11 11 TEX LATEX

The Magic Cauldron

by Eric S. Raymond June 1999

This paper analyzes the evolving economic substrate of the open-source phenomenon. We first explode some prevalent myths about the funding of program development and the price structure of software. We present a game-theory analysis of the stability of open-source cooperation. We present nine models for sustainable funding of open-source development; two non-profit, seven for-profit. We continue to develop a qualitative theory of when it is economically rational to be closed. We then examine some novel additional mechanisms the market is now inventing to fund for-profit open-source development, including the reinvention of the patronage system and task markets. We conclude with some tentative predictions of the future.

Contents

1 Indistinguishable From Magic **2**

2 Beyond Geeks Bearing Gifts **2**

3 The Manufacturing Delusion **3**

4 The "information wants to be free" Myth **5**

5 The Inverse Commons **6**

6 Reasons for Closing Source **7**

7 Use-Value Funding Models **8**

7.1 The Apache case: cost-sharing . . . 8

7.2 The Cisco case: risk-spreading . . . 9

8 Why Sale Value is Problematic **9**

9 Indirect Sale-Value Models **10**

9.1 Loss-Leader/Market Positioner . . . 11

9.2 Widget Frosting . . . 11

9.3 Give Away the Recipe, Open A Restaurant . . . 12

9.4 Accessorizing . . . 13

9.5 Free the Future, Sell the Present . . . 13

9.6 Free the Software, Sell the Brand . . . 13

9.7 Free the Software, Sell the Content . . . 13

10 When To Be Open, When To Be Closed **14**

10.1 What Are the Payoffs? . . . 14

10.2 How Do They Interact? . . . 14

10.3 Doom: A Case Study . . . 16

10.4 Knowing When To Let Go . . . 17

11 The Business Ecology of Open Source **17**

12 Coping With Success **18**

13 Open R&D and the Reinvention of Patronage **19**

14 Getting There From Here **20**

15 Conclusion: Life After The Revolution **21**

16 Bibliography and Acknowledgements **22**

17 Appendix: Why Closing Drivers Loses A Vendor Money **23**

18 History **24**

1 Indistinguishable From Magic

In Welsh myth, the goddess Ceridwen owned a great cauldron which would magically produce nourishing food – when commanded by a spell known only to the goddess. In modern science, Buckminster Fuller gave us the concept of 'ephemeralization', technology becoming both more effective and less expensive as the physical resources invested in early designs are replaced by more and more information content. Arthur C. Clarke connected the two by observing that "Any sufficiently advanced technology is indistinguishable from magic".

To many people, the successes of the open-source community seem like an implausible form of magic. High-quality software materializes "for free", which is nice while it lasts but hardly seems sustainable in the real world of competition and scarce resources. What's the catch? Is Ceridwen's cauldron just a conjuring trick? And if not, how does ephemeralization work in this context – what spell is the goddess speaking?

2 Beyond Geeks Bearing Gifts

The experience of the open-source culture has certainly confounded many of the assumptions of people who learned about software development outside it. "The Cathedral and the Bazaar" 16 () described the ways in which decentralized cooperative software development effectively overturns Brooks's Law, leading to unprecedented levels of reliability and quality on individual projects. "Homesteading the Noosphere" 16 () examined the social dynamics within which this 'bazaar' style of development is situated, arguing that it is most effectively understood not in conventional exchange-economy terms but as what anthropologists call a 'gift culture' in which members compete for status by giving things away. In this paper we shall begin by exploding some common myths about software production economics; then continue the analysis of 16 () and 16 () into the realm of economics, game theory and business models, developing new conceptual tools needed to understand the way that the gift culture of open-source developers can sustain itself in an exchange economy.

In order to pursue this line of analysis without distraction, we'll need to abandon (or at least agree to temporarily ignore) the 'gift culture' level of explanation. 16 () posited that gift culture behavior arises in situations where survival goods are abundant enough to make the exchange game no longer very interesting; but while this appears sufficiently powerful as a *psychological* explanation of behavior, it lacks suffiency as an explanation of the mixed *economic* context in which most open-source developers actually operate. For most, the exchange game has lost its appeal but not its power to constrain. Their behavior has to make sufficient material-scarcity-economics sense to keep them in a gift-culture-supporting zone of surplus.

Therefore, we now will consider (from entirely *within* the realm of scarcity economics) the modes of cooperation and exchange that sustain open-source development. While doing so we will answer the pragmatic question "How do I make money at this?", in detail and with examples. First, though, we will show that much of the tension behind that question derives from prevailing folk models of software-production economics that are false to fact.

(A final note before the exposition: the discussion and advocacy of open-source development in this paper should not be construed as a case that closed-source development is intrinsically wrong, nor as a brief against intellectual-property rights in software, nor as an altruistic appeal to 'share'. While these arguments are still beloved of a vocal minority in the open-source development community, experience since 16 () has made it clear that they are unnecessary. An entirely sufficient case for open-source development rests on its engineering and economic outcomes – better quality, higher reliability, lower costs, and increased choice.)

3 The Manufacturing Delusion

We need to begin by noticing that computer programs like all other kinds of tools or capital goods, have two distinct kinds of economic value. They have *use value* and *sale value*.

The *use value* of a program is its economic value as a tool. The *sale value* of a program is its value as a salable commodity. (In professional economist-speak, sale value is value as a final good, and use value is value as an intermediate good.)

When most people try to reason about software-production economics, they tend to assume a 'factory model' that is founded on the following fundamental premises.

1. Most developer time is paid for by sale value.

2. The sale value of software is proportional to its development cost (i.e. the cost of the resources required to functionally replicate it) and to its use value.

In other words, people have a strong tendency to assume that software has the value characteristics of a typical manufactured good. But both of these assumptions are demonstrably false.

First, code written for sale is only the tip of the programming iceberg. In the pre-microcomputer era it used to be a commonplace that 90% of all the code in the world was written in-house at banks and insurance companies. This is probably no longer the case – other industries are much more software-intensive now, and the finance industry's share of the total has accordingly dropped – but we'll see shortly that there is empirical evidence that around 95% of code is still written in-house.

This code includes most of the stuff of MIS, the financial- and database-software customizations every medium and large company needs. It includes technical-specialist code like device drivers (almost nobody makes money selling device drivers, a point we'll return to later on). It includes all kinds of embedded code for our increasingly microchip-driven machines - from machine tools and jet airliners to cars to microwave ovens and toasters.

Most such in-house code is integrated with its environment in ways that make reusing or copying it very difficult. (This is true whether the 'environment' is a business office's set of procedures or the fuel-injection system of a combine harvester.) Thus, as the environment changes, there is a lot of work continually needed to keep the software in step.

This is called 'maintenance', and any software engineer or systems analyst will tell you that it makes up the vast majority (more than 75%) of what programmers get paid to do. Accordingly, most programmer-hours are spent (and most programmer salaries are paid for) writing or maintaining in-house code that has no sale value at all – a fact the reader may readily check by examining the listings of programming jobs in any newspaper with a 'Help Wanted' section.

Scanning the employment section of your local newspaper is an enlightening experiment which I urge the reader to perform for him- or herself. Examine the jobs listings under programming, data processing, and software engineering for positions that involve the development of software. Categorize each such job according to whether the software is being developed for use or for sale.

It will quickly become clear that, even given the most inclusive definition of 'for sale', at least nineteen in twenty of the salaries offered are being funded strictly by use value (that is, value as an intermediate good). This is our reason for believing that only 5% of the industry is sale-value-driven. Note, however, that the rest of the analysis in this paper is relatively insensitive to this number; if it were 15% or even 20%, the economic consequences would remain essentially the same.

(When I speak at technical conferences, I usually begin my talk by asking two questions: how many in the audience are paid to write software, and for how many do their salaries depend on the sale value of software. I generally get a forest of hands for the first question, few or none for the second, and considerable audience surprise at the proportion.)

Second, the theory that the sale value of software is coupled to its development or replacement costs is even more easily demolished by examining the actual behavior of consumers. There are many goods for which a proportion of this kind actually holds (before depreciation) – food, cars, machine tools. There are even many intangible goods for which sale value couples strongly to development and replacement cost – rights to reproduce music or maps or databases, for example. Such goods may retain or even increase their sale value after their original vendor is gone.

By contrast, when a software product's vendor goes out of business (or if the product is merely discontinued), the maximum price consumers will pay for it rapidly goes to near zero regardless of its theoretical use value or the development cost of a functional equivalent. (To check this assertion, examine the remainder bins at any software store near you.)

The behavior of retailers when a vendor folds is very revealing. It tells us that they know something the vendors don't. What they know is this: the price a consumer will pay is effectively capped by the *expected future value of vendor service* (where 'service' is here construed broadly to include enhancements, upgrades, and follow-on projects).

In other words, software is largely a service industry operating under the persistent but unfounded delusion that it is a manufacturing industry.

It is worth examining why we normally tend to believe otherwise. It may simply be because the small portion of the software industry that manufactures for sale is also the only part that advertises its product. Also, some of the most visible and heavily advertised products are ephemera like games that have little in the way of continuing service requirements (the exception, rather than the rule) 16 ().

It is also worth noting that the manufacturing delusion encourages price structures that are pathologically out of line with the actual breakdown of development costs. If (as is generally accepted) over 75% of a typical software project's life-cycle costs will be in maintenance and debugging and extensions, then the common price policy of charging a high fixed purchase price and relatively low or zero support fees is bound to lead to results that serve all parties poorly.

Consumers lose because, even though software is a service industry, the incentives in the factory model all cut against a vendor's offering *competent* service. If the vendor's money comes from selling bits, most effort will go to making bits and shoving them out the door; the help desk, not a profit center, will become a dumping ground for the least effective and get only enough resources to avoid actively alienating a critical number of customers.

The other side of this coin is that most vendors buying this factory model will also fail in the longer run. Funding indefinitely-continuing support expenses from a fixed price is only viable in a market that is expanding fast enough to cover the support and life-cycle costs entailed in yesterday's sales with tomorrow's revenues. Once a market matures and sales slow down, most vendors will have no choice but to cut expenses by orphaning the product.

Whether this is done explicitly (by discontinuing the product) or implicitly (by making support hard to get), it has the effect of driving customers to competitors (because it destroys the product's expected future value, which is contingent on that service). In the short run, one can escape this trap by making bug-fix releases pose as new products with a new price attached, but consumers quickly tire of this. In the long run, therefore, the only way to escape is to have no competitors – that is, to have an effective monopoly on one's market. In the end, there can be only one.

And, indeed, we have repeatedly seen this support-starvation failure mode kill off even strong second-place competitors in a market niche. (The pattern should be particularly clear to anyone who has ever surveyed the history of proprietary PC operating systems, word processors, accounting programs or business software in general.) The perverse incentives set up by the factory model lead to a winner-take-all market dynamic in which even the winner's customers end up losing.

If not the factory model, then what? To handle the real cost structure of the software life-cycle efficiently (in both the informal and economics-jargon senses of 'efficiency'), we require a price structure founded on service contracts, subscriptions, and a *continuing* exchange of value between vendor and customer. Under the efficiency-seeking conditions of the free market, therefore, we can predict that this is the sort of price structure most of a mature software industry will ultimately follow.

The foregoing begins to give us some insight into why open-source software increasingly poses not merely a technological but an economic challenge to the prevailing order. The effect of making software 'free', it seems, is to force us into that service-fee-dominated world – and to expose what a relatively weak prop the sale value of closed-source bits was all along.

The term 'free' is misleading in another way as well. Lowering the cost of a good tends to increase, rather than decrease, total investment in the infrastructure that sustains it. When the price of cars goes down, the demand for auto mechanics goes up – which is why even those 5% of programmers now compensated by sale-value would be unlikely to suffer in an open-source world. The people who lose in the transition won't be programmers, they will be investors who have bet on closed-source strategies where they're not appropriate.

4 The "information wants to be free" Myth

There is another myth, equal and opposite to the factory-model delusion, which often confuses peoples' thinking about the economics of open-source software. It is that "information wants to be free". This usually unpacks to a claim that the zero marginal cost of reproducing digital information implies that its clearing price ought to be zero.

The most general form of this myth is readily exploded by considering the value of information that constitutes a claim on a rivalrous good – a treasure map, say, or a Swiss bank account number, or a claim on services such as a computer account password. Even though the claiming information can be duplicated at zero cost, the item being claimed cannot be. Hence, the non-zero marginal cost for the item can be inherited by the claiming information.

We mention this myth mainly to assert that it is unrelated to the economic-utility arguments for open source; as we'll see later, those would generally hold up well even under the assumption that software actually *does* have the (nonzero) value structure of a manufactured good. We therefore have no need to tackle the question of whether software 'should' be free or not.

5 The Inverse Commons

Having cast a skeptical eye on one prevailing model, let's see if we can build another – a hard-nosed economic explanation of what makes open-source cooperation sustainable.

This is a question that bears examination on a couple of different levels. On one level, we need to explain the behavior of individuals who contribute to open-source projects; on another, we need to understand the economic forces that sustain cooperation on open-source projects like Linux or Apache.

Again, we must first demolish a widespread folk model that interferes with understanding. Over every attempt to explain cooperative behavior there looms the shadow of Garret Hardin's Tragedy of the Commons.

Hardin famously asks us to imagine a green held in common by a village of peasants, who graze their cattle there. But grazing degrades the commons, tearing up grass and leaving muddy patches, which re-grow their cover only slowly. If there is no agreed-on (and enforced!) policy to allocate grazing rights that prevents overgrazing, all parties' incentives push them to run as many cattle as quickly as possible, trying to extract maximum value before the commons degrades into a sea of mud.

Most people have an intuitive model of cooperative behavior that goes much like this. It's not actually a good diagnosis of the economic problems of open-source, which are free-rider (underprovision) rather than congested-public-good (overuse). Nevertheless, it is the analogy I hear behind most off-the-cuff objections.

The tragedy of the commons predicts only three possible outcomes. One is the sea of mud. Another is for some actor with coercive power to enforce an allocation policy on behalf of the village (the communist solution). The third is for the commons to break up as village members fence off bits they can defend and manage sustainably (the property-rights solution).

When people reflexively apply this model to open-source cooperation, they expect it to be unstable with a short half-life. Since there's no obvious way to enforce an allocation policy for programmer time over the internet, this model leads straight to a prediction that the commons will break up, with various bits of software being taken closed-source and a rapidly decreasing amount of work being fed back into the communal pool.

In fact, it is empirically clear that the trend is opposite to this. The breadth and volume of open-source development (as measured by, for example, submissions per day at Metalab or announcements per day at freshmeat.net) is steadily increasing. Clearly there is some critical way in which the "Tragedy of the Commons" model fails to capture what is actually going on.

Part of the answer certainly lies in the fact that using software does not decrease its value. Indeed, widespread use of open-source software tends to *increase* its value, as users fold in their own fixes and features (code patches). In this inverse commons, the grass grows taller when it's grazed on.

Another part of the answer lies in the fact that the putative market value of small patches to a common source base is hard to capture. Supposing I write a fix for an irritating bug, and suppose many people realize the fix has money value; how do I collect from all those people? Conventional payment systems have high enough overheads to make this a real problem for the sorts of micropayments that would usually be appropriate.

It may be more to the point that this value is not merely hard to capture, in the general case it's hard to even *assign*. As a thought experiment let us suppose that the Internet came equipped with the theoretically ideal micropayment system – secure, universally accessible, zero-overhead. Now let's say you have written a patch labeled "Miscellaneous Fixes to the Linux Kernel". How do you know what price to ask? How would a potential buyer, not having seen the patch yet, know what is reasonable to pay for it?

What we have here is almost like a funhouse-mirror image of F.A. Hayek's 'calculation problem' – it would take a superbeing, both able to evaluate the functional worth of patches and trusted to set prices accordingly, to lubricate trade.

Unfortunately, there's a serious superbeing shortage, so patch author J. Random Hacker is left with two choices: sit on the patch, or throw it into the pool for free. The first choice gains nothing. The second choice may gain nothing, or it may encourage reciprocal giving from others that will address some of J. Random's problems in the future. The second choice, apparently altruistic, is actually optimally selfish in a game-theoretic sense.

In analyzing this kind of cooperation, it is important to note that while there is a free-rider problem (work may be underprovided in the absence of money or money-equivalent compensation) it is not one that scales with the number of end-users. The complexity and communications overhead of an open-source project is almost entirely a function of the number of developers involved; having more end-users who never look at source costs effectively nothing. It may increase the rate of silly questions appearing on the project mailing lists, but this is relatively easily forestalled by maintaining a Frequently Asked Questions list and blithely ignoring questioners who have obviously not read it (and in fact both these practices are typical).

The real free-rider problems in open-source software are more a function of friction costs in submitting patches than anything else. A potential contributor with little stake in the cultural reputation game (see 16 ()) may, in the absence of money compensation, think "It's not worth submitting this fix because I'll have to clean up the patch, write a ChangeLog entry, and sign the FSF assignment papers...". It's for this reason that the number of contributors (and, at second order, the success of) projects is strongly and inversely correlated with the number of hoops each project makes a user go through to contribute. Such friction costs may be political as well as mechanical. Together they may explain why the loose, amorphous Linux culture has attracted orders of magnitude more cooperative energy than the more tightly organized and centralized BSD efforts and why the Free Software Foundation has receded in relative importance as Linux has risen.

This is all good as far as it goes. But it is an after-the-fact explanation of what J. Random Hacker does with his patch after he has it. The other half we need is an economic explanation of how JRH was able to write that patch in the first place, rather than having to work on a closed-source program that might have returned him sale value. What business models create niches in which open-source development can flourish?

6 Reasons for Closing Source

Before taxonomizing open-source business models, we should deal with exclusion payoffs in general. What exactly are we protecting when we close source?

Let's say you hire someone to write to order (say) a specialized accounting package for your business. That problem won't be solved any better if the sources are closed rather than open; the only rational reasons you might want them to be closed is if you want to sell the package to other people, or deny its use to competitors.

The obvious answer is that you're protecting sale value, but for the 95% of software written for internal use this doesn't apply. So what other gains are there in being closed?

That second case (protecting competitive advantage) bears a bit of examination. Suppose you open-source that accounting package. It becomes popular and benefits from improvements made by the community. Now, your competitor also starts to use it. The competitor gets the benefit without paying the development cost and cuts into your business. Is this an argument against open-sourcing?

Maybe – and maybe not. The real question is whether your gain from spreading the development load exceeds your loss due to increased competition from the free rider. Many people tend to reason poorly about this tradeoff through (a) ignoring the functional advantage of recruiting more development help. (b) not treating the development costs as sunk, and By hypothesis, you had to pay th development costs anyway, so counting them as a cost of open-sourcing (if you choose to do) is mistaken.

There are other reasons for closing source that are outright irrational. You might, for example, be laboring under the delusion that closing the sources will make your business systems more secure against crackers and intruders. If so, I recommend therapeutic conversation with a cryptographer immediately. The really professional paranoids know better than to trust the security of closed-source programs, because they've learned through hard experience not to. Security is an aspect of reliability; only algorithms and implementations that have been thoroughly peer-reviewed can possibly be trusted to be secure.

7 Use-Value Funding Models

A key fact that the distinction between use and sale value allows us to notice is that only *sale value* is threatened by the shift from closed to open source; use value is not.

If use value rather than sale value is really the major driver of software development, and (as was argued in 16 ()) open-source development is really more effective and efficient than closed, then we should expect to find circumstances in which expected use value alone sustainably funds open-source development.

And in fact it is not difficult to identify at least two important models in which full-time developer salaries for open-source projects are funded strictly out of use value.

7.1 The Apache case: cost-sharing

Let's say you work for a firm that has a business-critical requirement for a high-volume, high-reliability web server. Maybe it's for electronic commerce, maybe you're a high-visibility media outlet selling advertising, maybe you're a portal site. You need 24/7 uptime, you need speed, and you need customizability.

How are you going to get these things? There are three basic strategies you can pursue:

Buy a proprietary webserver. In this case, you are betting that the vendor's agenda matches yours and that the vendor has the technical competence to implement properly. Even assuming both these things to be true, the product is likely to come up short in customizability; you will only be able to modify it through the hooks the vendor has chosen to provide. This proprietary-webserver path is not a popular one.

Roll your own. Building your own webserver is not an option to dismiss instantly; webservers are not very complex, certainly less so than browsers, and a specialized one can be very lean and mean. Going this path, you can get the exact features and customizability you want, though you'll pay for it in development time. Your firm may also find it has a problem when you retire or leave.

Join the Apache group. The Apache server was built by an Internet-connected group of webmasters who realized that it was smarter to pool their efforts into improving one code base than to run a large number of parallel development efforts. By doing this they were able to capture both most of the advantages of roll-your-own and the powerful debugging effect of massively-parallel peer review.

The advantage of the Apache choice is very strong. Just how strong, we may judge from the monthly Netcraft survey, which has shown Apache steadily gaining market share against all proprietary webservers since its inception. As of June 1999, Apache and its derivatives have *61% market share* <http://www.netcraft.com/survey/> – with no legal owner, no promotion, and no contracted service organization behind them at all.

The Apache story generalizes to a model in which software users find it to their advantage to fund open-source development because doing so gets them a better product than they could otherwise have, at lower cost.

7.2 The Cisco case: risk-spreading

Some years ago, two programmers at Cisco (the networking-equipment manufacturer) got assigned the job of writing a distributed print-spooling system for use on Cisco's corporate network. This was quite a challenge. Besides supporting the ability for arbitrary user A to print at arbitrary printer B (which might be in the next room or a thousand miles away), the system had to make sure that in the event of a paper-out or toner-low condition the job would get rerouted to an alternate printer near the target. The system also needed to be able to report such problems to a printer administrator.

The duo came up with a clever set of modifications to the standard Unix print-spooler software, plus some wrapper scripts, that did the job. Then they realized that they, and Cisco, had a problem.

The problem was that neither of them was likely to be at Cisco forever. Eventually, both programmers would be gone, and the software would be unmaintained and begin to rot (that is, to gradually fall out of sync with real-world conditions). No developer likes to see this happen to his or her work, and the intrepid duo felt Cisco had paid for a solution under the not unreasonable expectation that it would outlast their own jobs there.

Accordingly, they went to their manager and urged him to authorize the release of the print spooler software as open source. Their argument was that Cisco would have no sale value to lose, and much else to gain. By encouraging the growth of a community of users and co-developers spread across many corporations, Cisco could effectively hedge against the loss of the software's original developers.

The Cisco story generalizes to a model in which open source functions not so much to lower costs as to spread risk. All parties find that the openness of the source, and the presence of a collaborative community funded by multiple independent revenue streams, provides a fail-safe that is itself economically valuable – sufficiently valuable to drive funding for it.

8 Why Sale Value is Problematic

Open source makes it rather difficult to capture direct sale value from software. The difficulty is not technical; source code is no more nor less copyable than binaries, and the enforcement of copyright and license laws permitting capture of sale value would not by necessity be any more difficult for open-source products than it is for closed.

The difficulty lies rather with the nature of the social contract that supports open-source development. For three mutually reinforcing reasons, the major open-source licenses prohibit most of the sort of restrictions on use, redistribution and modification that would facilitate direct-sale revenue capture. To understand these reasons. we must examine the social context within which the licenses evolved; the Internet *hacker* <http://www.tuxedo.org/~esr/faqs/hacker-howto.html> culture.

Despite myths about the hacker culture still (in 1999) widely believed outside it, none of these reasons has to do with hostility to the market. While a minority of hackers does indeed remain hostile to the profit motive, the general willingness of the community to cooperate with for-profit Linux packagers like Red Hat, SUSE, and Caldera demonstrates that most hackers will happily work with the corporate world when it serves their ends. The real reasons hackers frown on direct-revenue-capture licenses are more subtle and interesting.

One reason has to do with symmetry. While most open-source developers do not intrinsically object to others profiting from their gifts, most also demand that no party (with the possible exception of the originator of a piece of code) be in a *privileged* position to extract profits. J. Random Hacker is willing for Fubarco to profit by selling his software or patches, but only so long as JRH himself could also potentially do so.

Another has to do with unintended consequences. Hackers have observed that licenses that include restrictions on and fees for 'commercial' use or sale (the most common form of attempt to recapture direct sale value, and not at first blush an unreasonable one) have serious chilling effects. A specific one is to cast a legal shadow on activities like redistribution in inexpensive CD-ROM anthologies, which we would ideally like to encourage. More generally, restrictions on use/sale/modification/distribution (and other complications in licensing) exact an overhead for conformance tracking and (as the number of packages people deal with rises) a combinatorial explosion of perceived uncertainty and potential legal risk. This outcome is considered harmful, and there is therefore strong social pressure to keep licenses simple and free of restrictions.

The final and most critical reason has to do with preserving the peer-review, gift-culture dynamic described in 16 (). License restrictions designed to protect intellectual property or capture direct sale value often have the effect of making it legally impossible to fork the project (this is the case, for example, with Sun's so-called "Community Source" licenses for Jini and Java). While forking is frowned upon and considered a last resort (for reasons discussed at length in 16 ()), it's considered critically important that that last resort be present in case of maintainer incompetence or defection (e.g. to a more closed license).

The hacker community has some give on the symmetry reason; thus, it tolerates licenses like Netscape's NPL that give some profit privileges to the originators of the code (specifically in the NPL case, the exclusive right to use the open-source Mozilla code in derivative products including closed source). It has less give on the unintended-consequences reason, and none on preserving the option to fork (which is why Sun's Java and Jini 'Community License' schemes have been largely rejected by the community).

These reasons explain the clauses of the Open Source Definition, which was written to express the consensus of the hacker community about the critical features of the standard licenses (the GPL, the BSD license, the MIT License, and the Artistic License). These clauses have the effect (though not the intention) of making direct sale value very hard to capture.

9 Indirect Sale-Value Models

Nevertheless, there are ways to make markets in software-related services that capture something like indirect sale value. There are five known and two speculative models of this kind (more may be developed in the future).

9.1 Loss-Leader/Market Positioner

In this model, you use open-source software to create or maintain a market position for proprietary software that generates a direct revenue stream. In the most common variant, open-source client software enables sales of server software, or subscription/advertising revenue associated with a portal site.

Netscape Communications, Inc. was pursuing this strategy when it open-sourced the Mozilla browser in early 1998. The browser side of their business was at 13% of revenues and dropping when Microsoft first shipped Internet Explorer. Intensive marketing of IE (and shady bundling practices that would later become the central issue of an antitrust lawsuit) quickly ate into Netscape's browser market share, creating concern that Microsoft intended to monopolize the browser market and then use de-facto control of HTML to drive Netscape out of the server market.

By open-sourcing the still-widely-popular Netscape browser, Netscape effectively denied Microsoft the possibility of a browser monopoly. They expected that open-source collaboration would accelerate the development and debugging of the browser, and hoped that Microsoft's IE would be reduced to playing catch-up and prevented from exclusively defining HTML.

This strategy worked. In November 1998 Netscape actually began to regain business-market share from IE. By the time Netscape was acquired by AOL in early 1999, the competitive advantage of keeping Mozilla in play was sufficiently clear that one of AOL's first public commitments was to continue supporting the Mozilla project, even though it was still in alpha stage.

9.2 Widget Frosting

This model is for hardware manufacturers (hardware, in this context, includes anything from Ethernet or other peripheral boards all the way up to entire computer systems). Market pressures have forced hardware companies to write and maintain software (from device drivers through configuration tools all the way up to the level of entire operating systems), but the software itself is not a profit center. It's an overhead – often a substantial one.

In this situation, opening source is a no-brainer. There's no revenue stream to lose, so there's no downside. What the vendor gains is a dramatically larger developer pool, more rapid and flexible response to customer needs, and better reliability through peer review. It gets ports to other environments for free. It probably also gains increased customer loyalty as its customers' technical staffs put increasing amounts of time into the code to do the customizations they require.

There are a couple of vendor objections commonly raised specifically to open-sourcing hardware drivers. Rather than mix them with discussion of more general issues here, I have written an 17 (appendix) specifically on this topic.

The 'future-proofing' effect of open source is particularly strong with respect to widget frosting. Hardware products have a finite production and support lifetime; after that, the customers are on their own. But if they have access to driver source and can patch them as needed, they're more likely to be happier repeat customers of the same company.

A very dramatic example of adopting the widget frosting model was Apple Computer's decision in mid-March 1999 to open-source "Darwin", the core of their MacOSX server operating system.

9.3 Give Away the Recipe, Open A Restaurant

In this model, one open-sources software to create a market position not for closed software (as in the Loss-Leader/Market-Positioner case) but for services.

(I used to call this 'Give Away the Razor, Sell Razor Blades', The coupling is not really as tight as the razor/razor-blade analogy implies.)

This is what Red Hat and other Linux distributors do. What they are actually selling is not the software, the bits itself, but the value added by assembling and testing a running operating system that is warranted (if only implicitly) to be merchantable and to be plug-compatible with other operating systems carrying the same brand. Other elements of their value proposition include free installation support and the provision of options for continuing support contracts.

The market-building effect of open source can be extremely powerful, especially for companies which are inevitably in a service position to begin with. One very instructive recent case is Digital Creations, a website-design house started up in 1998 that specializes in complex database and transaction sites. Their major tool, the intellectual-property crown jewels of the company, is an object publisher that has been through several names and incarnations but is now called Zope.

When the Digital Creations people went looking for venture capital, the VC they brought in carefully evaluated their prospective market niche, their people, and their tools. He then recommended that Digital Creations take Zope to open source.

By traditional software-industry standards, this looks like an absolutely crazy move. Conventional business-school wisdom has it that core intellectual property like Zope is a company's crown jewels, never under any circumstances to be given away. But the VC had two related insights. One is that Zope's true core asset is actually the brains and skills of its people. The second is that Zope is likely to generate more value as a market-builder than as a secret tool.

To see this, compare two scenarios. In the conventional one, Zope remains Digital Creations's secret weapon. Let's stipulate that it's a very effective one. As a result, the firm will able to deliver superior quality on short schedules – *but nobody knows that*. It will be easy to satisfy customers, but harder to build a customer base to begin with.

The VC, instead, saw that open-sourcing Zope could be critical advertising for Digital Creations's *real* asset – its people. He expected that customers evaluating Zope would consider it more efficient to hire the experts than to develop in-house Zope expertise.

One of the Zope principals has since confirmed very publicly that their open-source strategy has "opened many doors we wouldn't have got in otherwise". Potential customers do indeed respond to the logic of the situation – and Digital Creations, accordingly, is prospering.

Another up-to-the-minute example is *e-smith, inc.* `<http://www.e-smith.net/>`. This company sells support contracts for turnkey Internet server software that is open-source, a customized Linux. One of the principals, describing the spread of free downloads of e-smith's software, *says* `<http://www.globetechnology.com/gam/News/19990625/BAND.html>` "Most companies would consider that software piracy; we consider it free marketing".

9.4 Accessorizing

In this model, you sell accessories for open-source software. At the low end, mugs and T-shirts; at the high end, professionally-edited and produced documentation.

O'Reilly Associates, publishers of many excellent references volumes on open-source software, is a good example of an accessorizing company. O'Reilly actually hires and supports well-known open-source hackers (such as Larry Wall and Brian Behlendorf) as a way of building its reputation in its chosen market.

9.5 Free the Future, Sell the Present

In this model, you release software in binaries and source with a closed license, but one that includes an expiration date on the closure provisions. For example, you might write a license that permits free redistribution, forbids commercial use without fee, and guarantees that the software come under GPL terms a year after release or if the vendor folds.

Under this model, customers can ensure that the product is customizable to their needs, because they have the source. The product is future-proofed – the license guarantees that an open source community can take over the product if the original company dies.

Because the sale price and volume are based on these customer expectations, the original company should enjoy enhanced revenues from its product versus releasing it with an exclusively closed source license. Furthermore, as older code is GPLed, it will get serious peer review, bug fixes, and minor features, which removes some of the 75% maintainance burden on the originator.

This model has been successfully pursued by Aladdin Enterprises, makers of the popular Ghostscript program (a PostScript interpreter that can translate to the native languages of many printers).

The main drawback of this model is that the closure provisions tend to inhibit peer review and participation early in the product cycle, precisely when they are needed most.

9.6 Free the Software, Sell the Brand

This is a speculative business model. You open-source a software technology, retain a test suite or set of compatibility criteria, then sell users a brand certifying that their implementation of the technology is compatible with all others wearing the brand.

(This is how Sun Microsystems ought to be handling Java and Jini.)

9.7 Free the Software, Sell the Content

This is another speculative business model. Imagine something like a stock-ticker subscription service. The value is neither in the client software nor the server but in providing objectively reliable information. So you open-source all the software and sell subscriptions to the content. As hackers port the client to new platforms and enhance it in various ways, your market automatically expands.

(This is why AOL ought to open-source its client software.)

10 When To Be Open, When To Be Closed

Having reviewed business models that support open-source software development, we can now approach the general question of when it makes economic sense to be open-source and when to be closed-source. First, we must be clear what the payoffs are from each strategy.

10.1 What Are the Payoffs?

The closed-source approach allows you to collect rent from your secret bits; on the other hand, it forecloses the possibility of truly independent peer review. The open-source approach sets up conditions for independent peer review, but you don't get rent from your secret bits.

The payoff from having secret bits is well understood; traditionally, software business models have been constructed around it. Until recently, the payoff from independent peer review was not well understood. The Linux operating system, however, drives home a lesson that we should probably have learned years ago from the history of the Internet's core software and other branches of engineering – that open-source peer review is the only scalable method for achieving high reliability and quality.

In a competitive market, therefore, customers seeking high reliability and quality will reward software producers who go open-source and discover how to maintain a revenue stream in the service, value-add, and ancilliary markets associated with software. This phenomenon is what's behind the astonishing success of Linux, which came from nowhere in 1996 to over 17% in the business server market by the end of 1998 and seems on track to dominate that market within two years (in early 1999 IDC projected that Linux would grow faster than all other operating systems combined through 2003).

An almost equally important payoff of open source is its utility as a way to propagate open standards and build markets around them. The dramatic growth of the Internet owes much to the fact that nobody owns TCP/IP; nobody has a proprietary lock on the core Internet protocols.

The network effects behind TCP/IP's and Linux's success are fairly clear and reduce ultimately to issues of trust and symmetry – potential parties to a shared infrastructure can rationally trust it more if they can see how it works all the way down, and will prefer an infrastructure in which all parties have symmetrical rights to one in which a single party is in a privileged position to extract rents or exert control.

It is not, however, actually necessary to assume network effects in order for symmetry issues to be important to software consumers. No software consumer will rationally choose to lock itself into a supplier-controlled monopoly by becoming dependent on closed source if any open-source alternative of acceptable quality is available. This argument gains force as the software becomes more critical to the software consumer's business – the more vital it is, the less the consumer can tolerate having it controlled by an outside party.

Finally, an important customer payoff of open-source software related to the trust issue is that it's future-proof. If sources are open, the customer has some recourse if the vendor goes belly-up. This may be particularly important for widget frosting, since hardware tends to have short life cycles, but the effect is more general and translates into increased value for open-source software.

10.2 How Do They Interact?

When the rent from secret bits is higher than the return from open source, it makes economic sense to be closed-source. When the return from open source is higher than the rent from secret bits, it makes sense to go open source.

In itself, this is a trivial observation. It becomes nontrivial when we notice that the payoff from open source is harder to measure and predict than the rent from secret bits – and that said payoff is grossly underestimated much more often than it is overestimated. Indeed, until the mainstream business world began to rethink its premises following the Mozilla source release in early 1998, the open-source payoff was incorrectly but very generally assumed to be zero.

So how can we evaluate the payoff from open source? It's a difficult question in general, but we can approach it as we would any other predictive problem. We can start from observed cases where the open-source approach has succeeded or failed. We can try to generalize to a model which gives at least a qualitative feel for the contexts in which open source is a net win for the investor or business trying to maximize returns. We can then go back to the data and try to refine the model.

From the analysis presented in 16 (), we can expect that open source has a high payoff where (a) reliability/stability/scalability are critical, and (b) correctness of design and implementation is not readily verified by means other than independent peer review. (The second criterion is met in practice by most non-trivial programs.)

A consumer's rational desire to avoid being locked into a monopoly supplier will increase its interest in open source (and, hence, the competitive-market value for suppliers of going open) as the software becomes more critical to that consumer. Thus, another criterion (c) pushes towards open source when the software is a business-critical capital good (as, for example, in many corporate MIS departments).

As for application area, we observed above that open-source infrastructure creates trust and symmetry effects that, over time, will tend to attract more customers and to outcompete closed-source infrastructure; and it is often better to have a smaller piece of such a rapidly-expanding market than a bigger piece of a closed and stagnant one. Accordingly, for infrastructure software, an open-source play for ubiquity is quite likely to have a higher long-term payoff than a closed-source play for rent from intellectual property.

In fact, the ability of potential customers to reason about the future consequences of vendor strategies and their reluctance to accept a supplier monopoly implies a stronger constraint; without already having overwhelming market power, you can choose either an open-source ubiquity play or a direct-revenue-from-closed-source play – but not both. (Analogues of this principle are visible elsewhere, e.g. in electronics markets where customers often refuse to buy sole-source designs.) The case can be put less negatively: where network effects (positive network externalities) dominate, open source is likely to be the right thing.

We may sum up this logic by observing that open source seems to be most successful in generating greater returns than closed source in software that (d) establishes or enables a common computing and communications infrastructure.

Finally, we may note that purveyors of unique or just highly differentiated services have more incentive to fear copying of their methods by competitors than do vendors of services for which the critical algorithms and knowledge bases are well understood. Accordingly, open source is more likely to dominate when (e) key methods (or functional equivalents) are part of common engineering knowledge.

The Internet core software, Apache, and Linux's implementation of the ANSI-standard Unix API are prime exemplars of all five criteria. The path towards open source in the evolution of such markets are well-illustrated by the reconvergence of data networking on TCP/IP in the mid-1990s following fifteen years of failed empire-building attempts with closed protocols such as DECNET, XNS, IPX, and the like.

On the other hand, open source seems to make the least sense for companies that have unique possession of a value-generating software technology (strongly fulfilling criterion (e)) which is (a) relatively insensitive to failure, which can (b) readily be verified by means other than independent peer review, which is not

(c) business-critical, and which would not have its value substantially increased by (d) network effects or ubiquity.

As an example of this extreme case, in early 1999 I was asked "Should we go open source?" by a company that writes software to calculate cutting patterns for sawmills that want to extract the maximum yardage of planks from logs. My conclusion was “No.” The only criterion this comes even close to fulfilling is (c); but at a pinch, an experienced operator could generate cut patterns by hand.

An important point is that where a particular product or technology sits on these scales may change over time, as we’ll see in the following case study.

In summary, the following discriminators push towards open source:

(a)

reliability/stability/scalability are critical

(b)

correctness of design and implementation cannot readily be verified by means other than independent peer review

(c)

the software is critical to the user’s control of his/her business

(d)

the software establishes or enables a common computing and communications infrastructure

(e)

key methods (or functional equivalents of them) are part of common engineering knowledge.

10.3 Doom: A Case Study

The history of id software’s best-selling game Doom illustrates ways in which market pressure and product evolution can critically change the payoff magnitudes for closed vs. open source.

When Doom was first released in late 1993, its first-person, real-time animation made it utterly unique (the antithesis of criterion (e)). Not only was the visual impact of the technique stunning, but for many months nobody could figure out how it had been achieved on the underpowered microprocessors of that time. These secret bits were worth some very serious rent. In addition, the potential payoff from open source was low. As a solo game, the software (a) incurred tolerably low costs on failure, (b) not tremendously hard to verify, (c) not business-critical for any consumer, (d) did not benefit from network effects. It was economically rational for Doom to be closed source.

However, the market around Doom did not stand still. Would-be competitors invented functional equivalents of its animation techniques, and other “first-person shooter” games like Duke Nukem began to appear. As these games ate into Doom’s market share the value of the rent from secret bits went down.

On the other hand, efforts to expand that share brought on new technical challenges – better reliability, more game features, a larger user base, and multiple platforms. With the advent of multiplayer ‘deathmatch’ play and Doom gaming services, the market began to display substantial network effects. All this was demanding programmer-hours that id would have preferred to spend on the next game.

All of these trends raised the payoff from opening the source. At some point the payoff curves crossed over and it became economically rational for id to open up the Doom source and shift to making money in secondary markets such as game-scenario anthologies. And sometime after this point, it actually happened. The full source for Doom was released in late 1997.

10.4 Knowing When To Let Go

Doom makes an interesting case study because it is neither an operating system nor communications/networking software; it is thus far removed from the usual and obvious examples of open-source success. Indeed, Doom's life cycle, complete with crossover point, may be coming to typify that of applications software in today's code ecology – one in which communications and distributed computation both create serious robustness/reliability/scalability problems only addressible by peer review, and frequently cross boundaries both between technical environments and between competing actors (with all the trust and symmetry issues that implies).

Doom evolved from solo to deathmatch play. Increasingly, the network effect *is* the computation. Similar trends are visible even in the heaviest business applications, such as ERPs, as businesses network ever more intensively with suppliers and customers – and, of course, they are implicit in the whole architecture of the World Wide Web. It follows that almost everywhere, the open-source payoff is steadily rising.

If present trends continue, the central challenge of software technology and product management in the next century will be knowing when to let go – when to allow closed code to pass into the open-source infrastructure in order to exploit the peer-review effect and capture higher returns in service and other secondary markets.

There are obvious revenue incentives not to miss the crossover point too far in either direction. Beyond that, there's a serious opportunity risk in waiting too long – you could get scooped by a competitor going open-source in the same market niche.

The reason this is a serious issue is that both the pool of users and the pool of talent available to be recruited into open-source cooperation for any given product category is limited, and recruitment tends to stick. If two producers are the first and second to open-source competing code of roughly equal function, the first is likely to attract the most users and the most and best-motivated co-developers; the second will have to take leavings. Recruitment tends to stick, as users gain familiarity and developers sink time investments in the code itself.

11 The Business Ecology of Open Source

The open-source community has organized itself in a way that tends to amplify the productivity effects of open source. In the Linux world, in particular, it's an economically significant fact that there are multiple competing Linux distributors which form a tier separate from the developers.

Developers write code, and make the code available over the Internet. Each distributor selects some subset of the available code, integrates and packages and brands it, and sells it to customers. Users choose among distributions, and may supplement a distribution by downloading code directly from developer sites.

The effect of this tier separation is to create a very fluid internal market for improvements. Developers compete with each other, for the attention of distributors and users, on the quality of their software. Distributors compete for user dollars on the appropriateness of their selection policies, and on the value they can add to the software.

A first-order effect of this internal market structure is that no node in the net is indispensible. Developers can drop out; even if their portion of the code base is not picked up directly by some other developer, the competition for attention will tend to rapidly generate functional alternatives. Distributors can fail without damaging or compromising the common open-source code base. The ecology as a whole has a more rapid response to market demands, and more capability to resist shocks and regenerate itself, than any monolithic vendor of a closed-source operating system can possibly muster.

Another important effect is to lower overhead and increase efficiency through specialization. Developers don't experience the pressures that routinely compromise conventional closed projects and turn them into tar-pits – no lists of pointless and distracting check-list features from Marketing, no management mandates to use inappropriate and outdated languages or development environments, no requirement to re-invent wheels in a new and incompatible way in the name of product differentiation or intellectual-property protection, and (most importantly) *no deadlines*. No rushing a 1.0 out the door before it's done right – which (as DeMarco and Lister observed in their discussion of the 'wake me when it's over' management style in 16 ()) generally conduces not only to higher quality but actually to the most rapid delivery of a truly working result.

Distributors, on the other hand, get to specialize in the things distributors can do most effectively. Freed of the need to fund massive and ongoing software development just to stay competitive, they can concentrate on system integration, packaging, quality assurance, and service.

Both distributors and developers are kept honest by the constant feedback from and monitoring by users that is an integral part of the open-source method.

12 Coping With Success

The Tragedy of the Commons may not be applicable to open-source development as it happens today, but that doesn't mean there are not any reasons to wonder if the present momentum of the open-source community is sustainable. Will key players defect from cooperation as the stakes become higher?

There are several levels on which this question can be asked. Our 'Comedy of the Commons' counter-story is based on the argument that the value of individual contributions to open source is hard to monetize. But this argument has much less force for firms (like, say, Linux distributors) which already have a revenue stream associated with open source. Their contribution is already being monetized every day. Is their present cooperative role stable?

Examining this question will lead us to some interesting insights about the economics of open-source software in the real world of present time – and about what a true service-industry paradigm implies for the software industry in the future.

On the practical level, applied to the open-source community as it exists now, this question is usually posed in one of two different ways. One: will Linux fragment? Two: conversely, will Linux develop a dominant, quasi-monopolistic player?

The historical analogy many people turn to when suggesting that Linux will fragment is the behavior of the proprietary-Unix vendors in the 1980s. Despite endless talk of open standards, despite numerous alliances and consortia and agreements, proprietary Unix fell apart. The vendors' desire to differentiate their products by adding and modifying OS facilities proved stronger than their interest in growing the total size of the Unix market by maintaining compatibility (and consequently lowering both entry barriers for independent software developers and total cost of ownership for consumers).

This is quite unlikely to happen to Linux, for the simple reason that all the distributors are constrained to operate from a common base of open source code. It's not really possible for any one of them to maintain differentiation, because the licenses under which Linux code are developed effectively require them to share code with all parties. The moment any distributor develops a feature, all competitors are free to clone it.

Since all parties understand this, nobody even thinks about doing the kinds of maneuvers that fragmented proprietary Unix. Instead, Linux distributors are forced to compete in ways that actually *benefit* the consumer and the overall market. That is, they must compete on service, support, and their design bets on what interfaces actually conduce to ease installation and use.

The common source base also forecloses the possibility of monopolization. When Linux people worry about this, the name usually muttered is "Red Hat", that of the largest and most successful of the distributors (with somewhere around 90% estimated market share in the U.S.). But it is notable that within days after the May 1999 announcement of Red Hat's long-awaited 6.0 release – before Red Hat's CD-ROMs actually shipped in any quantity – CD-ROM images of the release built from Red Hat's own public FTP site were being advertised by a book publisher and several other CD-ROM distributors at lower prices than Red Hat's expected list.

Red Hat itself didn't turn a hair at this, because its founders understand very clearly that they do not and cannot own the bits in their product; the social norms of the Linux community forbid that. In a latter-day take on John Gilmore's famous observation that the Internet interprets censorship as damage and routes around it, it has been aptly said that the hacker community responsible for Linux interprets attempts at control as damage and routes around them. For Red Hat to have protested the pre-release cloning of its newest product would have seriously compromised its ability to elicit future cooperation from its developer community.

Perhaps more importantly in present time, the software licenses that express these community norms in a binding legal form actively forbid Red Hat from monopolizing the sources of the code their product is based on. The only thing they can sell is a brand/service/support relationship with people who are freely willing to pay for that. This is not a context in which the possibility of a predatory monopoly looms very large.

13 Open R&D and the Reinvention of Patronage

There is one other respect in which the infusion of real money into the open-source world is changing it. The community's stars are increasingly finding they can get paid for what they want to do, instead of pursuing open source as a hobby funded by another day job. Corporations like Red Hat, O'Reilly Associates, and VA Linux Systems are building what amount to semi-independent research arms with charters to hire and maintain stables of open-source talent.

This makes economic sense only if the cost per head of maintaining such a lab can easily be paid out of the expected gains it will enable by growing the firm's market faster. O'Reilly can afford to pay the principal authors of Perl and Apache to do their thing because it expects their efforts will enable it to sell more Perl- and Apache-related books. VA Linux Systems can fund its laboratory branch because improving Linux boosts the use value of the workstations and servers it sells. And Red Hat funds Red Hat Advanced Development Labs to increase the value of its Linux offering and attract more customers.

To strategists from more traditional sectors of the software industry, reared in cultures that regard patent- or trade-secret-protected intellectual property as the corporate crown jewels, this behavior may (despite its market-growing effect) seem inexplicable. Why fund research that every one of your competitors is (by definition) free to appropriate at no cost?

There seem to be two controlling reasons. One is that as long as these companies remain dominant players in their market niches, they can expect to capture a proportional lion's share of the returns from the open R&D. Using R&D to buy future profits is hardly a novel idea; what's interesting is the implied calculation that the expected future gains are sufficiently large that these companies can readily tolerate free riders.

While this obvious expected-future-value analysis is a necessary one in a world of hard-nosed capitalists keeping their eyes on ROI, it is not actually the most interesting mode of explanation for star-hiring, because the firms themselves advance a fuzzier one. They will tell you if asked that they are simply doing the right thing by the community they come from. Your humble author is sufficiently well-acquainted with principals at all three of the firms cited above to testify that these protestations cannot be dismissed as humbug. Indeed, I was personally recruited onto the board of VA Linux Systems in late 1998 explicitly so that I would be available to advise them on "the right thing", and have found them far from unwilling to listen when I did so.

An economist is entitled to ask what payoff is involved here. If we accept that talk of doing the right thing is not empty posturing, we should next inquire what self-interest of the firm the "right thing" serves. Nor is the answer, in itself, either surprising or difficult to verify by asking the right questions. As with superficially altruistic behavior in other industries, what these firms actually believe they're buying is goodwill.

Working to earn goodwill, and valuing it as an asset predictive of future market gains, is hardly novel either. What's interesting is the extremely high valuation that the behavior of these firms suggest they put on that goodwill. They're demonstrably willing to hire expensive talent for projects that are not direct revenue generators even during the most capital-hungry phases of the runup to IPO. And, at least so far, the market has actually rewarded this behavior.

The principals of these companies themselves are quite clear about the reasons that goodwill is especially valuable to them. They rely heavily on volunteers among their customer base both for product development and as an informal marketing arm. Their relationship with their customer base is intimate, often relying on personal trust bonds between individuals within and outside the firm.

These observations reinforce a lesson we learned earlier from a different line of reasoning. The relationship between Red Hat/VA/O'Reilly and their customers/developers is not one typical of manufacturing firms. Rather, it carries to an interesting extreme patterns that are characteristic of knowledge-intensive service industries. Looking outside the technology industry, we can see these patterns in (for example) law firms, medical practices, and universities.

We may observe, in fact, that open-source firms hire star hackers for much the same reasons that universities hire star academics. In both cases, the practice is similar in mechanism and effect to the system of aristocratic patronage that funded most fine art until after the Industrial Revolution – a similarity some parties to it are fully aware of.

14 Getting There From Here

The market mechanisms for funding (and making a profit from!) open-source development are still evolving rapidly. The business models we've reviewed in this paper probably will not be the last to be invented. Investors are still thinking through the consequences of reinventing the software industry as one with an explicit focus on service rather than closed intellectual property, and will be for some time to come.

This conceptual revolution will have some cost in foregone profits for people investing in the sale-value 5% of the industry; historically, service businesses are not as lucrative as manufacturing businesses (though as any doctor or lawyer could tell you, the return to the actual practitioners is often higher). Any foregone profits, however, will be more than matched by benefits on the cost side, as software consumers reap tremendous savings and efficiencies from open-source products. (There's a parallel here to the effects that the displacement of the traditional voice-telephone network by the Internet is having everywhere).

The promise of these savings and efficiencies is creating a market opportunity that entrepreneurs and venture capitalists are now moving in to exploit. As the first draft of this paper was in preparation, Silicon Valley's most prestigious venture-capital firm took a lead stake in the first startup company to specialize in 24/7 Linux technical support. It is generally expected that several Linux- and open-source-related IPOs will be floated before the end of 1999 – and that they will be quite successful.

Another very interesting development is the beginnings of systematic attempts to make task markets in open-source development. *SourceXchange* `<http://www.sourcexchange.com/process.html>` and *CoSource* `<http://www.cosource.com/>` represent slightly different ways of trying to apply a reverse-auction model to funding open-source development.

The overall trends are clear. We mentioned before IDC's projection that Linux will grow faster than all other operating systems *combined* through 2003. Apache is at 61% market share and rising steadily. Internet usage is exploding, and surveys such as the Internet Operating System Counter show that Linux and other open-source operating systems are already a plurality on Internet hosts and steadily gaining share against closed systems. The need to exploit open-source Internet infrastructure increasingly conditions not merely the design of other software but the business practices and software use/purchase patterns of every corporation there is. These trends, if anything, seem likely to accelerate.

15 Conclusion: Life After The Revolution

What will the world of software look like once the open-source transition is complete?

For purposes of examining this question, it will be helpful to sort kinds of software by the degree of completeness which the service they offer is describable by open technical standards, which is well correlated with how commoditized the underlying service has become.

This axis corresponds reasonably well to what people are normally thinking when they speak of 'applications' (not at all commoditized, weak or nonexistent open technical standards), 'infrastructure' (commoditized services, strong standards), and 'middleware' (partially commoditized, effective but incomplete technical standards). The paradigm cases today in 1999 would be a word processor (application), a TCP/IP stack (infrastructure), and a database engine (middleware).

The payoff analysis we did earlier suggests that infrastructure, applications, and middleware will be transformed in different ways and exhibit different equilibrium mixes of open and closed source. We recall that it also suggested the prevalence of open source in a particular software area would be a function of whether substantial network effects operate there, what the costs of failure are, and to what extent the software is a business-critical capital good.

We can venture some predictions if we apply these heuristics not to individual products but to entire segments of the software market. Here we go:

Infrastructure (the Internet, the Web, operating systems, and the lower levels of communications software that has to cross boundaries between competing parties) will be almost all open source, cooperatively maintained by user consortia and by for-profit distribution/service outfits with a role like that of Red Hat today.

Applications, on the other hand, will have the most tendency to remain closed. There will be circumstances under which the use value of an undisclosed algorithm or technology will be high enough (and the costs associated with unreliability will be low enough, and the risks associated with a supplier monopoly sufficiently tolerable) that consumers will continue to pay for closed software. This is likeliest to remain true in standalone vertical-market applications where network effects are weak. Our lumber-mill example earlier is one such; biometric identification software seems likeliest, of 1999's hot prospects, to be another.

Middleware (like databases, development tools, or the customized top ends of application protocol stacks) will be more mixed. Whether middleware categories tend to go closed or open seems likely to depend on the cost of failures, with higher cost creating market pressure for more openness.

To complete the picture, however, we need to notice that neither 'applications' nor 'middleware' are really stable categories. In 'Knowing When To Let Go' above we saw that individual software technologies seem to go through a natural life cycle from rationally closed to rationally open. The same logic applies in the large.

Applications tend to fall into middleware as standardized techniques develop and portions of the service becomes commoditized. (Databases, for example, became middleware after SQL decoupled front ends from engines.) As middleware services become commoditized, they will in turn tend to fall into the open-source infrastructure – a transition we're seeing in operating systems right now.

In a future that includes competition from open source, we can expect that the eventual destiny of any software technology will be to either die or become part of the open infrastructure itself. While this is hardly happy news for entrepreneurs who would like to collect rent on closed software forever, it does suggest that the software industry as a whole will *remain* entrepreneurial, with new niches constantly opening up at the upper (application) end and a limited lifespan for closed-IP monopolies as their product categories fall into infrastructure.

Finally, of course, this equilibrium will be great for the software consumer driving the process. More and more high-quality software will become permanently available to use and build on instead of being discontinued or locked in somebody's vault. Ceridwen's magic cauldron is, finally, too weak a metaphor – because food is consumed or decays, whereas software sources potentially last forever. The free market, in its widest libertarian sense including *all* un-coerced activity whether trade or gift, can produce perpetually increasing software wealth for everyone.

16 Bibliography and Acknowledgements

[CatB] The Cathedral and the Bazaar `<http://www.tuxedo.org/~esr/writings/cathedral-bazaar/>`

[HtN] Homesteading the Noosphere `<http://www.tuxedo.org/~esr/writings/homesteading/>`

[DL] De Marco and Lister, *Peopleware: Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6)

[SH] Shawn Hargreaves has written a good analysis of the applicability of open-source methods to games; *Playing the Open Source Game* `<http://www.talula.demon.co.uk/games.html>`.

Several stimulating discussions with David D. Friedman helped me refine the 'inverse commons' model of open-source cooperation. I am also indebted to Marshall van Alstyne for pointing out the conceptual importance of rivalrous information goods. Ray Ontko of the Indiana Group supplied helpful criticism. A good many people in audiences before whom I gave talks in the year leading up to June 1999 also helped; if

you're one of those, you know who you are.

It's yet another testimony to the open-source model that this paper was substantially improved by email feedback I received within days after release. Lloyd Wood pointed out the importance of open-source software being 'future-proof'. and Doug Dante reminded me of the 'Free the Future' business model. A question from Adam Moorhouse led to the discussion of exclusion payoffs. Lionel Oliviera Gresse gave me a better name for one of the business models. Stephen Turnbull slapped me silly about careless handling of free-rider effects.

17 Appendix: Why Closing Drivers Loses A Vendor Money

Manufacturers of peripheral hardware (Ethernet cards, disk controllers, video board and the like) have historically been reluctant to open up. This is changing now, with players like Adaptec and Cyclades beginning to routinely disclose specifications and driver source code for their boards. Nevertheless, therere's still resistance out there. In this appendix we attempt to dispel several of the economic misconceptions that sustain it.

If you are a hardware vendor, you may fear open-sourcing may reveal important things about how your hardware operates that competitors could copy, thus gaining an unfair competitive advantage. Back in the days of three- to five-year product cycles this was a valid argument. Today, the time your competitors' engineers would need to spend copying and understanding the copy is a substantial portion of the product cycle, time they are *not* spending innovating or differentiating their own product. Plagiarism is a trap you *want* your competitors to fall into.)

In any case, these details don't stay hidden for long these days. Hardware drivers are not like operating systems or applications; they're small, easy to disassemble, and easy to clone. Even teenage novice programmers can do this – and frequently do.

There are literally thousands of Linux and FreeBSD programmers out there with both the capability and the motivation to build drivers for a new board. For many classes of device that have relatively simple interfaces and well-known standards (such as disk controllers and network cards) these eager hackers can often prototype a driver as almost rapidly as your own shop could, even without documentation and without disassembling an existing driver.

Even for tricky devices like video cards, there is not much you can do to thwart a clever programmer armed with a disassembler. Costs are low and legal barriers are porous; Linux is an international effort and there is always a jurisdiction in which reverse-engineering will be legal.

For hard evidence that all these claims are true, examine the list of devices supported in the Linux kernel or in the driver subtrees of sites like *Metalab* `<http://metalab.unc.edu/pub/Linux/hardware/!INDEX.html>`, and notice the rate at which new ones are added.

The message? Keeping your driver secret looks attractive in the short run, but is probably bad strategy in the long run (certainly when you're competing with other vendors that are already open). But if you must do it, burn the code into an onboard ROM. Then publish the interface to it. Go open as much as possible, to build your market and demonstrate to potentional customers that you believe in your capacity to out-think and out-innovate competitors where it matters.

If you stay closed you will usually get the worst of all worlds – your secrets will get exposed, you won't get free development help, and you won't have wasted your stupider competition's time on cloning. Most importantly, you miss an avenue to widespread early adoption. A large and influential market (the people who manage the servers that run effectively all of the Internet and more than 17% of business data centers) will correctly write your company off as clueless and defensive because you didn't realize these things. Then they'll buy their boards from someone who did.

18 History

This is $Revision: 1.14 $.

Versions not described here are minor editorial and typo-fix updates.

20 May 1999, version 1.1 – draft.

18 Jun 1999, version 1.2 – first private review version.

24 Jun 1999, version 1.5 – first public release.

24 Jun 1999, version 1.6 – minor update; point at definition of 'hacker'.

24 Jun 1999, version 1.7 – minor update; clarify criterion (e).

24 Jun 1999, version 1.9 – 'future-proofing', the 'Free the Future' model, and a new section on exclusion payoffs.

24 Jun 1999, version 1.10 – better name for the 'Razor Blades' model.

25 Jun 1999, version 1.13 – corrected 13% claim about Netscape revenues; added better treatment of free-rider effects, corrected list of closed protocols.

25 Jun 1999, version 1.14 – added e-smith, inc.

9 Jul 1999, version 1.15 – new appendix on hardware drivers, and a better explanation of rivalrous goods due to Rich Morin.