Open Source Software:
Free Provision of Complex Public Goods

July 2005

By James Bessen*

Abstract: Open source software, developed by volunteers, appears counter to the conventional wisdom that without ownership rights or government intervention, public goods will not be efficiently provided. But complexity makes a difference: contracts are incomplete and ownership rights do not necessarily elicit socially optimal effort. I consider three mechanisms that improve the provision of complex software: pre-packaging, Application Program Interfaces and Free/Open Source software (FOSS). FOSS extends the range of products available to consumers, complementing, rather than replacing, proprietary provision. Pre-packaged software addresses common uses with limited feature sets, while firms with specialized, more complex needs use FOSS.

JEL codes: H41, L22, L86

Keywords: Software, Contracting, Information Goods, Complexity

*Boston University School of Law and
Research on Innovation
202 High Head Rd.
Harpswell, ME 04079
jbessen@researchoninnovation.org

Thanks for helpful comments from Jürgen Bitzer, Jacques Cremer, Paul David, Raymond Deneckere, Karim Lakhani, Justin Johnson, Jean-Jacques Laffont, Lawrence Lessig, Mike Meurer, Philip Schröder, Jean Tirole, Eric von Hippel, Jason Woodard, Brian Wright and participants at seminars at Harvard, IDEI, and Stanford. All errors are the author's responsibility.

I. Introduction

On first examination, open source software seems paradoxical. Open source software is a public good provided by volunteers—the "source code" used to generate the programs is freely available, hence "open source." Networks of thousands of volunteers have developed widely used products such as the GNU/Linux operating system and the Apache web server. Moreover, these are highly complex products and they are, arguably, of better quality than competing commercial products, suggesting that open source provision may be highly efficient. This appears to counter the common economic intuition that private agents, without property rights, will not invest sufficient effort in the development of public goods because of free-rider externalities.

Much of the initial research exploring this apparent paradox has focused on the motivations of individual programmers (See Rossi 2004 for a survey). Lerner and Tirole (2002) attribute much individual motivation to reputation building and career concerns. Harhoff et al. (2000) consider other individual motivations. Johnson (2002) and Kuan (2001) model individual user/developers with common needs but with heterogeneous valuations and abilities. Survey evidence suggests that individuals have a wide variety of motivations for participating in open source development (Hars and Ou 2002, Krishnamurthy 2002, Ghosh et al. 2002, Hertel et al. 2003, Lakhani and Wolf 2005).

But it has become increasingly apparent that *firms*, and not just individuals, play an important role. Many large firms such as IBM, HP, Computer Associates and Novell have dedicated substantial resources to Free/Open Source software (FOSS) development. Although some of these firms may contribute for strategic reasons, the software plays no such strategic role for many firms, e.g., many of the firms contributing to Linux are not direct Microsoft competitors. Indeed, many firms in the embedded software business (software embedded in electronic devices) contribute code to Embedded Linux even though this is a core part of their business (Henkel and Tins, 2004). And surveys show that about half of the entire development effort on FOSS projects is performed by programmers at work with the knowledge of their supervisors (Lakhani and Wolf, 2005).[1] Although most of the contributors in the early years of Linux and Apache may have been volunteers, firms and their employees appear to play a major, if not dominant, role today. And

[1] Lakhani and Wolf find that 38% of programmers contribute to FOSS at work with the knowledge of their supervisors (another 17% contribute at work without the knowledge of their supervisors), but these programmers contribute about 50% more hours than others. Analyzing his survey results, Henkel estimates that about 90% of the effort on embedded Linux is made by programmers at work (private communication).

firms appear to have very different motivations than individuals to participate in FOSS (Bonaccorsi and Rossi 2004).

This paper asks whether there is a sound economic reason why firms contribute to FOSS development even when proprietary products are available from non-rival firms. This inquiry may reveal something about the limits of effective proprietary provision. Also, it may help explain why FOSS appears so robust in some markets, with FOSS products successfully challenging well-funded proprietary products in areas such as web servers (Apache), server operating systems (Linux) and embedded systems (embedded Linux). This success and the continued rapid growth of FOSS developers and projects[2] seem hard to reconcile with a movement based solely on programmers' reputations and similar personal motivations.

Several research papers have studied duopoly competition between a commercial software firm and a community of volunteers (Kuan 2001, Casadesus-Masanell and Ghemawat 2003, Mustonen 2003, Bitzer 2004, and Gaudeul 2005). But these papers assume that the programming is done by volunteer programmers, not by profit-seeking firms.[3] In contrast, my paper begins by looking at why firms might make major contributions to FOSS even when a commercial product is available.

Perhaps firms are motivated mainly by the personal motivations of their employees—that is, firms allow employees to participate in FOSS projects as a kind of fringe benefit. This explanation has problems, however. For one thing, this raises the question of why this would be a superior fringe benefit to traditional benefits and, if so, then why these kinds of fringe benefits have not been seen before. Also, this would seem inconsistent with some of the explanations of personal motivation. For example, if employees are motivated by job signaling, then a firm would hardly want its employees to signal their value to *other* prospective employers by participating in FOSS development on company time. Although personal motivations are surely for both hobbyists and employees, profit-driven firms may have good reason to contribute to FOSS development even when commercial substitutes exist.

Of course, firms might choose to support FOSS as a simple public good. But the general intuition from the literature is that privately provided public goods are under-provided, delayed, or

---

[2] SourceForge, one website where many FOSS project and participants register, 900,000 participants and 86,000 projects (8/2004) and continues to grow.

[3] In Bitzer (2004) a profit seeking firm distributes open source software, but does not develop it.

of inferior quality (Bliss and Nalebuff 1984, Palfrey and Rosenthal 1984, Johnson 2002).[4] Yet if a high quality, efficiently provided commercial software substitute is already available, one would expect gains from trade. Assuming that the marginal cost of software is small, the commercial software provider could charge something less than the customer firm's private value, but more than the anticipated contribution of the customer firm to the FOSS project. Then customer firms would purchase the commercial product instead of participating in FOSS and the commercial provider would make additional profits.

Another possibility is that firms participate in FOSS because the availability of free software increases sales of complementary hardware or services. Casual empirical evidence suggests that many companies participating in FOSS do sell complementary products. But here, too, a commercial software provider should be able to realize profits by selling an appropriately priced substitute, providing mutual gains from trade. This is true as long as the commercial software provider is willing to sell. However, if the commercial software provider also sells rival complementary products or services and is engaged in a strategy of predatory tying, then it may not offer to sell on favorable terms. This may explain a portion of FOSS activity, e.g., some companies supporting Linux have also sued Microsoft for antitrust violations. However, many of the firms contributing to Linux are not clear rivals to Microsoft, so this is at best a partial explanation for firm participation in FOSS.

Perhaps asymmetric information or transaction costs prevent trade from taking place between a commercial software provider and a firm considering contributing resources to FOSS. But this by itself is not an adequate explanation of the FOSS phenomenon. Transaction costs and asymmetric information have affected many markets for many years, yet few can be said to exhibit anything like FOSS. Instead, a variety of contractual mechanisms, price discrimination, etc. are seen as ameliorating transaction costs in many markets.

I argue that a particular feature of software makes contracting difficult, namely, complexity. Moreover, complexity shapes private mechanisms to counter this obstacle and it provides a reason for firms to contribute to FOSS.

Typical software products have large numbers of features and the number of possible interactions between these features can be astronomical. A "successful" software innovation tailors these interactions so that the software performs well for a customer's needs. The role of complexity has been recognized in the economics literature on contracting (Segal, 1999, Hart and

[4] Bitzer and Schröder (2005) model a public good game for open source software where the software is produced without delay, but this model is based on signaling benefits to individual programmers.

Moore, 1999, Tirole, 1999) and complex contingencies are described as a source of high transaction costs more generally.

As in other markets, a variety of mechanisms are used to facilitate transaction and overcome the difficulties imposed by complexity. I explore two of these in the context of my model: pre-packaged software, where a large number of software features are bundled together, and an application program interface (API), where a set of bundled features can be accessed by customized programs. In short, there is not a single form of proprietary provision of software, as is often assumed. Both my model and empirical evidence point to a variety of forms of provision.

In my model, FOSS is yet another mechanism that allows some user needs to be met with greater efficiency. In this sense, FOSS is not an alternative to proprietary development, but a complement; it *extends* the market. In this model, FOSS does not displace a pre-packaged software monopolist, although the monopolist will charge lower prices and will lose API sales. Instead, pre-packaged software is sold for simpler applications used by large numbers of customers. Firms with specialized needs and more complex applications use FOSS.

This highly stylized model appears to provide a simple explanation for several observed features of software markets: the emergence and growth of the pre-packaged software segment as the markets grew in size; the persistence of self-development and contract programming despite this growth; the development of APIs; the coexistence of proprietary and FOSS development with successful FOSS projects tending to more complex, "geekier" applications.

The apparent paradox about efficient provision of a public good posed above is resolved because software is not a simple public good; it is instead a *complex* public good, used in many different applications by highly heterogeneous users. A FOSS software product includes a variety of features used in a non-rivalrous, non-excludable way. At the same time, it may contain combinations of features that meet unique needs of individual consumers. This makes a difference to the efficiency of provision.

The next section considers the role of complexity and the difficulty of contracting for software development. Section 3 develops a formal model and applies it to pre-packaged software, and APIs, and FOSS. Section 4 concludes.

2. Background: Complex products

2.1 Complexity and contracting

I begin with a version of Aghion and Tirole's model of innovation (1994) adapted to a setting where customers for software innovations may also develop software. Aghion and Tirole's model concerns the problem of designing a contract or an allocation of ownership rights that elicits a socially efficient allocation of effort (or other unobservable investments) from the two parties. A key result is that proprietary provision—either through contracting or through a simple allocation of property rights—may be socially inefficient when a complete contract cannot be written for an innovation, that is, the innovation is *ex ante* indescribable.[5] Contracting over software generally has this difficulty: the software code itself is the complete description of how the software will function in every circumstance and, consequently, writing a contract that covers every contingency costs roughly the same as writing the software itself. Practical contracts for software will thus not completely specify all details, interactions and contingencies.

This difficulty arises from the complexity of software. As noted above, the literature on incomplete contracts has recognized the problem posed by complexity. Building on this line of thought, a key insight of this paper is that the structure of this complexity also gives rise to mechanisms that can overcome the inefficiencies of proprietary software provision, at least partially. During the early decades of the computer industry, almost all software was either developed by the customers themselves or developed under individual contracts, often with the computer manufacturer. A series of organizational innovations, however, have allowed the software industry to evolve more sophisticated (and presumably socially preferable) mechanisms to provide software.

The first innovation, pre-packaged software, works by combining a select group of features in a pre-programmed bundle. Multiple customers can then purchase this bundle in efficient arm's-length transactions. I show that this form of proprietary provision is socially efficient for those customer's whose needs are met by the features in the bundle. However, in sufficiently complex environments, many customers' needs will not be adequately addressed by pre-packaged software. Pre-packaged software firms can also produce customized versions for

[5] Other necessary conditions are that innovative effort is also non-contractible, *ex post* renegotiation cannot be ruled out and there is a single customer.

some of these customers under contract, but I show below that this does not generate socially efficient outcomes.

A second innovation that allows pre-packaged software firms to more efficiently address some specialized needs is the applications program interface (API). In a situation where a pre-packaged software product includes some, but not all, of the features that a customer needs, the software firm also sells the tools to access functions performed by the code. Using these functions, a customer can reduce the effort needed to develop the software themselves. The API does allow more efficient innovation for some customers, but some customers are priced out of the market at the software firm's optimal price for the API.

Free/Open Source software development provides yet additional social welfare gains. With FOSS, customers can use publicly shared source code instead of an API. This reduces their required development effort as above. Because the code is freely available, even firms priced out of the market for the API can develop more efficiently. Moreover, because these firms share their code in turn, the base of available code can grow far greater than the code available in commercial APIs, allowing far more complex applications to be developed efficiently.

2.2 The cost of complexity

Why are software programs complex? They typically include many features that work together to meet heterogeneous needs. Because consumers have different preferences for each feature in a complex product, they use different combinations of features. This differs from simple commodities because, in effect, the customer consumes only a specific instance (the "use-product") of the general product the firm sells. The firm sells a single product with a large number of features that may be optionally used with each other, making a large number of possible use-products of which only one may be of interest to any particular customer consumer.

This distinction creates a real economic difference when the software development firm faces a cost, even a slight cost, for each use-product. And indeed, the quality of software depends on the extent to which different use-product combinations are worked out, tested and debugged. Because the features in a complex software program interact with each other, each use-product must be individually tested to ensure that it works. Yet firms cannot feasibly test all possible use-

products because the number of possible combinations is astronomical.[6] For example, Cusumano and Selby (1995, p. 310) describe the complexity facing Microsoft's operating systems:

> The many testing approaches Microsoft uses to evaluate products prior to release are still insufficient to detect all errors due to the very large number of combinations of product usage scenarios that can occur. The various commands, data inputs, and underlying system configurations can cause a possibly infinite number of combinations. For example, assume the following: A systems product has 100 features. Each feature has 1 normal execution completion and 2 error messages it could generate. The product runs on 20 different vendors' disk drives. It should run in 4, 8, 16, or 32 megabyte (MB) of memory. The product should run with 15 different vendors' printers, and 5 different vendors' video cards. It should support 100 of the most popular applications and the 50 most frequently used commands for each of these applications. In order to simply begin testing this product, testers would have to set up a lab that could support over 9 billion combinations of usage scenarios (because 100 x 3 x 20 x 4 x 15 x 5 x 100 x 50 is equal to more than 9 billion). Even if such a lab were practical, it would not be cost-effective—and this list of combinations is incomplete by far.

This complexity has at least three consequences: it affects the way pre-packaged software companies develop their products, it causes pre-packaged software firms to limit the number of features in their products, and it drives up the cost of specifying customized contracts.

To limit costs associated with the interaction of features, software firms build products using "structured code" and "object-oriented" regimens that help reduce interactions and help locate incorrect interactions that cause bugs. Firms also use a wide variety of testing techniques, including automated testing (Cusumano and Selby, 1995, chapter 5). And they provide partially debugged code to limited groups of customers for "beta" testing (Cusumano and Selby, 1995, p. 310). Also, they do not test exhaustively; rather, products are released when bug discovery rates fall below a specified level.

Nevertheless, complexity insures that most of the cost of software arises from testing, debugging and customer maintenance (that is, fixing bugs or providing work-arounds after product release), not from the original design and coding. One study found that testing, debugging and maintenance account for 82% of the cost of software (Cusumano, 1991, p. 65). In 1995, Microsoft employed 3,900 engineers for testing and customer support (Cusumano and Selby, 1995, p. 51). Yet it only employed 1,850 software design engineers and these split their time between initial coding and debugging.

Complexity-related costs also limit the ability of packaged software to meet all consumer needs and some consumers turn to custom programming and self-development. In the 1950's and

---

[6] If a product had 100 independent features and each combination took only one second to test, then the job could not be finished before the sun is predicted to swallow the earth even if every human currently alive spent every second until then testing.

60's, owning a computer almost always meant either self-developing or contracting custom software development. Proprietary software consisted of limited applications that were almost entirely sold bundled with computer hardware. Little packaged software was sold until the 1970's (Grimm and Parker 2000), when IBM was challenged by private and government lawsuits to unbundle, and when mini-computers became widely used. As Figure 1 shows, pre-packaged software has grown, especially with the dramatic expansion of the computer market with low cost personal computers beginning in the 1980s.

This growth in market share has been accompanied by dramatic growth in product complexity. Competing software firms, attempting to reach ever-larger markets, engage in "feature wars," adding large numbers of new features to product revisions, encouraging upgrades and hoping to increase market share. The result is an intense pressure to add new features. This growing complexity is evident in five Microsoft product upgrades that occurred during the late 80's and early 90's (Cusumano and Selby, 1995, p. 224 and 246). The number of lines of source code in each product grew substantially from one version to the next, the increases ranging from 31% to 109%.

Yet despite this rapid growth in features and despite the implied rapid acceleration in debugging, testing and maintenance costs, pre-packaged software appears to have reached some significant limits—it has failed to account for as much as 30% of total software investment (Grimm and Parker 2000). Despite the common view that Microsoft is the prototypical software company, most software investment involves self-developed software or contract programming and neither Microsoft nor the other pre-packaged software companies have been able to adequately address the needs of a substantial portion of the market.

When standardized software packages fail to meet such specialized needs, users develop their own software or contract with someone else to develop it for them, as the figure shows. Frequently, a user does not need to code an entire program from scratch, but can utilize "developer's toolkits" for packaged software.

An alternative approach is for users to modify a Free/Open Source program. A key feature of FOSS is that the code is available for users to freely modify—this is the sense in which the code is "free," not so much that the software has nominal price of zero.[7] This feature is significant,

[7] As Richard Stallman famously says, Free Software is "free as in free speech, not as in free beer." Copyleft software requires modifications of the code to be shared if they are distributed, thus ensuring a dynamic code base. License agreements for "open source" software do not have this requirement, but nevertheless code modifications are frequently shared. This occurs for two reasons. First, strong community norms support free re-distribution—few programmers want to contribute code enhancements to projects that will be taken

because open source advocates claim that this provides a substantial advantage in developing complex software products of high quality (Raymond).

A brief look at one FOSS product, the Apache web server, illustrates the importance of specialized needs not addressed by competing pre-packaged software products. The Apache server competes directly with Microsoft's IIS server and other proprietary products (Microsoft provides this software bundled with some versions of its Windows operating system). But despite this competition, over 60% of active web sites use the Apache web server (Netcraft) and a major reason is its customizability. In a usage survey of Apache security features, Franke and von Hippel (2002) found that over 19% of the firms using Apache had modified the code for these features themselves and another 33% customized the product by incorporating add-on security modules available from third parties.[8] Open source code facilitates the provision of add-on modules and over 300 of these have been developed for Apache.[9]

Moreover, many of these private enhancements are shared with the community and incorporated in new versions of the product. During the first three years of Apache, 388 developers contributed 6,092 feature enhancements and fixed 695 bugs (Mockus et al, 2000). This rate of feature enhancement far exceeds the rate for comparable commercial products (Mockus et al, 2000, Table 1).[10] Thus the open source code permits customization, facilitates third-party add-ons, and allows a wide range of users to incorporate new features and fixes, all improving the ability of Apache to meet specific users' needs. The breadth and dynamism of this participation demonstrate

---

private. Second, because many open source projects improve rapidly over time, it is advantageous to have enhancements incorporated in the free code. This eliminates the cost of re-incorporating code changes each time a new version is released. Thus the sharing of modifications, bug fixes and enhancements is an important part of all open source development.

[8] Security features represent only a fraction of Apache's total feature set, so, presumably, the total extent of customization is even greater.

[9] Apache Module Registry, http://modules.apache.org/, accessed 5/2002 with duplicates and bad records eliminated. Open source facilitates add-on development because the source code is accessible and because user customization helps create new add-ons. Indeed, Apache seems to have a much more active group of add-on developers compared to Microsoft's web server (IIS), which lists only 11 companies producing add-ons. See Microsoft, "IIS-Enabled Products from Industry Partners," http://www.microsoft.com/windows2000/partners/iis.asp, accessed 5/2002.

[10] This dynamic process of improvement-by-user-modification also appears to raise the quality of open source software. Kuan (2001) found evidence that complex open source projects had more effective debugging. And Miller et al (1995) found that open source Unix operating systems were noticeably more reliable than their commercial counterparts, even though the latter had been in use much longer.

the degree to which open source software extends the market. The many firms who customize Apache represent consumers whose needs are largely not met by proprietary products.[11]

Thus the complexity of software imposes costs on the provision of pre-packaged software that causes firms to limit the feature sets of this software, limiting the extent to which pre-packaged software addresses all users' needs. These needs can be further addressed by other organizational arrangements including APIs and FOSS. But clearly, a realistic appraisal of different modes of software provision has to account for a richer set of relationships between developer and customer than is the case for simple standardized commodities.

Complexity affects the relationship between developer and customer as well, however. In the next section I present a highly stylized model of this interaction.

3. Model

3.1 Contracting complex software development

Consider a case where a customer firm wants to contract with a commercial software development firm to write some customized software. Such contracting is notoriously difficult. Suppose that out of M possible features, the customer can identify m features likely to be important in this custom application. Even so, there are still a very large number of interactions between all of these features and it may (or may not) make a big difference to the customer how each of these interactions is coded. For example, if a word processing program determines that all of the characters in a line on the screen will not fit on a line when output on a given printer with a given font, should the program break the line before the last word, hyphenate the last word, squeeze the words together, squeeze the letters together, or do something else? Such details may determine whether the program works successfully for the customer, yet it is unlikely that the customer will be able to specify all such details in a contract.

Although the number of "features" in a product may be difficult to enumerate, I wish to capture the notion that the number of combinations of features grows exponentially with the

[11] Note that very little of this customization effort can be attributed to firms attempting to economize by using a free product and then correcting deficiencies through customization. The second most popular web server, Microsoft's IIS, is free for users of the Windows operating system. Apache runs on Linux (free), on proprietary Unix and also on Windows. If one assumes that these operating systems are equivalent for running web servers, then Apache offers no direct cost saving relative to IIS. Even if Linux were inferior to Windows, but could be fixed through customization of Apache, the cost difference would be minor—the price of Windows is $300 or less per license. Thus few firms would plausibly customize Apache to compensate for major deficiencies in Linux.

number of features. If we assume for the sake of concreteness that each feature can interact with all other features in just two ways, then there are 2^m such interactions and it will be very costly to write them all into a contract. Moreover, much of the work of figuring out these details is what the customer pays the developer for. That is, typically the developer gets a general idea of how the customer wants the program to work, then makes educated guesses about how the interactions should be coded, and allows the customer to review and modify these decisions.

The situation can be interpreted as an instance of indescribability. In Hart and Moore's (1999) terminology, the customer needs a "special widget," but is unable to specify in advance exactly which widget it needs among a large number of widgets.

To model this interaction, I start with a version of Aghion and Tirole's (1994) model of innovation. This model concerns two agents—firms in this case—the "developer," who obtains no use value from this software, and the "customer," who obtains value *V* from successfully developed software. This value is known to both parties, both are risk neutral and I initially assume that the developer cannot sell the software to another party other than the customer.

The developer and the customer can exert development efforts *e* and *E*, respectively. That is, *e* is outsourced development effort and *E* is inhouse development effort. These levels of effort are not directly contractible so there is moral hazard. To simplify things, I assume that both agents have an equal ability to develop software, that is, the customer is a "user-developer." Clearly, differences in programming competencies can affect contracting and also decisions regarding FOSS participation (see, for instance, Kuan 2001). However, the focus here is on contractual mechanisms, ownership and organizational arrangements, so this simplification aids the exposition.

The expended effort increases the probability, *p*, that the software will be successful (that is, delivering value *V* to the customer). Since both agents have equivalent programming competencies, I assume that *e* and *E* are perfect substitutes and $p = p(e + E)$. I follow Aghion and Tirole and assume that *p* is increasing and concave and $p(0)=0$. Also, to guarantee an interior solution, $p'(0)$ is infinite and $p < 1$.

The innovation or development project is *ex ante* partially indescribable; it cannot be written in a contract that can be enforced by a court or other third party. The problem is to design a contract or an allocation of property rights that elicits a socially optimal level of *e* and *E*. An inefficient contract or rights allocation will fail to reach these levels, resulting in a level of innovation that is less than the social optimum.

The interaction takes place in three stages: 1.) the parties draw up a contract that possibly specifies a license fee and ownership rights, 2.) the parties invest *e* and *E*. 3.) If the development effort is successful (which customers can determine costlessly), then the parties may choose to renegotiate the license fee (which cannot be ruled out in the contract). Following Aghion and Tirole and the bargaining literature, I assume that in this case, the parties split the bargaining surplus equally. Note that the initial license fee is ultimately replaced by the "real" license fee negotiated in stage 3. Following Aghion and Tirole, the agents are given only one chance to make a successful innovation (e.g., the customer cannot contract for development and then, if that fails, develop herself) and there is no time discounting.

Given licensing fee *y* that is paid only in the event of success, the developer maximizes utility $p(e+E)\,y - e$ and the customer maximizes utility $p(e+E)(V-y) - E$.

Consider contracts that assign ownership of the code to one party or the other. These are shown in Figure 2. First, if the customer owns the code, then under the optimal contract all development will be performed by the customer ($e = 0$), that is, this describes the case of "self-development." The customer exerts effort E^* which maximizes its utility:

(1) $$E^*: \quad \max_E \left[p(E)V - E \right].$$

This is a socially efficient level of effort.

On the other hand, if the developer owns the rights to the code, then, if the development effort is successful, the parties bargain in stage 3 and each receives $V/2$. Considering this, in stage 2, the Nash equilibrium is one where each party maximizes their utility holding the other party's effort fixed:

(2) $$\max_e \left[p(e+E)\frac{V}{2} - e \right] \text{ and } \quad \max_E \left[p(e+E)\frac{V}{2} - E \right].$$

Straightforward calculation shows that this form of proprietary provision yields a level of effort below the social optimum. Moreover, the customer makes out better under self-development.

So in this simple case, customers will choose to self-develop rather than to contract and self-development is socially efficient, but contracting is not. Of course, firms do contract for software development, but this model suggests that something other than simple contractual issues may be at play, e.g., the customer firm may have inferior competency at software development. Nevertheless, this highly stylized model highlights the difficulty of contracting for software development, setting the stage for consideration of mechanisms to overcome this obstacle.

Note that in this setting I have assumed that the customer can choose the initial assignment of ownership rights. This would be the case with copyright and trade secrecy protection, but not necessarily with patents. The developer could unilaterally patent an essential software concept, guaranteeing ownership in stage one prior to development of the code.[12] Then the developer could insist on contracting, resulting in socially inefficient proprietary development. That is, in this setting, patent rights may actually decrease innovation. In what follows, I develop the model assuming only copyright and trade secrecy rights, but I comment on effects of patents below.

3.2 Pre-packaged software

The social inefficiency of contracting (or of an initial allocation of property rights to the developer with no contract) may be seen as an instance of a more general result obtained by Aghion and Tirole. They point out that this result depends on the indispensability of the customer, that is, there is only one customer for this product. If there were more customers, then two things might improve the incentives for the developer: first, the developer may gain bargaining power in the stage three negotiation. In typical bargaining models with an outside option, the ability of the developer to stop bargaining with one prospective customer and switch to another gives the developer leverage. Second, if the developer can sell the same code to multiple customers (i.e., they are not competitors wanting exclusivity), then the incentives for proprietary development may be larger than for self-development (ignoring the possibility of self-developers also selling a proprietary product).

So an institution that allowed multiple customers for a single development effort might yield greater social welfare. One might suppose, in fact, that pre-packaged software corresponds to just this case. But there is a potential logical inconsistency with such an interpretation. How can a developer know that there are, say, *N* customers for a specific product? If a developer cannot know the "special widget" customer A needs, how can the developer know that customers B and C also need the same special widget? This seems implausible. Even if the developer knows that A, B, and C are in the same general line of business, this does not mean that all aspects of the software will be identical; each firm will likely have highly idiosyncratic needs.

On the other hand, it does seem that the developer might be able to obtain *some* knowledge about demand for certain groups of features without necessarily knowing *ex ante* all of the specific details needed to satisfy that group of customers. For instance, a developer might guess that color

---

[12] I assume here that a patent can be obtained without actual development of the code, consistent with court recent decisions on enablement.

printers might be useful features to add to computer systems (with the associated software support) without necessarily knowing all the details of how color printers might interact with other components of the software system. Moreover, a developer might plausibly build a prototype and test market the general appeal of such a system, again, without necessarily working out all the details in advance. Thus the developer can know the expected demand for a product with *m* features, but this product can be used in 2^m different ways—that is, it has 2^m different use-products—and the developer does not know the demand or the customer identities for each of these. This is, of course, a highly stylized treatment. In reality, developers are likely to have some information that some combinations of features may be more highly desired. Nevertheless, this captures the difficulty that firms have in managing complex interactions between features, as described in the Microsoft example above.

I model this limited knowledge formally as follows: a developer knows *ex ante* the expected number of customers, *N*, who will want a use-product involving the first *m* features (ranked in order of demand) of the *M* possible features. The developer does not know *ex ante* the specific use-product each of these customers wants; the developer just knows the total expected number of customers wanting use-products in the group using those *m* features. I assume that $N = N(m)$ includes the number of customers who want combinations of anywhere from 0 to *m* of the first *m* features. This is thus an increasing function and, since the features are ranked in order of their popularity, there are diminishing returns relative to the number of use-products, that is, $\frac{d^2 N}{d u^2} < 0,\ u \equiv 2^m$.

Pre-packaged software can then be interpreted as a strategy to develop *all* of the use-products created by combinations of the first *m* features. This can be compared to self-development in a revised version of the model. In this version, I change the stage timing slightly. Now, in the first stage (see Figure 3), the software developer decides a level of effort, *e*, and a number of package features, *m*. In the second stage, prospective customers observe whether the package successfully addresses their use-product and they decide to exert a level of effort, *E*, at self-development. If self-development fails, then customers still have the option to purchase the product at price *w* in stage 3. I also explored a model where the customer's decision to self-develop is made simultaneously with the developer's allocation decision in stage one. As long as the market is large enough, I find the main results of that model to be the same as the results of this model. Since this model is simpler to develop and because it provides a slightly more challenging problem for the software developer, I use this version in the exposition.

To complete the model, I specify several other details. Customers derive varying levels of utility, V_i, from their respective use-products. Let V_i be distributed according to a distribution function $F(\cdot)$ over support $0 < V_i \leq \overline{V}$ without gaps. As before, each customer receives zero utility from other use-products. I assume that V_i is uncorrelated with the number of features the customer desires. If these were correlated, then the firm could engage in "versioning," providing different versions of the software package (with some features disabled) to some customers at a different price. For simplicity, I leave out such considerations.

Of course, the more features included in a product, the greater the development effort required. To capture this notion, interpret *e* (or *E*) as the *intensity* of effort expended over a given number of tasks. Let the number of tasks increase with the number of use-products supported. Thus the developer's disutility for developing a product with *m* features is $e \cdot \left(c_0 + c\,2^m\right)$ where c_0 represents the tasks associated with initially coding the product, including the initial coding of each feature, and *c* represents the tasks associated with debugging and maintaining each use-product. In a more realistic model, the disutility of initial coding might increase with the number of features, *m*. Adding this source of variation makes the model more complicated without, however, affecting the basic insights; keeping c_0 constant captures the basic intuition that debugging and maintenance costs will substantially exceed the costs of initial coding. Given this assumption, a customer developing just a single use-product faces a disutility of $E \cdot \left(c_0 + c\right)$.

I assume that only a single firm develops a software package for the given market. With Bertrand competition, no rival would choose to enter the market with the same package of features. Competing firms might choose to develop different bundles of features, perhaps overlapping the first firm's feature set to some extent. This may give rise to a "features war," but such considerations are beyond the scope of this paper.

Given this, the firm will set a package price, *w*, in the third stage by solving a monopoly pricing problem, assuming zero marginal cost to reproduce software and assuming that the firm cannot price discriminate. Let *G*(*w*) represent the share of prospective customers for the pre-packaged product for whom $V_i > w$ *and* who have not successfully self-developed in stage 2. Then the firm's optimal price is

$$\hat{w} = \arg\max_{w} \left[w \cdot G(w) \right]. \tag{3}$$

Note that the firm cannot credibly commit to a different price in stage two.

The function *G* depends on the distribution *F* and the customers' decisions about self-development in stage two. If $V_i \le w$ or if the pre-packaged software does not successfully address the customer's application, then a customer has no alternative but to self-develop with expected net utility of

$$\pi^{C}_{self}(V_i) = \max_{E}\left[p(E)V_i - E\cdot(c_0 + c)\right]. \tag{4}$$

If, on the other hand, $V_i > w$ and the package does address the customer's application, the customer will still choose to exert some effort at self-development, and then purchase in stage three if this is not successful. In this case, the customer's optimal effort is

$$\begin{aligned}\hat{E}(w) &= \arg\max_{E}\left[p(E)V_i + (1-p(E))(V_i - w) - E\cdot(c_0 + c)\right] \\ &= \arg\max_{E}\left[p(E)\,w - E\cdot(c_0 + c)\right], \qquad V_i > w\end{aligned} \tag{5}$$

Note that this effort is independent of the customer's individual valuation as long as $V_i > w$. This means that

$$G(w) = (1-F(w))\left(1-p\left(\hat{E}(w)\right)\right). \tag{6}$$

It is then straightforward to show that $0 < \hat{w} < \overline{V}$. Also, note that $\hat{E}$ is strictly positive (although it may be quite small if c_0 is large), so that all customers exert at least some minimal effort at self development (recall the assumption that $p'(0) = \infty$).

The pre-packaged software firm then maximizes expected profits:

$$\pi^{D}_{pkg} = \max_{e,m}\left[p(e)\,N(m)\,\hat{w}\,G(\hat{w}) - e\cdot\left(c_0 + c\,2^{m}\right)\right] \tag{7}$$

where *e** and *m** are the level of effort and the number of features at the maximum. It is straightforward to show that this profit function is concave, however, *m** may take a corner solution, $m^* = M$ or an interior solution, $0 < m^* < M$. Also, *e**>0.

There are two main reasons why a customer might choose *not* to purchase the software package: because the price is too high ($V_i < \hat{w}$), or because the product is too simple, ($m^* < m$).[13] If a customer firm does purchase the packaged product, then it receives greater utility

[13] Also, it may happen that a customer who can afford the package happens to be successful self-developing in stage two. Note, however, that if c_0 is large, then this is unlikely (that is, $p\left(\hat{E}(\hat{w})\right)$ will be small) and, as noted, customers may exert zero effort if *p'* is finite.

than from self-development from scratch, so in this case, efficiency is enhanced. From this it follows:

> Proposition 1. Pre-packaged software improves social allocation of effort over what can be achieved by self-development or contract programming. However, this is not true for all potential customers. In particular, low value customers (low V_i) and customers with complex applications (high *m*) will choose self-development over purchasing pre-packaged software.

Remark 1: By taking the implicit derivative of (7), it is easy to show that *m** increases with the magnitude of *N*. This provides a simple explanation as to why pre-packaged software was not widely used during the early decades of computing and why the market share of pre-packaged software has been associated with the tremendous growth in the overall size of the software market accompanying low cost personal computers.

Remark 2: This kind of bundling of features into a single product to serve heterogeneous customers occurs, of course, with all sorts of other goods. For example, automobiles are sold with many options effectively built in to their production (although not necessarily included in the version each customer purchases). Two characteristics may make this issue particularly important for software. First, software products tend to be quite complex, that is, *M* is quite large and likely to exceed *m**. Second, trade secrecy of source code means that customers of pre-packaged software are not free to modify the product. This is not so for many physical goods, e.g., cars can be modified without special rights from the manufacturer. To the extent that other goods share these characteristics with software, much of the analysis applies to these goods as well as to software.

Remark 3. It may seem counter-intuitive that low value customers (low V_i) will self-develop. This is a consequence of the properties of *p* chosen to insure an interior solution. In a more realistic setting, low value customers will have a corner solution where they do not self-develop.

3.3 Proprietary extensions

Once a pre-packaged software product is successfully established, then secondary possibilities for greater social efficiency arise. This is because software code developed for the pre-packaged product can be reused. That is, the monopolist has already coded a basic product associated with effort $e^* \cdot c_0$. If this code can be reused, then the development effort necessary for a custom application that incorporates some or all of these coded features is less. In this new

setting, assuming that a successful pre-packaged software product already exists, let the additional disutility of effort needed for a custom application be only $e \cdot c_{ext}$ or $E \cdot c_{ext}$, depending on who performs the work. The socially optimal level of effort to customize a use-product not included in the software package is

(8) $$E^*: \quad \arg\max_E \left[p(E)V_i - E \cdot c_{ext} \right].$$

This is larger than the effort exerted under self-development from scratch in (4) as long as $c_{ext} < c_0 + c$, which one would expect.

Two proprietary forms of provision allow a pre-packaged software monopolist to reuse code: the monopolist might contract individually with individual customers in addition to providing a software package, and the monopolist might offer a developer's toolkit supporting an API. Both of these devices allow the monopolist to better address the needs of those prospective customers whose needs are too complex for the pre-packaged product.

Consider individual contracts first. As before, the specification of the use-product is *ex ante* indescribable and, as a result, the parties split the bargaining surplus in stage 3. The monopolist's effort toward a custom project utilizing this base code is the value of *e* that maximizes

(9) $$p(e)\frac{V_i}{2} - e \cdot c_{ext}.$$

By comparison with (4), the monopolist's effort under a custom contract will exceed the customer's effort at self development as long as $c_0 + c > 2c_{ext}$, although this is still less than the socially optimal level of effort. Moreover, the customer's expected utility under a custom contract, $p(e)\frac{V_i}{2}$, may be greater or less than the customer's utility from self development given in (4). Consequently, the customer may or may not choose to contract with the monopolist.

Another approach is for the monopolist to sell a developer's toolkit to access an application program interface (API). Let the monopolist's price for the toolkit be w_{API}. The customer will then realize a gross profit

(10) $$\pi^C_{API}(V_i) = \max_E \left[p(E)V_i - E \cdot c_{ext} \right]$$

and a net profit of $\pi^C_{API}(V_i) - w_{API}$. The monopolist will choose a revenue maximizing price, $\hat{w}_{API}$, and at this price, some customers will purchase the developer's toolkit and some will self-develop from scratch. Those customers who do choose to purchase the toolkit will exert a socially

optimal level of effort that exceeds the effort of the monopolist under a custom contract or of the customer developing entirely from scratch. Thus,

> Proposition 2. For prospective customers whose applications are too complex to be handled by a pre-packaged software product, a software monopolist can offer a custom programming contract or a developer's toolkit with associated API. These alternatives provide some of these customers a more profitable and socially efficient alternative to developing the software themselves from scratch. However, not all such prospective customers will be able to profitably take advantage of these alternatives.

3.4 Free/Open Source Software

These extensions to pre-packaged software work because they permit the reuse of base code incorporated in the packaged product, represented by c_0. Either the monopolist, in the case of contract programming, or the customer, in the case using the API, is required to exert less effort than if the customer were developing their software from scratch.

Once a Free/Open Source project is established, it, too, has a core of code that can be used by firms seeking to build customized solutions. It is like an API, but one available at a price of zero (and some important strings attached). Suppose, for example, that a FOSS project has coded the same m^* features as in the pre-packaged software product. Then $\pi^C_{FOSS}(V_i) = \pi^C_{API}(V_i)$, but with no additional charge to the customer as with the API. Clearly, in this case, no customer is priced out of the market, so the efficiency gains of the API are available to all customers with complex applications. FOSS thus further improves the provision of software.

More generally, suppose a FOSS project has coded $\tilde{m}$ features. Suppose that a prospective customer (prospective FOSS developer) needs just one additional feature. Let the disutility for that customer to code and debug one additional feature to the FOSS product be $E \cdot c_{FOSS}$, given intensity of effort, *E*. Then

$$\pi^C_{FOSS}(V_i) = \max_E \left[p(E)V_i - E \cdot c_{FOSS} \right]. \tag{11}$$

Note that that for some customers, $c_{FOSS} \leq c_{ext}$. This will surely be true if $m > \tilde{m} \geq m^*$ and the features in the pre-packaged software are a subset of the features in the FOSS code. This group of customers will make greater effort and have a greater probability of success under FOSS than with the API. With a sufficiently large code base, FOSS will be superior for customers with complex requirements, $m > m^*$,

> Proposition 3. Given a Free/Open Source software project that has developed a code base $\tilde{m} \geq m^*$ that includes the features of the pre-packaged software, complex applications with

$m > m^*$, will be developed with greater socially efficiency under Free/Open Source development than by a combination of a pre-packaged software product and custom programming, by a combination of a pre-packaged software product and a developer's toolkit, or by customer self-development.

3.5 Growth and viability of FOSS

This, of course, begs the question of whether and how a FOSS project can build such an initial code base. The main focus of this paper is on the existence and robustness of FOSS once it has begun. Still, the model suggests several points about its initial growth. First, FOSS development does not require a large initial code base. Consider a FOSS project where, say, $\tilde{m} << m^*$. As long as some prospective customer can make use of a product with $\tilde{m}+1$ features, then this code base allows this customer to efficiently develop by coding just one additional feature. And since that customer then returns the code for the additional feature to the shared resource, the code base for other prospective customers grows by one feature. Another FOSS developer may then add another additional feature, and so on. FOSS development can thus begin with a code base that is much smaller than m^*, and yet that code base can grow to something much larger than m^*, ultimately allowing much more efficient development of complex applications.

Note, however, that this argument assumes that no pre-packaged software product is on the market. With a pre-packaged software product on the market, some prospective FOSS developers may choose to purchase the package rather than to participate in FOSS development. There are at least three reasons, however, why the alternative of a pre-packaged software product might not prevent the initiation and evolution of a robust FOSS alternative:

1. Some customers will find the monopoly price for the software package too high. As I show below, a monopolist will charge a lower price for a pre-packaged software product when faced with a competing FOSS project. Nevertheless, some prospective customers will be priced out of the market and will choose to participate in FOSS development instead.

2. FOSS may have an advantage in small markets and/or markets where the initial development effort required to create a useful product is not large. As discussed below, pre-packaged software may not be sustainable in such markets. Moreover, it is often the case that the initial market for a new technology begins quite small and can be addressed with simple products, but as the technology improves the market grows rapidly. This growth is a staple of industry life-cycle studies and studies of "disruptive technologies" (Utterback, 1996, Christensen, 1997). In these cases, FOSS may get started earlier in the product life-cycle and may become well-developed

before proprietary competitors enter. This describes, for example, the development of web browsers and web servers. In effect, the historical path of development may provide an advantage to FOSS development.

3. Especially for small markets and small projects, the personal motivations of individual programmers may come into play and provide additional incentive to develop a FOSS project even though a commercial alternative is available. An example of this is Linux, which began as a personal project of Linus Torvalds using as a code base Minix, written by Andrew Tanenbaum as a teaching aid (Moody, 2001).

Two other factors might work against the early stages of FOSS development: if a prospective customer expects that other prospective customers might volunteer to code an initial project, then free-riding may give rise to some inefficiency as in Johnson (2002). Nevertheless, free-riding does not prevent the possibility of successful FOSS projects; it just diminishes the probability of success.

Second, software patents, especially obvious software patents, may prevent FOSS development. As noted above, software patents that cover a generic concept may limit the range of feasible contractual arrangements and such is the case here (FOSS *is* a form contracting after all). Note, however, that such generic patents are also a problem for pre-packaged software producers (albeit they may have more resources to litigate them), and forms of insurance are emerging for open source.[14]

In general, then, there is good reason to anticipate the emergence and growth of future FOSS projects. Moreover, these projects can grow even in the presence of a competing pre-packaged software product.

3.6 Coexistence

But will a successful FOSS project drive a pre-packaged software product from the market? Not necessarily.

First, note that even when a relatively sophisticated FOSS project is developed, many prospective customers may still prefer to purchase a competing pre-packaged software product. Even though, say, $\tilde{m} \geq m^*$, this does *not* mean that the FOSS project has developed support for all 2^{m^*} use-products supported by the pre-packaged software firm. So these customers (quite

[14] See Open Source Risk Management, http://www.osriskmanagement.com/.

possibly the vast majority of prospective customers) are faced with a choice of purchasing the pre-packaged product or customizing the FOSS code at a disutility of $E \cdot c_{FOSS}$.

Following the treatment for self-development above, the market for the pre-packaged product in stage three includes those customers who can afford price *w*, whose needs are met by the pre-packaged product, and who have not self-developed successfully in stage 2. These prospective customers will exert the following effort in stage two:

(12) $$\tilde{E}(w) = \arg\max_{E}\left[p(E)V_i + (1-p(E))(V_i - w) - E \cdot c_{FOSS}\right], \qquad V_i > w$$

Then the monopolist's optimal price is

(13) $$\tilde{w} = \arg\max_{w}\left[w \cdot H(w)\right], \qquad H(w) = (1-F(w))\left(1-p\left(\tilde{E}(w)\right)\right)$$

and maximum profit is

(14) $$\pi^{D}_{pkg} = \max_{e,m}\left[p(e)\,N(m)\,\tilde{w}\,H(\tilde{w}) - e \cdot \left(c_0 + c\,2^{m}\right)\right]$$

It is then straightforward to show that the monopolist will exert a positive effort, so,

> Proposition 4. A pre-packaged software monopolist and a FOSS project can coexist, both exerting positive effort.
>
> Proof: The partial derivative of the expression within brackets with respect to *e* is positive and infinite at *e*=0.[15]

As in the case where the alternative is self-development, two groups of customers will choose FOSS over the pre-packaged product: those with simple needs ($m \le m^*$) but low valuations who are unwilling to pay $\tilde{w}$, and those with complex needs, $m > m^*$. Since the first group is composed of low valuation customers, they will exert a relatively low level of effort *E** on FOSS development. The second group will exert greater effort. In effect, FOSS development will be concentrated on relatively complex applications.

The monopolist will charge a lower price in this scenario than in (7) because $\hat{E}(w) < \tilde{E}(w)$ since $c_0 + c > c_{FOSS}$. The corresponding values of *e** and *m** will be lower. So there will be some reduction in the probability of success of the pre-packaged software. This,

[15] This result depends on the assumption that $p'(0)$ is infinite. If $p'(0)$ were, instead, finite, then the same result would still hold as long as *N* is sufficiently large, that is, in a large market.

however, may be more than offset by the welfare gains of those customers who choose to participate in FOSS development.

Note that the model implies that FOSS will displace the use of an API in conjunction with a pre-packaged software product. This is clearly counterfactual, but it is a consequence of the assumption that customers are capable software developers. In a more general model, such customers (or programmers under contract with them) might purchase the API.

4. Conclusion

This analysis may help dispel two myths about Free/Open Source software. First, it is not a "communistic," "property destroying" alternative to proprietary software. It is better viewed as a complement to proprietary provision, recognizing that proprietary provision fails to effectively meet the needs of many customers in markets where customers have highly disparate needs and products are complex. Free/Open Source software and proprietary provision of pre-packaged software can both exist in a market, recognizing that they mainly serve different groups of customers. Free/Open Source will be most used by firms who have their own development capability and who have complex, specialized needs; pre-packaged software will be used by firms with simpler needs and by firms who lack development capabilities. The addition of Free/Open Source software to a market with a pre-existing packaged software product may reduce the monopolist's profits and may limit the monopolist's market for developer's toolkits, but this should not drive the monopolist out of the market.

Second, it is a mistake to assume that FOSS is somehow less robust because it is based on voluntary contributions rather than driven by the profit incentive. In fact, the firms that participate in FOSS are driven by the profit incentive—FOSS is just the most socially efficient means for many of them to obtain the software they need in their profit-making activities. Managers need to view FOSS as an alternative to simple "make-or-buy." This alternative will make the most sense for firms with specialized and complex needs. And it may be especially important in emerging technologies where markets are initially small.

References

Aghion, P. and J. Tirole. 1994. The Management of Innovation, Quarterly Journal of Economics, 109 pp. 1185-1209.

Bitzer, J. 2004. Commercial versus open source software: the role of product heterogeneity in competition, Economic Systems 28 pp. 369–381.

Bitzer, J. and P. Schröder. 2005. Bug-fixing and code-writing: The private provision of open source software, Information Economics and Policy 17 pp. 389-406.

Bliss, C., and B. Nalebuff. 1984. Dragon-slaying and ballroom dancing: the private supply of a public good, Journal of Public Economics 25, pp. 1–12.

Bonaccorsi, A. and C. Rossi. 2004. Altruistic individuals, selfish firms? The structure of motivation in Open Source software, First Monday, 9, no. 1 [http://firstmonday.org/issues/issue9_1/bonaccorsi/index.html]

Casadesus-Masanell, R., Ghemawat, P., 2003. Dynamic mixed duopoly: a model motivated by Linux vs. Windows. Harvard Business School, Working Paper No. 04–012.

Christensen, C. M. 1997. The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail, Harvard Business School Press, Cambridge.

Cusumano, M. A. 1991. Japan's Software Factories: A Challenge to U.S. Management. Oxford University Press, New York.

Cusumano, M. A. and R. W. Selby. 1995. Microsoft Secrets: How the world's most powerful software company creates technology, shapes markets and manages people. Simon and Schuster, New York.

Franke, N. and E. von Hippel, 2003. Satisfying Heterogeneous User Needs via Innovation Toolkits: The Case of Apache Security Software, Research Policy, 32, no. 7, pp. 1199-1215.

Gaudeul, A. 2005. Competition between open-source and proprietary organizations Working Paper.

Ghosh, R., R. Glott, B. Krieger, and G. Robles, 2002. Survey of Developers, Free/Libre and Open Source Software: Survey and Study, FLOSS, Final Report, International Institute of Infonomics, [http://floss.infonomics.nl/report/FLOSS_Final4.pdf]

Grimm, Bruce and Robert Parker. 2000. Recognition of Business and Government Expenditures for Software as Investment: Methodology and Quantitative Impacts, 1959-98, Bureau of Economic Analysis, mimeo.

Harhoff, D., J. Henkel and E. von Hippel. 2003. Profiting from voluntary information spillovers: How users benefit by freely revealing their innovations,Research Policy, 32, no. 10, pp. 1753-69.

Hars, A. and S. Ou, 2002. Working for Free? Motivations of participating in Open Source projects, International Journal of Electronic Commerce, 6, pp. 25–39.

Hart, O. and J. Moore. 1999. Foundations of Incomplete Contracts, Review of Economic Studies, 66, pp. 115-138.

Henkel, J. and M. Tins. 2004. Munich/MIT Survey: Development of Embedded Linux, working paper [http://opensource.mit.edu/papers/henkeltins.pdf].

Hertel, G., S. Niedner, and S. Hermann, 2003. Motivation of software developers in the Open Source projects: An Internet-based survey of contributors to the Linux kernel, Research Policy, 32, no. 7, pp. 1159–1177.

The Internet Operating System Counter Page. http://www.leb.net/hzo/ioscount/index.html. Accessed 2/2001.

Johnson, J. P.. 2002. Open Source Software: Private Provision of a Public Good,Journal of Economics and Management Strategy, 11, no. 4, pp. 637-62.

Krishnamurthy, S. 2002. Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects, First Monday, 7, no. 6 (June 2002), [http://firstmonday.org/issues/issue7_6/krishnamurthy/index.html]

Kuan, J. 2001, Open Source Software as Consumer Integration into Production, working paper [http://papers.ssrn.com/sol3/papers.cfm?abstract_id=259648].

Lakhani, K. and R. G. Wolf. 2005. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects, In: Feller, J., B. Fitzgerald, S. Hissam, K. Lakhani (eds.), Perspectives on Free and Open Source Software, MIT Press, Cambridge.

Lerner, J. and J. Tirole. 2002. Some Simple Economics of Open Source. Journal of Industrial Economics, 50, no. 2, pp. 197-234

Miller, B. P. and D. Koski, C. Pheow Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl. 1995. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. University of Wisconsin working paper. ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.pdf.

Mockus, A., R. T. Fielding and J. Herbsleb. (2000) A Case Study of Open Source Software Development: The Apache Server, forthcoming in Proceedings of ICSE2000, also at http://opensource.mit.edu/papers/mockusapache.pdf.

Moody, G.. 2001. Rebel Code: The Inside Story of Linux and the Open Source Revolution,. Perseus Publishing, Cambridge, Ma.

Mustonen, M., 2003. Copyleft—the economics of Linux and other open source software, Information Economics and Policy 15, pp. 99–121.

The Netcraft Web Server Survey. http://www.netcraft.com/survey/. Accessed 5/2002.

Palfrey, T.R. and H. Rosenthal, 1984, Participation and the Provision of Discrete Public Goods: A Strategic Analysis, Journal of Public Economics, 24, pp. 171–193.

Raymond, E. S. The Cathedral and the Bazaar. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/.

Rossi, M. 2004. Decoding the 'Free/Open Source (F/OSS) Software Puzzle'a survey of theoretical and empirical contributions, [http://opensource.mit.edu/papers/rossi.pdf]

Security Space, Apache Module Report, https://secure1.securityspace.com/s_survey/data/man.200204/apachemods.html, accessed 5/25/2002.

Segal, I. 1999. Complexity and Renegotiation: A Foundation for Incomplete Contracts, Review of Economic Studies, 66, pp. 57-82.

Tirole, J. 1999. Incomplete Contracts: Where do we stand? Econometrica, 67, no. 4, pp. 741-81.

Utterback, J. M. 1996. Mastering the Dynamics of Innovation, Harvard Business School Press.

Figure 1. Packaged Software Share of All Software Investment

Source: Grimm and Parker (2000)

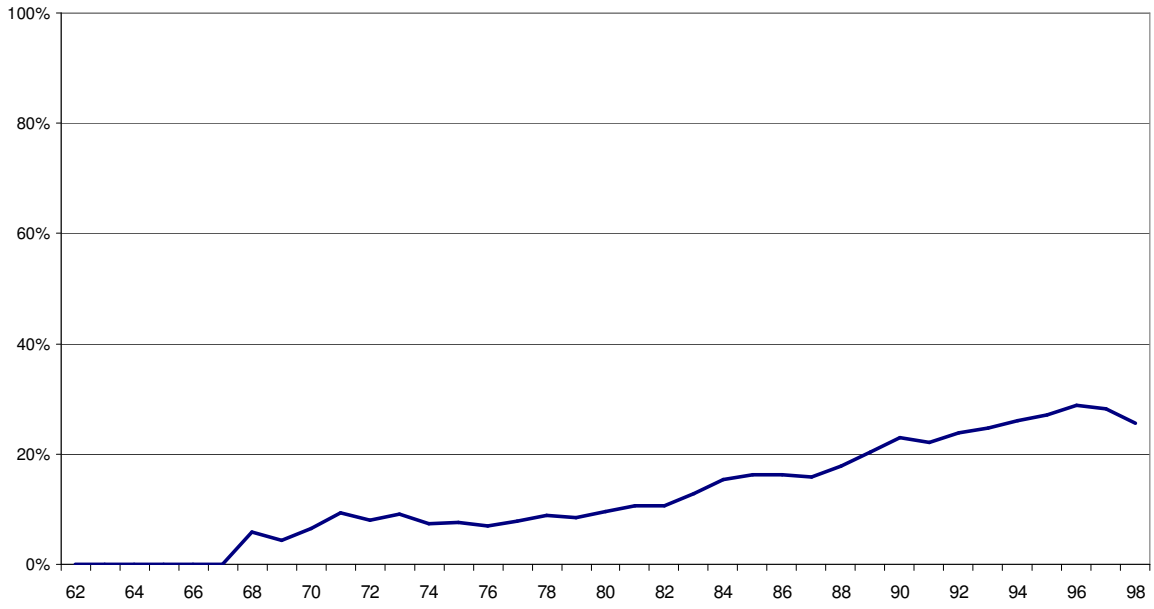

Figure 2. Development contract games

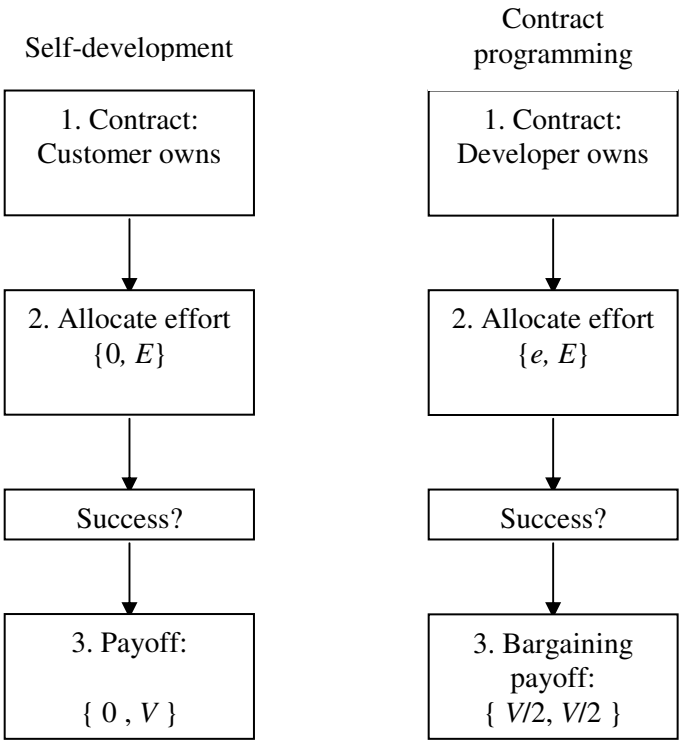

Figure 3. Pre-packaged software game