*, x_H^*) + CS$ or

$$TS(x_L^*, x_H^*) = m\theta_L x_L^* - c(x_L^*) + n\theta_H x_H^* - c(x_H^*)$$

In summary, because of the information asymmetry, the software company does not know the consumer's type and so cannot price discriminate perfectly. The company must therefore offer two software programs (and charge a high price to the high types and a low price to the low types) or offer one middling software program. Thus high type consumers will either pay a lot for high quality software, or get fairly low quality software that is also affordable to low types. Ordinarily, that would be the end of the story. However, because I think users also have the option to make their own software, the story now continues with a model of open source software.

4. A Model of Open Source Software

I make the claim early on in the paper, and in the title, that open source software is the product of users choosing to make their own software rather than buy it. And certainly, the perspective of Lakhani and von Hippel, Lee and Cole, and Tuomi is that users are the primary contributors to open source software. However, there are other claims about what drives open source is. So before getting into the model itself, I will discuss what I think open source software is (and what I will model), and what I think open source is not (and what I will not model). To get at what a model of open source should include, I start with a brief description of Apache.

4.1. The Apache Group

The Apache Group was formed by eight individuals who wanted to finish a project at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. The project was to create a server to complement another NCSA project, the Mosaic Web browser. The project was designed to be public domain software. The original author left NCSA in the middle of 1994 and development subsequently stalled. In late 1994, a group of enthusiasts who were using the software voluntarily got together to put all available documentation and bug fixes in a consolidated patch. The volunteers named the resulting patched up software “A patchy Web server.” The name evolved to Apache.

The Apache group of volunteers decided to use the copyleft idea to keep the software freely available. The group created a web site which contains a description of the software and the copyleft, as well as the source and object code for users to download. They also created a bug reporting electronic bulletin board, where people could report bugs and bug fixes. There is also an active electronic user group where users can post questions and other users can post answers or suggestions. Using this infrastructure, user feedback and improvements were incorporated into Apache 1.0, which was released on December 1, 1995.

The Apache Group grew to about a couple dozen people, and more than ten for-profit companies now sell Apache software bundled with support services. But because the number of potential contributors is so large, the process of improving the software could become chaotic. For this reason, it is necessary for the Apache group to have a well-defined process for testing software, incorporating changes, and releasing new versions. This process is described on the Group’s web site:

The Apache Group is a meritocracy – the more work you have done, the more you are allowed to do. The group founders set the original rules, but they can be changed by vote of the active members. Changes to the code are proposed on the mailing list and usually voted on by active members – three +1 (yes votes) and no -1 (no votes, or vetoes) are needed to commit a code change during a release cycle; docs are usually committed first and then changed as needed, with conflicts resolved by majority vote.

Our primary method of communication is our mailing list. Approximately 40 messages a day flow over the list, and are typically very conversational in tone. We discuss new features to add, bug fixes, user problems, developments in the web server community, release dates, etc. Anyone on the mailing list can vote on a particular issue, but we only count those made by active members or people who are known to be experts on that part of the server. Vetoes must be accompanied by a convincing explanation.

New members of the Apache Group are added when a frequent contributor is nominated by one member and unanimously approved by the voting members. In most cases, this “new” member has been actively contributing to the group’s work for over six months, so it’s usually an easy decision.

Furthermore, Apache is an organic entity; those who benefit from it by using it often contribute back to it by providing feature enhancements, bug fixes, and support for others in public newsgroups. The amount of effort expended by any particular individual is usually fairly light, but the resulting product is made very strong. This kind of community can only happen with freeware – when someone pays for software, they usually aren’t willing to fix its bugs. One can argue, then, that Apache’s strength comes from the fact that it’s free, and if it were made “not free” it would suffer tremendously, even if that money were spent on a real development team.

In June, 1999, the Apache Group was formally incorporated as The Apache Foundation, a not-for-profit corporation.

Notice that the original organizers of the Apache project were users of the software. As sophisticated users, which web server administrators had to be in 1994, they were also able to program. Since the program had been abandoned,

there were a lot of bugs and unfinished areas that any user would have to fix before being able to use it. Clearly, users had to fix bugs and fill in omissions because they needed to do so in order to use the software. Most likely, each user focused his efforts on the bugs and enhancements that were most important to him. It is this, that I will capture in my model of open source development.

4.2. Linux Developer Survey

Many other motives, besides the utility gain from fixing a bug or adding a feature, have been espoused by and attributed to open source contributors. These include a philosophical conviction that all software should be free, the desire for recognition by open source peers, the larger purpose of joining a social movement, the expectation that open source efforts will lead to a better job (career concerns). To get at how important these other motives are, a survey of Linux developers was conducted by Hermann, Hertel and Niedner. They find that the top three reasons for contributing to Linux are (1) “Facilitating my daily work due to better software”, (2) “Having fun programming”, and (3) “Improving my programming skills”, each being rated on average 4.6 on a scale of 1 to 5, with 5 being the most important.

They also study different categories of motives, including collective motives, relating to the general goals of open source projects such as improving software quality and software freedom; social motives such as the reaction of friends, family, or colleagues; and reward motives, based on the direct gains as a result of participation in Linux development. They find that reward motives are the strongest predictor of Linux participation, social motives have no significant influence, and collective motives have only a small positive effect.

Finally, Hermann et al’s analysis of the survey finds that “the more competent developers perceived themselves, the more lines of code they had contributed.” Other factors, such as “trust” (defined as the belief that other team members contribute their share and will not exploit others) and “valence” (the importance of the group outcomes for each member of the group) had no strong influence on contributions. They also found that “the perceived importance of own contributions for the success of the subsystem did not result in a bigger, but rather in a smaller output!”

In short, the survey of Linux developers finds that contributors contribute because they benefit directly from their efforts. They improve their own work situation, much as the early Apache founders benefited in their role as web server

administrators by fixing bugs in their web server software. Moreover, they are not strongly motivated by “social” or “collective” motives; that is, they are not driven by the motives of a larger social movement to make software free, nor are they chiefly motivated by the desire to impress their friends and colleagues. Finally, open source contributors probably do expect some free-riding (i.e., “trust” was not a factor) and some externality (i.e., the *group* outcome was not important, meaning benefits to other people are not internalized by contributors). The model I will lay out in the next section incorporates these features of contributors’ motives: they contribute to benefit themselves *directly*, i.e., they fix only their own bugs and add only the features that they themselves want.

4.3. The Open Source Model

In this model of open source production, consumers make the software for their own consumption. As before, demand is divided into two types, but this time, rather than being divided according to (high and low) willingness to pay, consumers are divided according to their ability to contribute effort. That is, consumers are divided into programmers (high) and nonprogrammers (low). Here again, $\theta = \{\theta^L, \theta^H\}$, of which there are a high types and b low types.

$\theta = \{\theta^L, \theta^H\}$	taste parameter
a	number of high types, $\theta = \theta^H$
b	number of low types, $\theta = \theta^L$
x	“quality” of the good
$c(x^H)$	amount high type “contributes”
$c(x^L)$	amount low type “contributes”

Each utility maximizing consumer calculates how much he would like to contribute, taking into account the contributions of all other consumers. Assuming common knowledge, the result is a Nash equilibrium that specifies the level of quality, utility, and cost. Each high type solves the same problem:

$$\max_{x^H} u_j^H = \theta^H(x_j^H + \sum_{i \neq j} x_i^H + \sum_{n=1}^b x_n^L) - c(x_j^H)$$

and each low type solves the problem

$$\max_{x^L} u_m^L = \theta^L(\sum_{i=1}^a x_i^H + x_m^L + \sum_{n \neq m} x_n^L) - c(x_m^L)$$

where x^H is the amount that each high type contributes, and x^L is the amount that each low type contributes.

First order conditions are

$$\begin{aligned}c'(x^{H*}) &= \theta^H \\c'(x^{L*}) &= \theta^L\end{aligned}$$

The first order conditions suggest that a high type will fix only the bugs that matter to him, a result that is reflected in the Apache story and the Linux survey. Similarly, each low type will only bother to make a contribution (by reporting bugs or requesting enhancements) if doing so is worth it to him. Intuitively, this, too, seems obvious; for example, a low type would only report a bug that he actually encountered while using the software.

Notice that there is an externality, since each consumer only considers his own benefit even though he gains from every other consumer's contribution. This reflects the Linux survey's finding that users are not motivated by "collective" motives. In addition to the externality, however, there is also the problem of free-riding. The model deals with free-riding through the parameters a and b , which represent the number of programmers (high type contributors) and the number of nonprogrammers (low type contributors), respectively. Since it is possible for a high type (programmer) to pretend to be a low type (nonprogrammer), a and b represent what consumers estimate these numbers to be, taking into account free-riding. That is, a is not the actual number of programmers capable of fixing bugs and adding features but rather the number of programmers who people think can be counted on to actually do so, probably a much smaller number.

To finish off the analysis of the model, the consumer surplus for both consumer types is positive. Each high type's consumer surplus is

$$CS^H = \theta^H(ax^{H*} + bx^{L*}) - c(x^{H*})$$

while each low type's consumer surplus is

$$CS^L = \theta^L(ax^{H*} + bx^{L*}) - c(x^{L*})$$

Since the consumers are the producers, the aggregate consumer surplus is the total surplus:

$$TS_m = (a\theta^H + b\theta^L)(ax^{H*} + bx^{L*}) - ac(x^{H*}) - bc(x^{L*})$$

5. Comparing Open and Closed Source

We now know what consumers get if they buy software from a software company and what they get when they make their own software. We also know that neither solution is perfect. On the one hand, when users buy software, high types pay a lot for high functionality or pay much less for lower quality (depending on what is profit-maximizing for the software company), but they get an information rent. On the other hand, when they make their own software, users are able to aggregate a lot of individual effort, but the result is dragged down by free-riding and externalities. It is this trade-off that users will be making when they compare the two, make or buy. To evaluate the make or buy question, we need only compare the consumer surplus in the make situation with the consumer surplus in the buy situation.

5.1. Comparison of Consumer Surplus

Notice, however, that the population of buyers is divided differently in each situation. In the “buy” case, users are either high or low willingness to pay. In the “make” case, they are programmers or nonprogrammers. Because these are two orthogonal dimensions of demand, a comparison of consumer surplus must be done according to the following 2x2 matrix. That is, whether consumers will choose to make or buy will depend on how demand is divided into the following four groups.

	programmer	nonprogrammer
High WTP	θ_H, θ^H	θ_H, θ^L
Low WTP	θ_L, θ^H	θ_L, θ^L

It is interesting to look at what each of the four groups will choose, which we will do next. But as we will see, just looking at the individual group choices alone will not predict whether users as a whole will prefer open source or closed source production.

First, let us look at the bottom two cells. Because low willingness to payers get a consumer surplus of 0 under the buy regime, while all consumers have some consumer surplus under the make regime, both of the bottom two groups would choose to make.

Next, we need to have a look at the upper cells. For consumers in the upper left cell, each consumer’s decision to make or buy comes down to a comparison

of his consumer surplus under pooling by the seller to the consumer surplus for make.² That is, the comparison comes down to whether $CS^H > CS_p$. If $CS^H - CS_p > 0$, consumers of the upper left cell will prefer open source production. Whether $CS^H - CS_p > 0$ depends on the parameters, θ^H , a , b , n , m , $c(x)$, θ_H , θ_L ; that is, intensities of preferences and the distribution of demand.

The expression $CS^H - CS_p > 0$ can be rewritten for comparative statics as

$$CS^H - CS_p = \theta^H(ax^{H*} + bx^{L*}) - c(x^{H*}) - x_p(\theta_H - \theta_L) > 0$$

Taking the partial derivative of this difference with respect to the size of various consumer types, i.e. a , b , n , and m , indicates how different demand parameters can affect the preference for open source.

$$\begin{aligned} \frac{\partial(CS^H - CS_p)}{\partial a} &= \theta^H x^{H*} > 0 \\ \frac{\partial(CS^H - CS_p)}{\partial b} &= \theta^H x^{L*} > 0 \\ \frac{\partial(CS^H - CS_p)}{\partial n} &= -(\theta_H - \theta_L) \frac{\partial x}{\partial n} < 0 \\ \frac{\partial(CS^H - CS_p)}{\partial m} &= -(\theta_H - \theta_L) \frac{\partial x}{\partial m} < 0 \text{ because } \frac{\partial x}{\partial n}, \frac{\partial x}{\partial m} > 0 \end{aligned}$$

Similarly, for consumers in the upper right cell, each consumer's decision is a comparison of CS^L to CS_p . The comparative statics are similar.

$$CS^L - CS_p = \theta^L(ax^{H*} + bx^{L*}) - c(x^{L*}) - x_p(\theta_H - \theta_L) > 0$$

$$\begin{aligned} \frac{\partial(CS^L - CS_p)}{\partial a} &= \theta^L x^{H*} > 0 \\ \frac{\partial(CS^L - CS_p)}{\partial b} &= \theta^L x^{L*} > 0 \end{aligned}$$

²Note that the consumer surplus for the separating case is smaller than for the pooling case because

$$\begin{aligned} CS_p &= x_B^*(\theta_H - \theta_L) \\ CS_s &= x_L^*(\theta_H - \theta_L) \end{aligned}$$

Since $c(x_B^*) = (n+m)\theta_L$ and $c(x_L^*) = (n+m)\theta_L - n\theta_H$, we know that $x_B^* > x_L^*$ because $c'(x) > 0$.

$$\begin{aligned}\frac{\partial(CS^L - CS_p)}{\partial n} &= -(\theta_H - \theta_L)\frac{\partial x}{\partial n} < 0 \\ \frac{\partial(CS^L - CS_p)}{\partial m} &= -(\theta_H - \theta_L)\frac{\partial x}{\partial m} < 0\end{aligned}$$

Unlike for the lower two groups, for which open source is always better than closed source, the decision for the upper two groups is not clear-cut. We do know a couple of things about the upper two groups, however. First, $CS^H - CS_p > CS^L - CS_p$.³ This means that while the comparative statics for both upper cells look very similar, the advantage of open source software is greater for programmers than nonprogrammers. Thus, it is possible for upper right consumers to prefer proprietary software while upper left consumers would prefer open source software.

So depending on the parameters of the model (demand distribution and preferences), we might expect open source to really take off, for example, if all four groups prefer it to closed source software. Or we might predict that open source will languish, because not enough upper left programmers prefer it (i.e., either the overall number of programmers is small or the number of lower left programmers is small while the upper left programmers prefer closed source). On the other hand, with different parameter values, we might instead predict that there will be an open and a closed source program, if either of the upper groups prefer closed source while enough programmers prefer open source. In other words, it seems that without knowing something about the parameter values, we make any useful predictions.

Fortunately, the model allows us to do two very useful things. First, the model identifies some of the determinants of open source formation and success, which helps us to interpret what we have already observed of the phenomenon and to make predictions about it. For example, take the lower left group of low willingness to pay programmers. This description might well fit the typical computer science graduate student. In fact, we observe that the very successful Linux operating system kernel was written and launched by Linus Torvalds when he was a computer science graduate student in Finland. Similarly, many engineering tools and utilities are open source and used by both low and high willingness to pay users who also happen to be programmers (upper and lower left side groups). If we can identify what types of users are likely to start and support open source programs, we can certainly also identify the sorts of

³ $\theta^H(ax^{H*} + bx^{L*}) - c(x^{H*}) > \theta^L(ax^{H*} + bx^{L*}) - c(x^{L*})$ if $c' > 0$ and $c'' < 0$.

users who are unlikely to do so. The model thus also provides a guide for more sophisticated marketing research if better estimation of the parameters is desired.

The second thing the model allows us to do is to make a prediction about the quality of open source software. By comparing the total surplus under open source production with the total surplus generated by closed source production, we arrive at the prediction that when users choose to make open source software, that software will be better (i.e., more will be produced) than closed source software, *ceteris paribus*.

5.2. Comparison of Total Surplus

First, I assume that the market is covered by both modes of production, so $a+b = n+m$. Next, I look at the total surplus from open source production and the total surplus from pooling.

$$TS_p = n\theta_H x_p + m\theta_L x_p - c(x_p)$$

$$TS_m = (a\theta^H + b\theta^L)(ax^{H*} + bx^{L*}) - ac(x^{H*}) - bc(x^{L*})$$

To see that $TS_m > TS_p$, assume that all consumers prefer open source software. Thus, consumers in the upper right cell are at least indifferent between open source and proprietary software, and consumers in the upper left strictly prefer open source. Then

$$CS^L = CS_p \text{ or } \theta^L(ax^{H*} + bx^{L*}) - c(x^{L*}) = x_p(\theta_H - \theta_L)$$

$$CS^H > CS_p \text{ or } \theta^H(ax^{H*} + bx^{L*}) - c(x^{H*}) > x_p(\theta_H - \theta_L)$$

Substituting $c(x^{L*})$ from the $CS^L = CS_p$ equation into the expression for $TS_m - TS_p$ gets

$$a[\theta^H(ax^{H*} + bx^{L*}) - c(x^{H*})] - x_p[(n+b)\theta_H + (m-b)\theta_L]$$

To show that this expression is positive, we can substitute from the expression for $CS^H > CS_p$ from which we know that $\theta^H(ax^{H*} + bx^{L*}) - c(x^{H*}) > x_p(\theta_H - \theta_L)$. If the above expression is positive after substituting $x_p(\theta_H - \theta_L)$ for $[\theta^H(ax^{H*} + bx^{L*}) - c(x^{H*})]$, then $TS_m > TS_p$. Substituting, we get,

$$ax_p(\theta_H - \theta_L) - x_p[(n+b)\theta_H + (m-b)\theta_L]$$

$$= (a - n + b)\theta_H - (a - m + b)\theta_L$$

$$= m\theta_H - n\theta_L$$

Since $m > n$ and $\theta_H > \theta_L$, the expression is positive. Thus, if consumers have all chosen to make their own software, the software that they make will be better than the software they would have bought.

The intuition behind this finding, goes back to the trade-off mentioned earlier between information asymmetry in the software company case and free-riding and externalities in the open source case. In the closed source case, the entrepreneur cannot perfectly price-discriminate because consumers have private information about their types. In the open source case, some programmers can be expected to shirk, while all users fail to internalize the contributions of other users.

6. Empirical Test

I would now like to test the model of open source software as consumer integration into production by testing the model's prediction of high open source quality. While claims of higher quality and robustness are regularly made by open source enthusiasts, the prediction of higher quality is certainly not an obvious outcome of the trade-off I have described above. It will therefore be interesting to see whether the model, which was needed to generate the prediction, will withstand statistical tests.

To get at the question of quality, we need to understand what “better” means in software. Certainly, there are many dimensions of quality, such robustness, hardware portability, flexibility, ease-of-use, feature set, etc. Also, the important each quality dimension varies according to user. This complexity in defining software quality makes measuring quality difficult. However, there are a few key aspects of software quality that we can use to get at a good measure of quality.

First, when I have described contributions to open source software, it has been in terms of bugs fixed or added features. Regardless what the dimension of quality, the act of improving software boils down to fixing bugs and adding features. For example, in the case of hardware portability, a bug might be a problem with a certain hardware platform, while a feature might be adding portability to a new hardware platform.

Second, in many well-managed software projects, bugs fixes and features work their way into a software program via a bug-tracking database or service-request system. The many names for this system ultimately describe the same thing: a database in which requests for bug fixes or additional features or functionality changes are logged in, assigned to a programmer, checked back in for

testing or integration into the software, and then closed. These databases are used by software developers to manage the software development process, but can be used to measure the rate of change of quality.

I propose using software bug data, especially the life-expectancy of service requests, to measure the rate of change of quality. I can then compare the rate of change of quality for open and closed source, with the prediction that open source has a greater rate of change than closed source. I further argue that compare the rate of change is better than comparing quality levels. If open source were found to have a lower or higher level of quality at a given point in time, this would say nothing of its relative quality over time. On the other hand, if open source is found to have a higher rate of improvement than closed source, then its quality level can be predicted to become higher than closed source and to remain higher.

There are questions of comparability, which the following discussion on the software development process and my data will address.

6.1. Bugs and the software development process

The fact that all software has bugs should alarm no one, since the process of software development is one of bug “discovery” and fixing. Bugs typically go through several more or less discrete steps in the process of being discovered, fixed, integrated, and then released. This process is more formal in some settings than in others. In a typical, formal process, bugs are discovered by a quality assurance group (QA), who then document the bug in a database. A separate group of people learn of bugs through this bug database and then fix the bugs. Next, the fixed piece of software gets sent to yet another group of people (usually known as “software control”) who “integrate” the fix into the next revision of the software. Each step in the process; bug discovery, bug fix, integration, release; is recorded in the bug database. However, in a less formal setting, such as at a start-up company, there might be just a handful of people working the entire software process with very little attention to database documentation of the process.

While the bug process of open and closed source programs differ from firm to firm and open source program to open source program, these processes are nevertheless very consistent across open and closed source program. The variation between open and closed source process is no greater or different than the variation from firm to firm within the closed source regime. Therefore, data

from open and closed source software development programs are comparable.⁴

There may remain some concerns about using bug life expectancies as a metric for quality, however. The most common objection is that “bigger” programs are more complex than “smaller” programs, where size is measured by lines of code. If complex programs have harder-to-fix bugs, and if harder-to-fix bugs take longer to fix than easy-to-fix bugs, then bug life expectancies would be shorter for “smaller” programs, which, presumably, are open source. Summerville (1992) provides some arguments to address this issue.

Regarding the point that bigger is more complex, “McCabe (1976) devised a measure of program complexity using graph-theoretic techniques. His theory maintains that program complexity is not dependent on size but on the decision structure of the program.” Other measures of complexity include code size, but only among other metrics. “Rather than use a single metric, Kafura and Reddy (1987) use a spectrum of seven metrics to assess the complexity of a system. These include Halstead’s effort metric, McCabe’s complexity metric, the code size, and other metrics which take into account the way in which a component uses its data.” Thus, bigger (in this case, closed source) is not necessarily more

⁴Having said this, however, I would like to mention a couple of data issues that arise in practice. First, with closed source programs, bug discoverers are often firm employees (the QA groups are inside the firm). Some bugs get reported from users (for example, a floating point bug in Intel’s Pentium chip), but most bugs are found by employees. In some bug processes, bug discoverers may fix the bug, while in others, bug discoverers are not permitted to fix bugs. Since some bug fixers consider logging bugs into the database to be a “bureaucratic hassle”, they do not log the bug when they find it, but rather after the bug has been fixed (and only then because many systems are set up so that fixes cannot be integrated unless they are documented in the database). This delay in reporting the bug affects the bug discovery date in the database, making it appear later, and the bug lifetime shorter (sometimes zero minutes). If open source programmers find and fix bugs, these bugs do not appear in the database at all, because no e-mail announcing the bug gets generated. Open source bug databases are built automatically from e-mails.

To address this problem, I have tried to eliminate from the dataset all bugs from closed source projects that have a zero lifetime.

A second data problem arises because open source projects are not as closed-loop as closed source projects. Bug fixes in closed source projects are tested after integration by someone other than the bug fixer. If the fix fails the test, the bug goes back to the programmer for further fixing. In open source projects, there is seldom any confirmation that the bug fix worked or did not work.

To address this problem, I use the fix-date, rather than the final release date, as the end-date of a bug’s life. While this underestimates the life expectancy of the bug, it does so consistently. Also, the resulting metric is still a measure of the responsiveness of programmers to bug reports.

complex than smaller (open source).

While more complex programs might generate more difficult-to-fix bugs, it certainly cannot be argued that more difficult-to-fix bugs take longer to fix. How quickly bugs get fixed is a function of how important a bug is and how severe the problem is that the bug creates. These priority and severity ratings are recorded in the databases.⁵ Thus easy-to-fix bugs are not necessarily fixed more quickly than hard-to-fix bugs.

6.2. Sample and Data

I have assembled bug databases from six programs: three open source, three closed source, listed below.

Program Type	Open Source Data	Closed Source Data
Web Servers	All bugs from 4/96 - 12/99	All bugs from 1/97 - 10/99
Operating Systems	All bugs from 9/94 - 2/00	All bugs from a single version (1/97-6/99)
User Interfaces	All bugs fixed from 2/99-1/00	All bugs from a single version (1/97-6/99)

Because there may be heterogeneity among types of programs, I do three separate comparisons. First, I look at web servers, for which Apache is the open source program. Next, I analyze Unix operating systems, of which FreeBSD is open source. Finally, I look at window-based user-interfaces for Unix, including open source Gnome⁶. The open source data are available on the Internet (web sites are listed in References). Closed source data come from private software producers. The closed source programs that I have selected are very similar to the open source programs in terms of functionality, purpose, and age. These pairs of programs are very good matches.

From each database, I use the bug discovery date and the bug fix date to calculate a response time or bug lifetime. The data are also coded for priority and severity, as mentioned above, and seem to be assigned consistently across programs. For example, while I have assigned numeric values to these, the various bug reporting systems typically rate priority as “high priority”, “medium priority”, and “low priority”. Severity ratings also fall into categories that consistently have names like, “Critical”, “Serious”, “Non-critical”, “Wish-list”. Categories are also accompanied by definitions. So for example, “critical” might be defined

⁵While all of the databases I use contain priority and severity fields, some of the databases do not assign priority ratings consistently. However, severity ratings do seem to be assigned consistently within programs and across program types. That is, bug discoverers assign severity ratings the same way whether the program is open source or closed source.

⁶For an interesting perspective on Gnome, see Weber.

as “important and no work-around”. These category names and definitions guide both open and closed source users of the databases in assigning values. I have scaled the ratings so that different programs can be compared. The data are partitioned to control for program age, general level of technology (calendar year), and program type. I then compare hazard rates (see below).

The data are summarized graphically in the attached figures. Figures 1 and 2 show bug counts by quarter for web servers and operating systems. Because I have census data for these two programs, it is possible to observe the bug discovery process over time. Each point represents the number of bugs found during that quarter. Figures 3 and 4 show how these bug counts break down according to severity level. Each bar corresponds to a point on Figure 1 or 2, where the bars are divided into three severity levels. Severity level 1 is the most severe.

Clearly, more bugs are being found in the closed source programs compared to the open source programs. How properly to interpret relative bug counts is ambiguous, though. On the one hand, it is natural to suppose that the more bugs were found, the buggier the software must have been. On the other hand, the task of software engineering depends upon discovering bugs; perhaps the more bugs found, the better the job of finding them. Because of the difficulty in interpreting these data, it helps to consider statistical bug life expectancies.

6.3. Statistical Model

To compare bug life expectancies, I use a proportional hazard model (Kiefer, 1988), because its interpretation is straightforward. This model compares two hazard rates by taking the ratio of the two. The hazard rate is the rate of failure or death. In our case, since we are looking at software bugs, the higher the hazard rate, the more quickly the bug will “die” or get fixed. So a high hazard rate is a good thing because it means that bugs get fixed more quickly.

Because we are taking the ratio of two hazard rates, it is easy to see that if the hazard rates are equal, the ratio will be 1. The way I have the ratio set up, if the ratio is greater than 1, it means that closed source bugs take longer to get fixed than open source bugs. If the ratio is less than 1, open source bugs take longer than closed source bugs. My theoretical prediction is that, on average, open source bugs get fixed more quickly than closed source bugs, so the ratio is predicted to be greater than 1.

The web server and operating system data are right-censored, so some of

the bugs are not fixed at the time the data are collected. The model takes into account this right censoring.

6.4. Results for Web Servers

Figure 5 and Table 1 show the proportional hazard ratio over time for web servers. Again, the time scale is the same as that used in Figures 1 and 3, and the comparison is of open and closed source bugs according to the quarter in which they were discovered. Figure 5 shows the comparison of hazard rates for bugs that were found in a particular quarter. Figure 5 shows that while closed source bugs that are found early on in the project's life take much longer to get fixed than open source bugs found early on in its project's life, closed source bugs catch up and eventually overtake open source bugs.

In this tortoise and hare sort of race, it is very difficult to tell whether open source bugs are fixed more quickly on average than closed source bugs. To determine this, I take the average hazard ratio, listed in Table 4. The average hazard ratio for web servers is slightly positive, but not significant. In other words, it looks like a tie.

6.5. Results for Operating Systems

Figure 6 and Table 2 show the results for the operating systems comparison. As with the web servers, Figure 6 shows the ratio over time, according to when the bugs were found. The curve of the resulting graph is strikingly similar to the results for web servers; again, a tortoise and hare story. However, this time, the closed source bugs only approach the same hazard rate as open source bugs. Clearly, the winner in this race is the open source operating system.

While it is interesting to look at how bugs found in different stages of the project's life get fixed at different rates (shown in Figure 6) the average hazard ratio (Table 4) is, again, probably the more appropriate measure in addressing the question of whether open source bugs are responded to more quickly than closed source bugs. The resulting hazard ratio is positive and significant, suggesting that open source bugs are indeed fixed more quickly than closed source counterparts.

6.6. Results for Windows Interfaces

The open source windows interface program is, unfortunately, incomplete. The sample that I have is all bugs that were *fixed* within a certain period (12 months). I selected the corresponding data from the closed source database which contains a census of bugs like the other programs in my test. Thus, the analysis of this program is not as complete as that of the other two programs. Also, the data are not right-censored, since I have only bugs that have been fixed.

Figure 7 shows the proportional hazard ratio over time, where the data are bugs that are fixed in a given quarter (rather than bugs that were found in a given quarter). The shape of the graph for windows interface hazard ratios is therefore completely different from the shape of the web server and operating system hazard ratios; the graph of the windows interface hazard ratios does not organize the data the same way as the other two.

Nevertheless, it is clear that the proportional hazard ratio for this program is significantly greater than 1, suggesting, again, that open source bugs for this program are fixed more quickly than comparable closed source bugs. Again, the average hazard ratio is still the best summary measure of the comparison between open and closed source. Shown in Table 4, the average hazard ratio for windows interfaces is much greater than 1 and significant.

The results are largely supportive of the hypothesis that the rate of quality improvement is higher for open source than closed source programs. For the open source operating system and user interface programs, the bug fix rates are significantly faster than for the corresponding closed source programs. In the web server case, the open and closed source bug fix rates are about the same. Why the web server is different from the other two programs is a matter of speculation at this point. One possible difference is that the closed source web server firm was a start-up, while the other two programs were written by large incumbent firms. Anecdotally, we hear about the agility and responsiveness of start-ups, where employees often work around the clock. But how start-ups differ from incumbent firms is not well characterized in the literature, and is most certainly not captured in the generic firm-level objective function used in my model to describe profit-maximizing firms generally.

7. Discussion

The model and empirical test are intended to provide a way of thinking about a puzzling business phenomenon. The paper starts out with the plight of venture capitalists in trying to identify profit-making opportunities. Business practitioners are similarly confused about how threatening open source software is. Microsoft manager Valloppillil (1998) in the so-called “Halloween document” sounds a warning to Microsoft’s executive management that open source is a potential competitive threat. Microsoft’s Bill Gates, in public statements, is unconcerned that open source software can meet the needs of Microsoft’s customers.

By viewing open source software as consumer integration into production, we can apply standard economic analysis to the problem; identify objective functions, compare different governance structures, make predictions. For example, Valloppillil and Gates might both be right: how serious a competitor open source programs are depends on the product and how demand for that product breaks down into the four groups described above. Similarly, companies providing service to nonprogramming users can be a profitable business, but it again depends on the same factors: how do users fall into the four categories?

Aside from providing a way to think about the particular business phenomenon of open source software, the model is interesting from other, theoretical perspectives.

7.1. The Role of Institutions

The very simple model presented above captures only the very basic structure of incentives and actors. Keeping the model simple makes the analysis as transparent as possible. But as important as what is in the model is what is not in the model. I have assumed, rather than modeled, the necessary underlying institutions that make open source software a viable alternative governance structure. Von Hippel (1988) observed that very often, firms innovate only after its customers have already conducted some of the initial stages of innovation themselves. The question is, how does open source differ from this user-driven innovation? What does it take for users to go from doing some innovation themselves but then passing the ball to suppliers to actually producing a good themselves? In other words, what institutions are necessary for users to go from one governance structure to another?

In the case of open source software, one very important facilitating institution is the Internet. We know that open source software and copylefting have been around for almost twenty years. Yet as a business phenomenon, open source has only gained commercial significance in the last three or four years. The Internet, and especially the world wide web, have made it less expensive for download software, upload code, communicate, etc. The Internet has also provided a meeting place for individuals wishing to use and contribute to open source projects.

Also, the principles and norms that help govern autonomous contributors, as described by Tuomi among others, are similarly important. The undeniable sense of community that open source contributors feel helps to organize open source activity, and to mitigate free-riding.

7.2. Understanding Nonprofit Organizations

Perhaps a less obvious application of the open source model is in understanding nonprofit organizations. Of course, some very high-profile projects have incorporated as nonprofit organizations, including The Apache Foundation, and the Free Software Foundation. But more importantly, the approach, of viewing open source as user integration into production, is the same one I use in a previous paper (2000), in which I examine the performing arts industry as an archetypal nonprofit-only industry. By applying this approach to the software industry, several interesting things arise.

First, the extreme case of zero marginal cost of production, which is rare in goods production but a reasonable approximation for software, shows that nonprofits can operate in goods industries. A long-held belief in the nonprofit literature is that nonprofits operate only in service industries. With open source software, nonprofits are now shown to compete in goods industries, and this model shows that it is *nonrivalry* that matters rather than whether the thing being produced and sold is a service or a good.

Second, with open source software, as with other cases of consumer integration into production, the total surplus is consumed by the producers, and may thus be unobservable. This unobserved consumption causes a great deal of confusion among practitioners and researchers. What appear to be voluntary contributions of effort to a public good make observers think that these “donations” are motivated by something other than self-serving utility-maximization. Open source philosophers speak of the mores and ethics of their community;

others attribute donations to the importance of reputation, staking out intellectual “territory”, and status (for example, see DiBona, et al). The Hermann et al survey suggests that, while these factors certainly may exist, they are not primary drivers of contributions, and are very unlikely to have driven the formation of open source projects. We know from the Apache case, for example, that the original founders made their initial source code improvements autonomously to solve their own individual problems.

Finally, if open source software is a nonprofit governance mode, then the software industry is a “mixed” industry, with nonprofits and for profits competing to provide a product. It turns out that nonprofits and for profits compete in many large and important industries, including health care, insurance, education, social services, R&D, and stock markets. Yet standard economic explanations for why nonprofits exist, let alone compete with for profits, are often unsatisfactory. Modeling a “mixed” industry generates some intuition about why nonprofits form and why they often compete with for-profits.

In the model, we see that different demand segments have different preferences for nonprofit production and for profit production. For example, the upper right group was the last to prefer open source software. These different thresholds might account for why nonprofits and for profits are able to operate in the same markets. Depending on demand, it might be possible that one or both of the high willingness to pay groups would continue to buy closed source software while the lower two groups made their own software. According to the model, we can predict that such co-existence might occur, and base our predictions on reasonable analysis of demand segmentation. The division of demand into high and low willingness to pay, on one dimension, and high and low ability to contribute (program) on the other dimension, is far from arbitrary. Less easily predictable (and not in the model) is the matter of whether supporting institutions exist for nonprofit formation. If users or consumers are unable to organize for their own production and somewhat overcome free-riding, nonprofits cannot form.

7.3. Intellectual Property

This study of open source software also raises some interesting questions about the role of intellectual property rights in the innovation process. The traditional view says that patent law encourages inventors to disclose their innovations and thus provide a public good. It has further been argued that the monopoly rights

conferred by a patent give inventors an incentive to invent, and that patent races can arise (e.g., Gilbert & Shapiro). Baba, et al argue that poor intellectual property protection for software has hindered the growth of the packaged software industry in Japan. Yet the use of intellectual property rights as an incentive to innovate varies by industry, suggesting that industry-specific attributes help determine IP-intensity. This study of open source software presents an example in which copyrights are used extensively in one segment of the industry but not in another segment. The copyright-intensive segment produces high levels of innovation, as might be expected. But the copyleft-intensive segment can produce even higher levels of innovation.

7.4. Application in Further Research

As mentioned before, there are striking similarities between the institutional support for open source software and that for academic research. The reliance of open source projects on peer review, status, and reputation, described by Tuomi, for example, are remarkably similar to the rewards and incentives used in university-based academic environments. Like open source software, university-based R&D is organized as a nonprofit and co-exists with for profit R&D performed in industrial firms. Applying a user-integration-into-production approach to R&D would be interesting because R&D is an activity of vital concern to policy makers as a source of economic growth and development (Nelson). In order to formulate effective economic policies and incentives, we need to better understand nonprofit production of R&D and how it relates to for profit production of R&D. Analyzing producers and consumers of R&D in the approach suggested by the model may be a simple and effective way to do this.

Another possible area to apply this type of model is in other Internet-related businesses. We know that the Internet was a facilitating factor in open source software formation, making the open source governance structure possible. Other business which the Internet has enabled include auction sites and exchanges. In the case of exchanges, new for profit, on-line only exchanges (ECNs or electronic communications networks) have arisen to challenge the monopoly previously enjoyed by the NASDAQ, an exchange run by the nonprofit NASD (National Association of Securities Dealers). Commodities exchanges, options exchanges, and even the staid, old New York Stock Exchange are among the many nonprofit organizations that have recently voted to demutualize (reorganize as for profit companies). Understanding the differences, advantages and

disadvantages, of competing governance structures can help clarify the competitive situation in this other, very important, growth-related industry.

8. Conclusion

This paper attempts to understand an emerging business model known as open source software; a business model of enormous and growing interest among practitioners as well as scholars. I present a model of user integration into production; users deciding to make or buy software. This model helps to clarify an important new source of competition in an innovative industry, and also expands our understanding of how and why nonprofits compete with for-profits.

I also test the model's prediction using a clean and simple metric for software quality. The analysis provides support for the hypothesis that open source software has a higher rate of quality improvement than closed source software, with the open source operating system and windows interface bugs being fixed significantly faster than the closed source bugs, and the open source web server bugs being fixed faster than the closed source program but not significantly so. The data also provide other interesting findings. First, the comparison between open and closed source software bugs suggests a tortoise and hare sort of relationship. Also, the data suggest that among hares, start-ups might be different (quicker) than more-established firms. While this is not an intuitively controversial observation, it is also not readily modeled in firm-level models such as the one I use.

Finally, by applying a consumer integration into production analysis to open source software, I would like to suggest that other emerging and confusing phenomena can be understood in similar straightforward fashion, especially when nonprofits are involved. Nonprofits have been difficult to deal with in standard economic analysis because it has been difficult to identify objective functions for nonprofits. Part of this is due to the observational problem that consumption of surplus is not observed, and so we attribute contributions to unpredictable, hard-to-model motives like altruism, rather than standard, easy-to-model motives like consumption. But if "nonprofits" are sometimes merely the result of a consumer's make-or-buy decision, comparing nonprofits to for profits becomes simple and tractable. I hope to suggest that open source software is not the only important and confusing phenomenon that can be understood by this straightforward approach. Rather, there are many others, like the age-old question of federal R&D policy, as well as new, Internet-driven phenomena like ECNs, can

be addressed in the same way.

References

- [1] Baba, Yasunori, Shinji Takai, and Yuji Mizuta, "The User-Driven Evolution of the Japanese Software Industry: The Case of Customized Software for Mainframes," in *The International Computer Software Industry*, ed. David C. Mowery, Oxford University Press, 1996.
- [2] Clapes, Anthony. *Softwars*, Quorum Books, Westport, CT, 1993.
- [3] DiBona, Chris, Sam Ockman and Mark Stone ed., "Open Sources: Voices from the Open Source Revolution," O'Reilly and Associates, Inc., 1999.
- [4] Gilbert, Richard and Carl Shapiro, "Optimal patent length and breadth," *Rand Journal*, vol. 21, Spring, 1990.
- [5] Goldstein, Paul, Copyright, 2nd Edition, Little, Brown, 1997.
- [6] Hamm, Steve, "The Wild and Woolly World of Linux," *Business Week*, November 15, 1999.
- [7] Heffan, Ira, "Copyleft: licensing Collaborative Works in the Digital Age," *Stanford Law Review*, 49, July, 1997.
- [8] Hermann, Stephanie and Guido Hertel and Sven Niedner, <http://www.psychologie.uni-kiel.de/linux-study/writeup.html>.
- [9] <http://apache.org/>
- [10] <http://www.bugs.apache.org/index>
- [11] <http://www.bugs.gnome.org/>
- [12] <http://www.freebsd.org/support.html#gnats>
- [13] Kafura, D. and G. R. Reddy, "The use of software complexity metrics in software maintenance," *IEEE Transactions Software Engineering*, 13 (3), 335-43.
- [14] Kiefer, Nicholas M., "Economic Duration Data and Hazard Functions," *Journal of Economic Literature*, 26, June, 1988, 646-679.

- [15] Klein, Benjamin and Keith B. Leffler, "The Role of Market Forces in Assuring Contractual Performance," *Journal of Political Economy*, 1981, vol. 89, no. 4.
- [16] Kogut, Bruce and Anca Turcanu, "The Emergence of E-Innovation: Insights from Open Source Software Development," November, 1999, unpublished.
- [17] Kuan, Jennifer, "The Phantom Profits of the Opera: Nonprofit Ownership in the Arts as a Make-Buy Decision," 2000, unpublished.
- [18] Lakhani, Karim and Eric von Hippel, "How Open Source software works: "Free" user-to-user assistance," May, 2000, MIT Sloan School of Management Working Paper #4117.
- [19] Lee, Gwendolyn K. and Robert E. Cole, "The Linux Kernel Development As A Model of Knowledge Creation," October, 2000, unpublished.
- [20] Lerner, Josh and Jean Tirole, "The Simple Economics of Open Source," March, 2000, NBER Working Paper 7600.
- [21] McCabe, T. J., "A complexity measure," *IEEE Transactions Software Engineering*, 2 (4), 308-320.
- [22] Merges, Robert P., "A Comparative Look at Intellectual Property Rights and the Software Industry," in *The International Computer Software Industry*, ed. David C. Mowery, Oxford University Press, 1996.
- [23] Nelson, Richard, "The Simple Economics of Basic Scientific Research," *Journal of Political Economy* 67, 1959, 297-306.
- [24] Red Herring Magazine, Feb. 1999.
- [25] Shapiro, Carl and Hal R. Varian, *Information Rules*, Harvard University Press, 1999.
- [26] Sommerville, Ian, *Software Engineering*, Fourth Edition, Addison-Wesley, 1992.
- [27] Tuomi, Ilkka, "Learning from Linux: Internet, innovation and the new economy," February, 2000, unpublished.

- [28] Valloppillil, V., “Open Source Software: A (New?) Development Methodology,” available at <http://www.opensource.org/halloween/halloween1.html>, 1998.
- [29] von Hippel, Eric, *The Sources of Innovation*, Oxford University Press, 1988.
- [30] Weber, Thomas E., “Here’s a Plan to End Microsoft’s Dominance (No Lawyers Needed), *Wall Street Journal*, May 15, 2000.