The Emerging Economic Paradigm of Open Source

Bruce Perens <bruce@perens.com>
Senior Research Scientist, Open Source
Cyber Security Policy Research Institute, George Washington University.
Last edited: Wed Feb 16 06:22:06 PST 2005

Abstract

Open Source developers have, perhaps without conscious intent, created a new and surprisingly successful economic paradigm for the production of software. Examining that paradigm can answer a number of important questions.

It's not immediately obvious how Open Source[1] works economically. Probably the worst consequence of this lack of understanding is that many people don't understand how Open Source could be economically sustainable, and some may even feel that its potential negative effect upon the proprietary software industry is an overall economic detriment. Fortunately, if you look more deeply into the economic function of software in general, it's easy to establish that Open Source is both sustainable and of tremendous benefit to the overall economy.

Open Source can be explained entirely within the context of conventional open-market economics. Indeed, it turns out that it has much stronger ties to the phenomenon of capitalism than you may have appreciated.

A Strong Economic Foundation

In the early days of Open Source, its proponents did not fully understand its economics. Through our lack of understanding, we created the perception that Open Source's economic foundation was intangible. This led many people to feel that Open Source would not be sustainable over the long term and would be incapable of scaling to meet the market's need for new technology. It's important to correct that perception now. In *The Cathedral and the Bazaar*[16], Eric Raymond attempted to explain Open Source as a *gift economy*, a phenomenon of computer programmers having the leisure to do creative work not connected to their employment, and an artistic motivation to have their work appreciated. Raymond explains excellently how programmers behave within their own private subculture. The motivations he explored dominated during the genesis of Open Source and continue to be effective within a critically important group of Open Source contributors today.

Raymond did not attempt to explain why big companies like IBM are participating in Open Source, that had not yet started when he wrote. Open Source was just starting to attract serious attention from business, and had not yet become a significant economic phenomenon. Thus, *The Cathedral and the Bazaar* is not informed by the insight into Open Source's economics that is available today.

Unfortunately, many people have mistaken Raymond's early arguments as evidence of a weak economic foundation for Open Source. In Raymond's model, work is rewarded with an intangible return rather than a monetary one. Fortunately, it's easy to establish today that there *is* a strong monetary return for many Open Source developers. But that return is still not as direct as in proprietary software development. Thus, I'll ask you to follow a few more steps than you would in understanding the economics of proprietary software.

How Are You Going To Be The Next Microsoft?

Prospective software entrepreneurs are often asked: *how are you going to be the Next Microsoft?* And those who base a business upon Open Source are asked: how are you going to be the next Microsoft with *Free* software? But this isn't the right question if our goal is to achieve an *improvement* over the Microsoft model. It reflects the fact that most people have been thinking about software from an *extremely vendor-centric viewpoint.*

Whether or not we will admit it, most of us are very impressed with Microsoft's wealth and arrogance, and when we

think of producing software, we automatically think of Microsoft and the way *they* do it. But it turns out that the Microsoft model accounts for only a *minority* of the software that is made and used in business today. Around 30% of the software that is written is *sold as software*[2]. Most software is not sold at all. It is developed directly for its customer, by the customer's own employees or by consultants who bill for the *service of software creation* rather than for the end product. It's important to look at why that is the case, in order to understand the economics of Open Source.

Indication of an Unfulfilled Need

In February 1998, Eric Raymond and I formed the *Open Source Initiative.* We were standing on the shoulders of a giant: Richard Stallman's *Free Software* campaign had existed since 1983[17] and created much of what people call *Open Source* today. The combination of the Linux operating system kernel and Stallman's *GNU System* was becoming viable for business use. But the way that Stallman chose to present Free Software depends upon the *a priori* acceptance of the virtue of certain freedoms. Stallman is a programmer, and chose a philosophical presentation that appealed to programmers. In contrast, business people are pragmatists and are more impressed by economic benefit. Because Stallman's presentation limited his audience, his campaign had not been able to achieve the economic serendipity that is visible today[18]. Raymond and I chose to approach business people in a more pragmatic fashion, with the expectation that they would come to appreciate Stallman's philosophy once they'd seen its concrete benefits[3].

The first time the public heard of *Open Source* was in an announcement that I posted on *Slashdot* and a few mailing lists. It contained an introduction and the *Open Source Definition,* both a manifesto of Open Source and a definition of acceptable Open Source software licensing, which I'd created as a policy document of the *Debian Project* six months earlier. Eric Raymond edited *The Cathedral and the Bazaar,* then a year old, to replace the words *Free Software* with *Open Source.*

Some time after my announcement, a reporter asked Steve Ballmer, then president of Microsoft, if Microsoft would "Open Source" their Windows product. Ballmer was reported to have explained that Open Source wasn't just source code, it was a form of software licensing. The vice president of Microsoft had read my manifesto, and the press was asking him about it.

Manifestos from then-unknowns like myself and Raymond don't get that sort of recognition, unless for some reason the world is waiting for them. I submit that the reason was widespread dissatisfaction with the dominant economic paradigm of software creation at the time: Microsoft and its products. Microsoft was widely perceived as placing their own interests ahead of those of the customer. Their software was known to contain a large number of un-rectified bugs, and their desktop operating system was extremely crash-prone. They could get away with this because they essentially had no viable competition on the desktop.

People were searching for an alternative. If we were going to make software better, we'd have to find some other way to do it than what Microsoft was doing. Open Source created new economic relationships. There was a new peer-to-peer relationship between enterprise software customers, who were enabled to participate directly in the development of the Open Source software and thus became software developers for each other. There were new relationships between software vendors, who collaborated on Open Source projects while they competed elsewhere. And there were changes to the customer-vendor relationship: the customer could now participate with the vendor in software development. Because many vendors could access the same source code, and thus support the product, there was less lock-in and less exclusivity to the customer-to-vendor relationship. As a result of these changes, new suppliers arose, and existing suppliers began to substitute Open Source products for proprietary software.

The public sharing of the creation, ownership, and benefits of software was the antithesis of the Microsoft model. Open Source had already created a technically successful operating system and the first practical web servers and clients, and thus showed signs of being surprisingly effective. And so, the public buy-in to Open Source started out strong. It was followed by tremendous economic growth, with the Linux sub-sector alone showing an annual growth rate of 37 to 45 percent and predicted to be a *$35 Billion dollar market* by 2008[4].

The unusual acceptance and startling financial figures argue that Open Source must be the answer to some previously-unfulfilled need. Otherwise we would not have seen such wild, seemingly absurd phenomena:

- The hobby project of a student in his twenties, Linux, takes over enterprise computing.

- IBM, the epitome of conservative business, de-emphasizes its billion-dollar "AIX" operating system in favor of a product developed by a loose coalition of programmers with no financial motive in common, upon whom no corporate directive can be binding, whose leader has no power but the respect of others.
- Microsoft faces its first serious competitor in a decade: programmers who give away their work.

These events seem absurd: they certainly don't fit the common economic paradigm of technology production. A new economic phenomenon is operating, and to explain it we'll have to look more deeply into the economics of software production.

Looking for Economic Impact

Since Microsoft is the dominant model of a software manufacturer in most people's minds, and we're exploring the economics of software, let's ask: ***What is the greatest economic impact of Microsoft?***

Is Microsoft's greatest economic impact the wealth of Bill Gates? Of course not! A lot of people envy his wealth, but Bill's piece isn't the whole pie.

Is it the fact that they have amassed 40 Billion dollars in the bank, are the worlds largest software company, and employ 55,000 people in 85 countries? No, not that either.

Is the greatest economic effect of Microsoft the fact that they have enabled a great many businesses - their customers - to do business more efficiently, and to have businesses that they could not operate at all without the software that enables them? Yes, that is the biggest economic impact of Microsoft.

Microsoft is a tool-maker, and the effect of the tool-maker on the economy is tiny next to the economic effect of all of the people who are enabled by the maker's tools. The *secondary economic effect* caused by all of the people and businesses who use an enabling technology is greater than the primary economic effect of the dollars paid for that technology. And of course the same is true for Open Source software.

Considering the Economic Function of Software in a Business.

It's important to consider that most companies are not in the business of software manufacturing. They sell ships, shoes, services, everything under the sun. However, all but the smallest companies are *enabled* by software to some extent. Without software, their business would be less efficient, or impossible. Consider financial planning before computer spreadsheets, correspondence before email, and customer-to-computer interfaces when the most sophisticated input device in the customer's home was a touch-tone phone. Today, we *need* software to do business! Indeed, we need it so badly that even though most businesses don't sell software, any business of 50 people or greater is likely to employ a programmer, web designer, or a script-programming systems administrator[5]. For those businesses, software is essential enabling technology, rather than their product.

Enabling Technology vs. Business Differentiation

Enabling technology is essential to your business, but it's not what you sell. If you sell books, books are your *profit-center*, and software lives in a *cost-center* as an unavoidable cost of doing business. A polemicist has said *IT doesn't matter*, but what he is really telling us is that IT isn't your profit-center. You still need it to the extent that it enables your business.

There are two main forms of enabling, cost-center technology: *differentiating*, and *non-differentiating*. Differentiating technology is what makes your business more desirable to your customer than your competitor's business. For example, if you visit the Amazon.com web site and look for a book, Amazon will also tell you about *other* books that were purchased by people who bought the book you're interested in. And often the books it suggests *are* interesting enough that you will buy one of those books as well.[6] If you go to the Barnes and Noble site, they don't have that feature, and it's no surprise that Amazon sells more books online. So, for Amazon, the "recommendation" software is a business differentiator. Obviously, it would be a mistake to Open Source your business differentiators, because then your competitor's business might use them to become as desirable to the customer as your own business.

But in contrast, it wouldn't hurt your business for your competitor to understand how every bit of your non-differentiating software works. Indeed, that competitor might be the best collaborator you could have, if the partnership is limited to working on your non-differentiating software, because their needs are most similar to yours. This fact is demonstrated every day in the Open Source world, in which HP and IBM are partners in developing software that helps sell the systems of both vendors, and they remain fierce competitors at higher levels in the software stack where differentiation between them is possible and effective.

Perhaps 90% of the software in any business is non-differentiating. Much of it is referred to as *infrastructure,* the base upon which differentiating technology is built. In the category of infrastructure are such things operating systems, web servers, databases, Java application servers and other middle-ware, graphical user interface desktops, and the general tools used on GUI desktops such as web browsers, email clients, spreadsheets, word processing, and presentation applications. Any software that provides differentiating value to a non-software company is built on top of one or more of those infrastructure components.

An important indicator of whether software is differentiating is whether or not your competitor can get the same software. Neither Microsoft software nor Linux and Open Source can help you differentiate your business for long, because they are available to everyone. They differentiate against each other, they just don't differentiate *your* business. One or the other can save you money or make you more efficient, but in general they don't make your business more attractive to your customer.

Another important indicator of whether software is differentiating or not is whether the customer can see the software's effects. Your customer doesn't care what OS you run, unless your systems are crashed all of the time. She doesn't care whether you run Microsoft Office or OpenOffice. You might have good reasons to care, but the customer can't see them.

Thus, to make your business more desirable to customers, you should spend more on differentiating software that makes your business more desirable, and less on software that doesn't differentiate your business. Open Source is the key to spending less on non-differentiators, by distributing cost and risk that was formerly your company's alone across multiple collaborating companies.

Of course, you will need to take an honest look at what software in your business *is* differentiating and what isn't. And this turns out to be difficult. A lot of companies have *not invented here syndrome.* That's when managers and technical people aren't willing to consider the work of outsiders, because they don't believe that it could be as good as their own. "NIH" is an expensive disease: it will cause your employees to duplicate effort that is available elsewhere, instead of spending their time on the differentiating software that is most important to your business.

Participation in Open Source makes our software cost-centers more effective, because we can share cost and risk that we would otherwise be sustaining alone. What do we do with the money that we save? It becomes additional profit, or is spent in other areas where we need it more.

Economic Paradigms of Software Development

Since our businesses need software, we'll need to develop it somehow. The main economic paradigms for software development are:

- Retail.
- In-House and Contract.
- Efforts At Collaboration Without Open Source Licensing.
- Open Source.

Each is different from the other in:

- How they distribute the cost of development.
- How they distribute the risk of failure.
- Their efficiency in funding software development rather than overheads of the process.
- The degree to which others can be excluded from using the software.

These factors determine which paradigm is suitable for use in development of a particular piece of software. It's important to remember that the ultimate source of funds for software is the customer, *not* the vendor. The customer can direct its funds to operations using *any* of the economic paradigms to acquire the software it needs.

The Retail Paradigm

Retail is the paradigm most familiar to the average person, yet it is responsible for less than 30% of all software development[2]. In this paradigm, the cost of software development is usually borne by a single manufacturer. The manufacturer aims to recover those costs, plus profit, from sale of the finished product. Thus, the development expense is extremely front-loaded, with the entire development of a product necessary before the manufacturer can begin to recover expense. The risk of failure to produce a profitable product is borne entirely by the manufacturer.

Eventually, the cost of development is distributed to customers, if the product is successful in the market. Since the retail development paradigm can not directly distribute cost and risk until the product matures, it is often necessary for manufacturers to turn to investment markets as an external mechanism to distribute cost and risk. The manufacturer's need for outside capital is of long duration, since it could take years to develop the software and its market, and additional time before the cost of development can be amortized and it becomes possible to take a profit. Stock markets increase the liquidity of the investment by allowing investors to monetize the perception of the company's future potential, reflected in its stock price, rather than the company's actual earnings. Successful companies may re-invest profits in the development of subsequent software products rather than return to investment markets.

The overhead of the traditional brick-and-mortar retail sales paradigm is extremely high, with the result that less than 10% of the money paid for the software by the end-user actually goes into product marketing, software development, and documentation[7]. Micrsoft spent only 16.8% of its 2004 revenues on research and product development. The rest of Microsoft's income went into items that don't directly benefit the customer, such as the very expensive process of finding customers for Microsoft's products: overhead such as advertising, the design and manufacture of an attractive package that is discarded after the sale, payment to the retailer for shelf space upon which the product is displayed, sales staff, and profit. The 16.8% figure does not include the mark-ups of the retailer and wholesaler, which would bring the portion of the product price that represents software development well below 10%.

There is inefficiency on the purchasing side too, due to a mismatch between customer requirements and the software purchased. The customer often purchases software that turns out upon closer evaluation not to be usable for the application in question, or is not deployed. Often the customer can not recover the cost of this "shelfware". There are also losses because software companies fail or discontinue and de-support products that the purchaser has not yet amortized. Since generally even discontinued products do not have source available, they are left in an unsupportable state and the customer has little choice but to write off their investment.

Combining these risk factors, we can arrive at a conservative estimate that 50%[8] of purchased retail software is not used, or not fully and effectively deployed - the purchase is a failure.

If we multiply the less-than-10% efficiency of the retail paradigm at directing dollars toward software development by the 50% failure rate, we find that the efficiency of funding software development via retail software purchases is lower than 5%.

The most important implication of this extremely low efficiency is that the retail paradigm *can only be used economically to create products for a mass market.* A mass market will mask the inefficiency of the paradigm, because each customer can pay a relatively small cost compared to the cost of the software development. As an aggregate, they would still pay more than twenty times the cost of software development.

There are many important software products that simply can't be created using the retail paradigm, because they would not provide a large enough market to amortize both the cost of development and the large overheads of the paradigm. In addition, many new products that could eventually build a large market will not be considered by a manufacturer, because companies and investors can not be convinced that such a market would develop, or the risk of failure is too high. *This tends to damp innovation* within software manufacturers that make use of the retail software development paradigm.

One telling example of the failure of the retail paradigm to innovate is the fact that the most important innovation in the

last decade of global computer business, the web server and browser, had to be developed as an Open Source product at a federally-funded university research laboratory. None of the companies that could have completed the work were convinced that it would make money. Indeed, the only company that did invest significant funds to develop the web (Autodesk, by investing in Ted Nelson's Xanadu Project), chose not to complete the project because the revenue model that Xanadu foresaw for the web (payment to content producers for each word, with complicated tracking of derivative works and references) was too difficult to develop. The eventual leader in the development of a successful web, Tim Berners-Lee, did not see a need to develop any canonical revenue model, but left it to his users to figure out a means of making money from the web.

And as we've discussed, since a retail business must approach as many customers as possible in order to generate the highest possible profit, retail software is generally made available to everyone. Thus, it is difficult or impossible for retail software to differentiate the customer's business.

The In-House and Contract Development Paradigm

In-house development is done by the customer's own programmers, while contract development is done by an outside company, generally as a work-for-hire for the customer's exclusive use. In both cases, the programmers are generally paid for the act of writing software, rather than for the software as a product as would be the case in the retail paradigm.

Another form of contract development is extensive customization of an otherwise-available product by the vendor. For example, a medium-to-large company may purchase a standard web server environment and then pay its vendor for the customization of that environment to the customer's particular business needs.

The customer has excellent control of in-house and contract development, because the programmers won't get paid unless they do what the customer says. The customer can generally dictate whether the software will be exclusively for that customer or will be made accessible to others. Because the customer can control distribution, *this is an excellent paradigm for development of differentiating software.* Indeed, it may be the only paradigm that really works for development of software that would differentiate an end-user's non-software business.

In general, a contractor will *try* to use some of the work he does for one customer to leverage his business with other customers by re-selling work that's already been paid for. How successful he is at this depends on the contract terms and the honesty of the contractor. If your contractor is confident that he can re-sell work he does for you, he might charge you less and you might benefit from some distribution of cost and risk to the contractor and his next customer. However, this is not generally an appropriate scenario for differentiating software: if your contractor is selling your business differentiators, your business could soon be in trouble.

In exchange for the total control available through contract or in-house development, the customer generally sustains *all* of the cost and risk of the development. Because of that cost and risk, contract and in-house development may not be the most cost-effective means of developing non-differentiating software. If a customer is paying 100% of the cost to develop new software that duplicates the function of existing software that would be available to that customer for less, that's called *re-inventing the wheel* and it's a waste of money. If the customer can stand having the software distributed to others, and does not need absolute control over the development, the Open Source or retail paradigms may be more cost-effective for developing that particular piece of software.

In-house and contract development are reasonably efficient at directing most of each dollar spent toward software development. Their efficiency in this regard is 50% to 80%, in contrast to the 10% of a brick-and-mortar retail paradigm. The major sources of inefficiency are the costs of finding new customers for a contractor's business, the cost of retaining expertise that may not be fully utilized at all times, and the contractor's mark-up.

Many in-house and contract projects fail to produce working software that is deployed in the customer's business and meets the goals set for it. We can attribute to this paradigm a success rate of 50%[9], similar to the overall success rate we assigned to the retail paradigm.

Efforts At Collaboration Without Open Source Licensing

Consortia used to be the standard means of collaborating between companies upon software development. Closed consortium software development has a record of titanic failures. More recently, Billion-dollar closed consortia have a record of being replaced by more successful Open Source projects. Consider *Taligent* and *Monterey*, two consortia intended to create a replacement for Unix. Linux replaced them. Consider the *Common Desktop Environment* project, which has been replaced by the Open Source *GNOME* desktop at most of the companies that supported *CDE*.

A few years ago, the huge agricultural corporation *Cargill* founded a consortium with the stated intent of providing its partners with the benefits of Open Source while also providing secrecy and sharing the benefits of the software only with consortium members. This is called a *gated community*. Two years later, Cargill walked away from the project it founded. A closed consortium is simply the wrong structure for the development of *non-differentiating* software. It makes sense to throw the doors wide open when you don't have differentiation to protect. and admit members that can make a useful contribution even when they can't pitch in funding. A consortium costs more because there are fewer members to share cost and risk than with an Open Source project, yet there's much more structure and overhead than there would be for an equivalent Open Source project. Closed consortia generally are directed through *pay-for-say*, while technical merit would be the case for Open Source. With pay-for-say, a member can work to the detriment of the overall project when that is to the member's advantage. Consortium product planning often devolves into irresolvable arguments among the companies, because each has a different marketing idea and marketing arguments between companies are subjective and difficult to resolve.

Given the poor history of consortium development and, in contrast, the high rate of success for large Open Source projects carried out by the same groups of companies, it seems that the fairness imposed by Open Source licensing is an essential component of effective collaboration between a large number of parties with different interests.

The Open Source Paradigm

In the Open Source paradigm, multiple entities (individuals, companies, academic institutions, others) come together to develop a software product. Generally the initial development is done by a single entity as in the in-house and contract development paradigm, and the software is released to the public as soon as it is useful to others, generally before it would be considered a finished product and thus much earlier than a retail product would be released. Once the software is useful, other entities make use of it. Only when the software becomes useful to others does the Open Source paradigm work fully, because only then will other parties have an incentive to use the software. Once they are using the software, these other parties will have an incentive to extend the software to implement additional features that are of interest to them. This extension is performed by the customer's own employees or contractors under the customer's control.

The incremental cost of adding a feature is much smaller than the cost of the entire development. Parties that create modifications have an incentive to write them in such a way that they will be accepted by the other developers on the project and will be merged into the main body of source code that is shared by all developers. If this merge does not happen, the continuing cost of maintaining the added feature will be higher, since its developers must track changes to the main source code and maintain compatibility with that changing base.

Thus, Open Source tends to foster a community of developers who make contributions to a useful product. The cost and risk of developing the product is distributed among these developers, and any combination of them can carry on the project if others leave. Distribution of cost and risk begins as soon as the project is mature enough to build a community outside of its initial developer.

Open Source is developed directly by its end-users. For example, Apache web server features are added by the companies that need those features to operate their own web sites, or sometimes by contractors working for those companies.

The customers for a particular Open Source product generally identify themselves: they search for the product in a directory of Open Source software, and then they download and test the software. If tests are successful, they deploy it. Thus they gain a continuing interest in the product. At that point if they desire additional features to the product, they have an incentive to become co-developers of the software they are using.

The companies that join Open Source collaborations are seeking to use the software in a non-differentiating,

cost-center role. *It's not important to these companies that Open Source does not in itself produce a profit.* Their profit-centers are things other than software, and software is for them an enabling technology. In order to continue to operate their profit-centers, they must make some investment in their cost centers. In the case of differentiating software, they have little choice but to make use of the in-house or contract development paradigm, because they need to prevent their differentiators from falling into the hands of their competitors. For their non-differentiators, they have the choice of the retail or Open Source paradigms. But which is more efficient?

Because the customers are self-identifying, Open Source does not have the inefficiency of the retail paradigm, which must make use of advertising or other expensive mechanisms to seek customers [10]. It is at least as efficient in directing dollars to software development rather than overhead as the in-house or contract development paradigm.

Because of its greater efficiency in allocating resources to development rather than overhead, the Open Source paradigm can be used to develop products that would not support a mass market, and thus could not economically be developed within the retail paradigm.

Engineer Craig Small, a builder of network infrastructure, sums up the advantages for his customers:

> *Using Open Source cut way down on startup software costs. This was important because it was the way to get in the door with that particular customer. I was going to build the network management system from scratch, until I found a project that was close enough to what I needed. I could spend a fraction of the hours on the problem compared to what I would have needed to implement from zero.*
>
> *Expanding and customizing Open Source is by far easier to do, because the developers expect from the start that people will extend their work in ways that they haven't thought of, and thus they write in a style that is easier to extend. Proprietary software creators write as if nobody outside of their company will ever read their code. We took an Open Source project for entirely different hardware from ours, and easily got it to monitor our equipment.*
>
> *I have returned my extensions back to the Open Source project. Thus, I have confidence that if I leave the company, the extensions will still be looked after, at least by other members of the project.*
>
> *There is now a community of people with the same sort of technical needs, both developers and non-programmer users, that has sprung up around the software project that we all collaborated upon. We discuss how to better satisfy our needs, and pool ideas that we would never have had on our own.*
>
> *Proprietary software has costs that businesses have only started to consider. Organizations like the Business Software Alliance have raided businesses with federal marshals, demanding sufficient documentation that the company purchased a license for every computer on site. Much time and effort is spent making sure that companies comply with increasingly onerous licensing rules. Why can't I take that Windows XP off of my laptop and put it on the desktop legally? Open Source avoids that unproductive nonsense.*

Not all Open Source projects succeed. Most immature Open Source projects die on the vine or remain solo projects that use little in resources. But the expense of such projects is insignificant, because they die without first attracting significant customers or a developer community. More mature Open Source projects may die when something better comes along, but often it is possible to use the actual source code, data, and skills from one project in another, and thus the investment in development of the project is not lost.

The *cost-of-participation* in mature Open Source projects is very different from the costs of retail or in-house and contract development. The major expense is the time-cost of employee participation. This figure is a combination of the personnel cost of software evaluation, the personnel or contractor resources spent to adapt existing Open Source to customer needs and to support the Open Source for internal users, and the possibility that time will be invested into software that is eventually replaced due to a failure to track customer needs[19]. The maximum cost for Open Source would come when there is no community other than the customer: this would be similar to the cost of contract or in-house development, in which one customer supports the entire expense. The actual cost will be lower depending on the number of active participants and the work required. The lost investment is generally personnel time. Taking this into account, the Open Source paradigm yields an economic efficiency at least as great as the in-house and contract development paradigm, and much greater than the retail paradigm.

Summary of Software Production Business Methods

The Open Source paradigm has several significant economic advantages over the retail or in-house and contract software production paradigms: it combines efficiency in allocating resources to software development with distribution of cost and risk better than either of the other dominant paradigms of software development. It is amenable to the development of products that are untenable under the retail paradigm because there is no mass market. Open Source can distribute the cost and risk of projects, while the in-house and contract development paradigm would focus all cost and risk on one party. Open Source distributes cost and risk directly, rather than requiring the use of investment markets. Distribution of cost and risk starts much earlier than it would in the case of the retail paradigm. Open Source gives the customer complete control over customization of the product. However, it does *not* give the customer control over who has access to the product. Thus, Open Source is generally *not* a good mechanism for developing differentiating software.

Businesses that require non-differentiating software (really, all businesses) would be well advised to shift some development to Open Source and reap greater economic efficiency. Participation in Open Source development communities should be an important part of any businesses overall strategy to shift development dollars from non-differentiating to differentiating software.

Paradigm	Efficiency	Failure Rate	Distributes Cost	Distributes Risk	Protects Customer Differentiation	Protects Vendor Differentiation	Required Market Size
Retail	less than 10%	50%	Late, sometime after sales start.	No.	No.	Yes.	Mass market 100,000 and up, specialized markets less.
In-House and Contract	60% to 80%	50%	No.	No.	Yes.	Maybe.	1
Consortium and Non-Open-Source Collaboration	60% to 80%	Perhaps 90%, unacceptably high.	Yes.	Yes.	Maybe.	Maybe.	5 and up.
Open Source	60% to 100%	50%	Early, during development.	Yes	No.	No.	5 and up.

Who Contributes to Open Source, and How Do They Fund That?

There are several main sorts of Open Source contributor:

- Volunteers.
- Linux distribution companies.
- Companies with a single Open Source program as their main product.
- Companies for whom Open Source software enables sales of hardware or solutions.
- Service Businesses.
- End-user businesses and their contractors.
- Government.
- Academics and scientific researchers.

Each of these sorts of entity works differently within the economics of Open Source software, drawing from a different source to fund their contributions. We'll explore each one.

Volunteers

I (Bruce Perens) was an example of this sort of contributor from 1993 through 1998, when I worked extensively on Open Source development that had no connection to my employment.

The 2002 FLOSS study[11] reported that the creation of Open Source was, at that time, *primarily a hobby activity!* But their participation of individuals without an immediate financial motivation in Open Source software development is serious enough to transcend the term *hobby*. Perhaps *volunteer* is a better description, because of the obvious professionality of their work [12], and the fact that their motivation is not pecuniary in nature.

Eric Raymond proposed that the volunteer's motivation is mainly intangible, and that a particularly important motivator is participation in a community of respect in which developers are recognized by their peers for the quality and innovation in their work. The FLOSS study surveyed Open Source developers regarding their motivation, found that many of them are motivated by technical curiosity and the desire to learn. I feel that their motivation is similar to that of an artist: just as a painter wants people to appreciate his paintings, a programmer wants to have users who appreciate his software.

There is a continuing transition from volunteer to professional, as Open Source is used increasingly in companies that then become motivated to participate in its continued development. Former volunteers are gaining employment in organizations that support their Open Source work on company time. Open Source supporters within companies come "out of the closet" to become internal experts as their employers express an interest in Open Source.

Linux Distribution Companies

Red Hat and *Novell* are well-known for distributing Linux-based systems. But, surprisingly, businesses dedicated to selling Open Source software as their main product *do not* create the majority of Open Source software. They act mainly as integrators of the work of others. They do a lot of maintenance work in order to eliminate bugs for their paying customers, and they do original Open Source software development where they feel this is necessary to enable their product or a new market. The Linux distributions are for the most part medium-sized companies, and their employees represent a small but active component of the overall population of Open Source contributors. Sometimes they inflate their impact in marketing communications. For example, one Linux distribution promotes that it employs 300 programmers, but a much smaller proportion make regular contributions to Open Source software. The majority seem to be doing "sales engineering" or working on internal solutions for customers.

Indeed, the economics of Open Source seem to work *worst* for Linux distributions. When you Open Source your business differentiators, your competitor can appropriate them and reduce your business differentiation. That is the quandary of the Linux distributions: their product is mostly Open Source, their customers want it to stay that way, and they struggle to differentiate a product that the customer knows is available elsewhere without charge.

Linux distributions originally tried to deal with the differentiation problem with the first generation of Open Source vendor business plans, which Eric Raymond explains in *The Cathedral and the Bazaar*. These models mostly coupled some other product to Open Source software as the money-maker: service on the Open Source software, or a proprietary software addition to the Open Source component. The Open Source software would be enabling technology for a business with some other component driving its revenues, but the money-making component would be very closely coupled with the Open Source software. But unfortunately, services alone were not enough to make the distributions profitable during Linux's early-adopter period.

Today we are experiencing the second generation of Open Source business plans. Some Linux distributions are attempting to imitate proprietary software. Behind their costly box of software or per-seat license is a product that the customer could acquire without charge through other channels. Several strategies are combined to make this work: the development of a brand that is perceived to hold more trust or value than the naked software. The vendor invests resources into certification by proprietary application vendors, who each want to support only a few distributions[14]. Customers who need support for a proprietary product on Linux thus have an incentive to pay for that vendor's version of Linux. Some support services are sequestered, such as security problem reports or bug patches, so that they will only be available to people who have purchased the costly box and per-seat licenses. If the customer loads the free software on more systems than he's paid for and the vendor finds out, the customer will be penalized by the revocation of his

service contract and the withdrawal of security information critical to the continued operation of the software. Perhaps the best name for this business model is *proprietary Open Source*, in which services are offered but the business is operated in the proprietary box-software model. This business model is essentially antagonistic to the volunteers who have created much of the Open Source software. They weren't out to develop another Microsoft, and they resent the sequestering of service information on *their* software. It is in conflict with the spirit, but not the letter, of Open Source licensing such as the GPL[15]. In general, volunteers help the companies they approve of. As the internal experts on Open Source for their employers, they recommend the companies that they approve of. And thus there will be significant challenges to the *proprietary Open Source model.*

It's important to emphasize that not all Linux distributions engage in *proprietary Open Source*, and that proprietary Open Source is *not* the dominant business model for Open Source software development. The vast majority of contributors to Open Source belong to the other categories examined here.

The money used to purchase Linux distribution "seats" and their associated services comes from enterprise IT department cost-centers.

Companies With a Single Open Source Program As Their Main Product

This sort of company can be divided into several categories:

- Mixed Open Source and proprietary licensing model.
- A core Open Source program with proprietary software accessories.
- Pure Open Source plus services model.

We'll explore each category.

Mixed Open Source and Proprietary Licensing Model.

Examples of this sort of company are MySQL AB and Sleepycat Software, which produce databases, and Trolltech, which produces the *Qt* graphical user interface toolkit. This sort of company vends the same software under two different licenses: an Open Source license and a commercial license.

The Open Source license chosen, often the GPL, includes a "poison pill" meant to make production of proprietary derivative works commercially untenable. Since the GPL requires that all derivative works must be distributed in source-code form under the GPL or a GPL-compatible license, this would remove the business differentiation from any software to which it is applied.

To escape the poison pill and preserve business differentiation, the producer of a proprietary derivative work must purchase a commercial license for the same product. This provides the producer of the Open Source software a direct revenue capture for unit sales of software to creators of proprietary derivative works.

This model works only for software that will be combined into derivative works, such as a software library. It generally isn't usable for applications.

The future viability of this model is in question, because a programmer can make a "server" of a software library and export all of its functionality to another program without creating something that would be considered a derivative work under copyright law or the definitions in Open Source licenses. In Unix parlance, servers are referred to as "daemons", and thus the practice of embedding software in a daemon in order to avoid creating a derivative work is called *daemonization.* It is possible that a future Open Source license could restrict this practice.

Today, it is possible to make use of the MySQL database server in a proprietary application without a commercial license, by using the MySQL database engine as a server (its usual mode) and making use of a special variant of the MySQL client library that is under appropriate licensing terms for proprietary applications. Of course, MySQL AB doesn't support that client library.

This sort of company generally supplements its commercial licensing revenue with additional revenue from training and software development services.

The main customer of this sort of company is the enterprise user, in the case of MySQL. Trolltech's main customers are embedded device developers and software application developers. The funds used to purchase these products come from cost-centers within IT and software development departments.

A Core Open Source Program With Proprietary Software Accessories

Eric Raymond calls this model *Widget Frosting* and discusses it in detail. Sendmail Inc. is an example of this sort of participant. Sendmail Inc. has created a constellation of proprietary products around the Open Source sendmail email server. This funds ongoing maintenance of the core Open Source product. Some of the Linux distribution companies also intend to operate using this model, and some of IBM's business is under this model: for example their sales of the proprietary *DB2* database on Linux. This sort of company essentially operates as if it were a proprietary software business. The funds used to purchase its products come from IT department cost-centers.

Specialization In One Open Source Program Plus Services Model

This model was proposed to be an important one in Eric Raymond's paper, but has not performed as well as was expected so far. Many Open Source developers supplement their income by providing services upon the software that they develop, and there are a number of new companies pursuing this model - some with sizable venture funding. Some of these new companies seem to be operating a *certification model*, servicing a particular version of the software that they certify, perhaps in the manner of the *proprietary open source* business model at some of the Linux distribution companies.

Some small and medium-sized companies have been able to produce sustainable revenue while developing Open Source software as their major business focus and providing services on that Open Source as their only profit-center. But many have failed, some spectacularly like VA Linux Systems and Linuxcare. Companies that would be support customers seem to prefer to do their Open Source support inside, or through a vendor with whom they already have an existing relationship or whom can service more than one program. There may also have been an early-adopter problem. Early-adopters in general do not want the hand-holding of a service company. Time will tell if this model can perform effectively.

The funds for services provided by this sort of company come from IT department cost-centers.

Hardware Vendors

Examples of this sort of company are IBM and HP. Hardware is a great product to sell along with Open Source software. It costs a penny to copy software, but you can't copy a loaf of bread without a pound of flour. Until we have the *Star Trek* "replicator"[13], hardware is a difficult-to-copy product. Allowing the customer to know something of hardware internals doesn't necessarily remove all of its business differentiation, as might be the case for software. The hardware manufacturers that participate in Open Source development do so to enable sales of their hardware products. Hardware is useless without software, and specifically computers are useless without the operating system that interfaces the computer hardware to software applications. Open Source developers seem to be better at systems programming than any other form of programming, so far, and the *Linux* operating system kernel is now as good as, or better than, many proprietary operating systems for similar hardware. Hardware manufacturers formerly spent billions on proprietary operating systems that, for them, were always enabling technology rather than a profit-center. The margins were in the hardware itself. Many of these manufacturers have eagerly embraced Linux because it allows them to distribute the cost and risk of the operating system among multiple companies, has a cost-efficiency greater than that of similar proprietary operating systems, and is in general desirable to the customer.

Hardware companies are good at producing Open Source because there isn't nearly so much conflict between Open Source and differentiation for them as there might be in a software company. Making software products Open Source enables sales, doesn't reduce their hardware-based business differentiators, and thus does not threaten their profit center.

Generally this sort of company supplements its hardware income with services, and sometimes training.

The funds for hardware and services provided by this sort of company come from IT department cost-center hardware

budgets.

End-User Businesses and Their Contractors

eBay is an example of this sort of contributor. Many companies make use of Open Source software in their own operations. Web applications are particularly important, but there are many others. And these companies make up a significant portion of the Open Source contributors. In general the contributions come from internal software support and development staff, or contractors supporting the company, who modify the Open Source software to fit the company's needs.

What is most interesting about this sort of contributor is that they are the main customers of essentially all of the other sorts of companies that contribute to Open Source software development. Their cost-center dollars fund the work of the other sorts of companies analyzed here. And they appear to be putting some of those dollars directly into Open Source software, either through use of their own staff or via contractors working directly for them, instead of going through the traditional intermediates.

But why should this sort of company should do work outside of its core competency? Core competency is a property of individuals more than of companies. Thus, you should consider not whether an Open Source software project is the focus of your business, but whether you can operate your business most economically by making the Open Source participation a focus of some staff members. The main advantage for your company is the reduction of cost and risk, and an improvement of the degree of control that you have over your software. Consider today the degree to which software controls your business. Do you control the software?

Service Businesses

A number of service businesses create solutions by integrating multiple Open Source programs with "glue" software specialized for the particular customer. Other businesses provide service for a collection of Open Source programs. This sort of business participates in the development and maintenance of many Open Source programs, but perhaps not intensively in any one. In general, the motivation of the end-user businesses that employs this sort of contractor dominates that of the contractor itself, and thus these businesses should be considered under *End-User Businesses And Their Contractors*, above. Some businesses vend a web service under the application service provider (ASP) model, building mostly upon Open Source software. There is a loop-hole in many Open Source licenses (particularly the GPL as it exists today) that protects differentiation of ASP businesses. The requirement to provide source code doesn't trigger until the software is distributed, and an ASP need only perform the software for the customer, rather than distribute it. Since this is widely regarded as a flaw in the GPL, it may not persist in the next version of that license.

Government

Government's use of Open Source is similar to the way that business approaches Open Source for its cost centers. However, government is expected to function for the benefit of the citizens and is not generally thought of as having profit-centers of its own. Rather, it provides services that enable economic and social activities.

Government contracting should not provide a commercial advantage to a particular vendor outside of the direct revenue from the products or services purchased. Government should especially not lock itself to a particular vendor after the contract term because of switching costs. It's poor policy for government to lock its vendors or citizens into use of a particular vendor's product for communicating with the government, as this would provide an inappropriate advantage to the vendor. All vendors can make use of Open Source components with appropriate licensing, and thus can facilitate e-government to make use of Open Source for government-to-citizen, government-to-business and government-to-government interfaces.

All vendors can make use of Open Source components with appropriate licensing. Such use can assure that the software interface to government facilities is an *open* interface that can be utilized by all vendors. Thus, it can facilitate e-government to make use of Open Source for government-to-citizen, government-to-business and government-to-government interfaces.

Government carries out some activities solely for the public benefit, and can carry out Open Source development in

this capacity. This is generally done through research funding.

Academics and Scientific Researchers

Academic research projects have historically been large contributors to Open Source, and mostly graduate rather than undergraduate work. In general, an undergraduate class does not provide sufficient time for a student to come up to speed on an Open Source project and make a contribution. In contrast, graduate research projects are often years in duration. A large body of Open Source software came out of the BSD (Berkeley System Distribution) project funded by the U.S. Department of Defense. Additional academic research projects produce more software daily.

There are vigorous Open Source communities involved in scientific research. In science, the maxim is *publish or perish*, and Open Source fits that well. To be considered valid, scientific research must be capable of being duplicated. If an experiment doesn't turn out the same way when performed by another scientist, that may mean that the research is erroneous. These days software makes up a large part of many experiments, and a human-language description of the experiment may not convey every detail of how it has been performed. If the researchers can share code, outsiders can examine their software for errors and duplicate their experiments more easily.

The community of scientists researching a particular field of study is small. Retail software would not be effective in developing such specialized software. Open Source is the best way for scientists to share the cost of developing software to support their research.

Proprietary software supporters have tried to turn back the tide of academic contributions to Open Source by forging new partnerships between college research projects and proprietary software companies. This trend is especially disturbing when publicly-funded research work results in patents that are transferred to the proprietary software company partners, since it is likely that those patents will be prosecuted against the very taxpayers who funded their creation. In general, publicly-funded work should be maximally utilizable by the public that funded it. DARPA (the U.S. Department of Defense research grant organization) and the University of California recognized this when they applied the *BSD License* to the pioneering extensions of Unix created at the university. That licensing allowed any Open Source or proprietary use of the software.

Some research work is performed by unsalaried students. When they *are* salaried, the funds for their software development generally come from grants. Governments are the largest provider of such grants, a secondary source is philanthropic, and some funds come from industry partners.

Summary of Contributors To Open Source Projects

The largest contributor to Open Source development today may still be the volunteer. There is a conflict between business differentiation and Open Source that makes its economics work worst for the sort of business that would generally fund the same sort of development in the case of proprietary software. Businesses without such a conflict are more effective at funding Open Source development. Thus, hardware manufacturers have taken a large role, and end-user businesses are taking an increasing role.

Type	**Example**	**Conflict between Open Source and Business Differentiation?**	**Differentiation Protection Mechanism**	**Source of Funding.**	**Commercially Effective**
Volunteer	Bruce Perens from 1993 to 1998.	No	n/a, doesn't need one.	Donates own time.	n/a, motivation is non-commercial.
Academic Contributors	University research projects.	No	Government, philanthropic grants.	Paid to do research, not the result.	Yes.

Linux Distribution	Red Hat	Yes	Branding, certification, sequesters service data.	Per-box or seat sales to IT cost-centers.	Too early to tell.
Companies with Single Open Source Program as Product.	MySQL, Trolltech, Sleepycat Software	No	Dual-licensing, "poison pill" in the Open Source license protects differentiation in the case of commercial users, provides incentive for commercial license purchases.	Per-box or seat sales to IT cost-centers.	Yes.
Core Open Source Product with Proprietary Additions	Sendmail Inc.	No	Differentiating products are proprietary.	Proprietary software sales to IT cost-centers.	Dubious.
Specialization in Open Source Program Plus Services	?	No	Service, not the program, is what is differentiated.	Services sales to IT cost-centers.	Dubious.
Service Businesses	?	No	Service, not the program, is what is differentiated.	Service sales to IT cost-centers.	Yes.
Hardware Manufacturer	IBM, HP	No	Hardware can't be copied for pennies as software can be.	IT cost-center hardware budgets.	Yes.
End-User Business	eBay	No	Software lives in a cost-center, enables sales of profit-center products.	Their own IT cost-centers.	Yes, from savings alone.

How Open Source Does Product Marketing

A common objection to Open Source is the perception that Open Source development will not be well focused or targeted. People are used to development that is directed by one company in an extremely focused manner, driven by a product marketing process. Some mature Open Source projects do perform their product marketing the way a company would. But while a company generally would develop an overall strategy that drives all of its software development, there is no global planning authority for Open Source, and no overarching strategy followed by all Open Source developers. However, it's an error to ask for such a thing: you can't compare Open Source to a company, it's an entire industry. A central planning authority for an entire industry would indicate something other than an open market. The product marketing for the global Open Source community operates in the way that a capitalist nation operates its economy, rather than the way a company plans its products.

The paradigm by which Open Source does product marketing can be described as *a massively-parallel drunkard's walk filtered by a Darwinistic process.* First, very many people all over the world develop whatever they want, going in any direction they wish without any central coordination. Out of this process come many potential products that would not interest more than a few other people, some products that are interesting to at least fifty people, and a few products that are interesting to millions of people.

Fifty people who want the same thing, but are geographically distributed all around the world, can form a viable software development team via the Internet. The spare time of fifty people who are otherwise employed turns out to be sufficient resource to enable the development of large and complex software products, and of course the size of the development team increases as the product matures and becomes of interest to more people. Thus, by leveraging upon the excellent collaboration that is possible when using Open Source licensing, it is possible to initiate and successfully

carry out projects that would otherwise be beyond the capability of any of the participating entities.

But how do 50 people who haven't met work together to form a viable software product? Part of the reason this works so well is that software is extremely modular by nature, and thus many people can work on different segments of the software, almost autonomously, if they can come to agreement about how the pieces fit together. A good example of this is the *Debian* GNU/Linux Distribution. This system includes more than 16,000 software packages maintained by over 1000 volunteer developers in many nations around the world. When these packages are combined, the result is a reliable and well-integrated system. That system has supervised experiments while in orbit on the Space Shuttle, and has a user community second in size only to Red Hat (yes, it's bigger than Novell).

But how can we develop products when everyone has the freedom to go their own way and there isn't a real boss? This seems odd to business people, until they realize that this is exactly how capitalism works in democratic countries. In the broader economy of a capitalistic nation, many companies set out to develop products without any central guidance from the government. They all compete with each other, and some products succeed and build a market while other products fail. Loose collaborations arise between companies for the purpose of building markets for new products. The role of the government, the only entity that conceivably could guide the economy, is in general limited to the injection or removal of capital from the economy, tax incentives and funding programs, and the enforcement of laws designed to create fair markets.

Most sensible people have accepted that a "sure thing" is rare in gambling or stock-picking, and yet they expect marketing departments to make reliable forecasts in guiding new product development. Marketing departments have no crystal ball. If a sufficient number of self-guided developers are available, a drunkard's walk strategy will outperform conventional product marketing. The massively-parallel drunkard's walk tries many different paths, and thus has a higher probability of accessing a successful path. The Darwinistic filtering is what recognizes that a particular path is successful.

Most business people are adamant that they want to live in a free market economy, and that the government should not take a strong guiding role in determining what products will be produced and how they are made. They say this because they understand that the free market is more likely to produce good products and a healthy economy than any central planning process. It should be no surprise to them, then, that the open-market-like paradigm of Open Source can sometimes do a better job than the central control of marketing departments and management at creating desirable products.

Is Open Source Self-Sustaining?

Many people have trouble understanding how Open Source could be self-sustaining if it does not operate according to the retail development paradigm. What pays for such software? It is funded directly or indirectly as a cost-center item by the companies that need it. Those companies need a great deal of cost-center, non-differentiating software. They are willing to invest in its creation through the Open Source paradigm because it allows them to spend less on their cost centers by distributing the cost and risk among many collaborators, and makes more efficient use of their software dollar than the retail paradigm. This is essentially the same source of funding that pays for proprietary software. It's important to remember that the software manufacturer isn't the ultimate source of funds: the customer is.

What is the Economic Impact of Open Source?

We defined the major economic impact of Microsoft as:

> The fact that they have enabled a great many businesses - their customers - to do business more efficiently, and to have businesses that they could not operate at all without the software that enables them.

The same definition applies to Open Source.

It is a fact that Open Source enables a majority of web servers today, a majority of email deliveries, and many other businesses, organizations, and personal pursuits. Thus, its economic impact must already be numbered in many tens of Billions of dollars.

Any improvement in technology that permits business to function more efficiently means the economy runs more efficiently. In this case, Open Source enables business to spend less on software and to have better quality and more control over its software. The money that is saved on software doesn't disappear, the people who save it spend it on things that are more important to them.

What Does All Of This Mean To Software Producers?

This discussion has been mostly from the perspective of the software-using business, who ultimately pays for software development, rather than the software-producing one. But what does this mean to software producers?

Does your business really sell finished software? Some businesses sell the act of software creation rather than the software itself. This is the case, in general, for consulting and "solutions" companies. Your customers will be willing to pay for the creation of Open Source software.

In the case of a business that wishes to produce software for sale, rather than sell the service of programming or training, Open Source will be a difficult product to monetize.

It may be that Open Source eventually causes a reduction in demand the for proprietary software. This would not, however, reduce the demand for programmers, because the demand for software in general would not decrease. The displaced proprietary programmers would move to an organization that can produce Open Source software in an economically successful manner.

It is possible that programmers who move to a less entrepreneurial setting could earn less. However, surplus profits in proprietary software companies have historically been distributed to management in much larger portions than to staff. It is the rare programmer that has been able to profit from a stock incentive windfall from a company whose products are predominantly software offered for sale.

What Effect Does The Free-Rider Problem Have Upon Open Source?

The *Free-Rider Problem* is familiar in economics. What do you do about people who take advantage of a product or service without providing any return to the provider of that product or service? Do you have a mechanism to prevent free-riders?

All Open Source users start out as free-riders. They download and try the software, and perhaps deploy it, and do not generally consider contributing to that software's development until they are already using it and desire an additional feature.

If they desire an additional feature, they may implement that feature themselves rather than pay one of the initial developers. At this point, are they still free riders? No. Businesses that join an Open Source project as developers contribute some software to the product, and all of those businesses derive an economic benefit from making use of the software in a cost-center of their business. The interests of the various developers are generally similar because they have self-selected a particular software product as one useful to them. The contributions of any one developer are generally of use to other developers.

There are developers that are not motivated by the desire to provide software for a business cost-center. These are individuals whose motivations are primarily artistic, and scientific researchers.

Volunteers derive emotional fulfillment from having users for their software, just as artists derive fulfillment from having others appreciate their paintings. For volunteers, users provide an intangible benefit which the volunteer desires. Thus, those users should *not* be considered free-riders.

Companies that place importance in a particular Open Source product tend to hire developers who have already gained stature as a developer of that product. Thus, individuals who have started with no pecuniary interest in the Open Source project tend to find employment with an organization that does have such an interest. And thus individuals who participate in Open Source development often reap an economic gain from that participation. This is another reason that users should not be considered free-riders by these individuals.

Scientific researchers have their own paradigm of constant exchange of knowledge similar to that in the Open Source community, because science advances most rapidly when discoveries are made known to other scientists who can add their intuitions to them. Scientists gain fulfillment from the publication of their work, because this increases their stature among other scientists and in general determines the success of their careers. Scientists routinely use Open Source as a means of publishing the software component of their work. In addition, scientists are motivated by the desire to be of benefit to society, and wish to see other people benefit from the use of software developed through scientific research. Thus, to scientific participants, users are of benefit and should *not* be considered free-riders.

And finally, users are the people whom we recruit to become more active participants in an Open Source project, just as retailers try to recruit the general public to buy their products. Invariably we are successful with some of them.

There is some question regarding whether the free-rider problem is as significant in the case of software as it is for other sorts of products, and whether it applies to Open Source at all. A free-rider on a bus uses the scarce resource of a seat, so that a potential paying rider could be denied a chance to ride the bus. A free-rider who has bootlegged a copy of Microsoft Windows may or may not diminish the market for paid copies of Windows, but does not use any scarce resource that would exclude other Windows users. A free-rider using Open Source does not diminish a market or use any scarce resource.

All of this leads me to believe that the free-rider problem is not a significant detriment to Open Source development.

If Open Source Works, Why Don't We All Build Our Own Cars?

The Open Source paradigm works well for many products where the major value of the product is *its design.* It's most successfully been used to date to produce software, an encyclopedia, and integrated circuit designs.

The integrated circuit designs are programmed into a field-programmable gate-array, a special device that can be programmed with a design and will then immediately behave as a circuit with that design. This is an example of a field in which hardware is very much like software.

However, most things are not software. It only takes a cent's worth of resources to make a copy of a piece of software, but it takes a pound of flour to make a loaf of bread. Someone has to farm the wheat and grind it into flour, and those efforts have to be paid for.

Automobiles, of course, are much more complex than bread, and it takes a great many physical processes with expensive tooling to manufacture them. Consider that to make an electric motor, one must mine and refine metal, then draw wire, roll sheet metal, cast and machine bearings, and then assemble all of these pieces into very precise forms. It should be no wonder that it takes an entire economy to manufacture an automobile, while a single individual may produce an important software product.

When the day comes that we can make complex physical products by producing their designs and directing a machine to manufacture them from easily-available materials and electricity[13], the economy will change radically. Today, we are limited to producing individual parts with computer-controlled milling machines, a slow and dirty process that still requires manual intervention. A healthy Open Source community has evolved around such machines, and we are starting to see them share part designs. But the science of computer-controlled manufacturing will have to improve tremendously before we can have "Open Source cars".

Summary

Open Source is self-sustaining, with an economic foundation that operates in a capitalistic manner. It does not require any sort of voodoo economics to explain. It is an extremely beneficial component of a free-market economy, because of the very many people and businesses that it enables to make their own economic contribution. It is more efficient than other economic paradigms of software development for producing software that does not differentiate its user's business. Non-differentiating software makes up the lion's share of all software in a business, and businesses would be well advised to pursue Open Source collaborations for producing such software.

Topics for Further Development

- Economists would be interested in a discussion of Open Source as a public good.

About the Speaker

Bruce Perens is one of the founders of the Open Source movement in software, and the creator of the *Open Source Definition,* the manifesto of Open Source. He is founder or co-founder of the *Linux Standard Base,* the standardization project of Linux, the *UserLinux* project, and *No-Code International.*

Perens currently sits on the board of directors of *Open Source Risk Management,* an insurance company covering the risks of Open Source software and *Software in the Public Interest,* the non-profit that supports the Debian Project. He is a significant stockholder in *Progeny Linux Systems,* which was funded through his VC firm *Linux Capital Group.*

Perens is Senior Research Scientist for Open Source with the Cyber Security Policy Research Institute of George Washington University. He is series editor of the *Bruce Perens' Open Source Series* with Prentice Hall PTR publishers, which has published 13 books with all of their text under Open Source licenses.

Perens operates a strategy consulting firm, *Perens LLC,* which specializes in issues related to Linux and Open Source software.

Perens formerly served as Senior Global Strategist for Linux and Open Source with the Hewlett Packard Company, and as invited expert on the World Wide Web Consortium's Patent Policy Board. He was Project Leader for the *Debian GNU/Linux Distribution,* and wrote the *Debian Social Contract,* a portion of which later became *The Open Source Definition.*

Acknowledgements

The generous support of NTT Docomo Labs USA is making this and other articles possible. Thanks to The Cyber Security Policy Research Institute of George Washington University, and Tony Stanco there, for granting me an affiliation with their prestigious institution. Thanks to Prentice Hall PTR and executive editor Mark Taub for publishing *Bruce Perens' Open Source Series,* currently numbered at 13 books, in which this article will eventually find a home. Martin Fink, now Vice President for Linux at HP, was the person to name *proprietary open source,* even if he's being forced to accept it now. Martin also provided my introduction to management of large corporations as my manager at HP, and has taken lots of guff for and from yours truly. The Debian developers never cease to amaze me with their erudition. They were the first reviewers of this document, and contributed tremendously to its quality. Participants were: Bill Allombert, Ben Armstrong, Jeff Breidenbach, Andrew M.A. Cater, Tim Cutts, Ryan Golbeck, Joshua Kwan, Alexander List, Andrew McMillan, Raul Miller, Nick Phillips, M.J. Ray, Craig Small, Andrew Stribbehill, David N. Welton, Florian M. Weps, Matt Whipp, Matthew Wilcox. Other reviewers were Kael Goodman, Mark Shuttleworth.

Footnotes

1. For the purposes of this paper, *Open Source* and *Free Software* mean the same thing. There are philosophical differences between the two groups, but the great majority of software to which the definition of *Open Source* applies would also fit the *Free Software Foundation's* definition of *Free Software.*

   It is important to note that when I created the document that later became the *Open Source Definition,* as a policy document for the Debian Project, the FSF had not created its own definition of Free Software. At that time, Richard Stallman commented in personal email that my document *was* a good definition of Free Software.

   Stallman would not agree with my suggestion that a business might be better off not assigning a Free Software license to software if that license would reduce the differentiating value of the software to that business. Thus I am using the nomenclature *Open Source* in order to avoid diverting Stallman's agenda for *Free Software.*

2. A report on the size of the U.S. software market is at http://web.ita.doc.gov/ITI%5CitiHome.nsf/AutonomyView/87200518f179196c85256cc40077ede1 . In that report, "Packaged Software" represents 24.6% of the industry. All other industry sectors that represent computer programming, including all of Computer Programming Services, Computer Integrated Systems Design, Computer Processing and Data Preparation and Processing, Information Retrieval Services, Computer Facilities Management Services, and some sub-categories of Software Publishing represent the remainder.

   However, that report does not account for the software produced solely for company-internal use by programmers employed by companies that have a non-software product. I conservatively estimate the population of internal programmers to be equal to that of contract ones, who are listed under Computer Programming Services (19% of market).

3. To some extent, Raymond deprecated Stallman when speaking for Open Source, when he might simply have promoted to the different audience of Open Source without any conflict with Stallman and his Free Software campaign. In my opinion, this unfortunate outcome is attributable more to a difference in personalities than philosophy.

4. IDC Software Consulting: *The Linux Marketplace - Moving from Niche to Mainstream.* Prepared for OSDL. http://www.osdl.org/docs/linux_market_overview.pdf

5. Bureau of labor statistics reports for 2002 show 2,153,000 people hold programming-related jobs in the United States: 499,000 are Computer Programmers, 675,000 are Computer Software Engineers, 979,000 are Computer Systems Analysts, Database Administrators, and Computer Scientists. This does not count 758,000 jobs held by Computer Support Specialists and Systems Administrators, who generally perform at least some script programming, and Computer Operators, who held 182,000 jobs. http://www.bls.gov/oco/ocos110.htm, http://www.bls.gov/oco/ocos267.htm, http://www.bls.gov/oco/ocos042.htm, http://stats.bls.gov/oco/ocos268.htm, http://stats.bls.gov/oco/ocos128.htm

6. Ironically, the Amazon "recommendation" feature is the topic of a patent-infringement suit brought by Cendant against Amazon.

7. Microsoft's 10-Q report to the SEC for September 30, 2004 shows that only 16.8% of their revenue goes to research and product development. "Research and development expenses include payroll, employee benefits, stock-based compensation and other head-count-related costs associated with product development. Research and development expenses also include third-party development and programming costs, localization costs incurred to translate software for international markets, and the amortization of purchased software code and services content". Of course, this report does not show the markups of wholesalers and retailers, which would bring the software-development-per-dollar figure below 10%. The report is at http://microsoft.shareholder.com/redesign/EdgarDetail.asp?CIK=789019&FID=1193125-04-189841&SID=04-00

8. *A terrible thing to waste,* article on "Shelfware", CFOEurope.com: http://www.cfoeurope.com/displayStory.cfm/1739037, *CRM Software or CRM Shelfware?,* CNET News.com, http://news.com.com/2100-1012-990880.html

9. *Collaborating on Project Success,* Standish Group, http://www.softwaremag.com/archive/2001feb/CollaborativeMgt.html, *Immature Software Acquisition Processes Increase FAA System Acquisition Risks* http://ntl.bts.gov/lib/000/200/234/ai97047.pdf

10. We do, however, see advertising by various entities connected to Open Source. This generally concerns hardware or services, rather than the software. In some cases the Open Source teams make use of low or no-cost advertising venues such as web banner advertising to draw new users to their products and thus expand their development communities. A recent ad in *The New York Times* for *Mozilla Firefox* is notable as it is motivated by the political goal of maintaining the open-ness of the web and reducing dominance by Microsoft over the browser. The advertising was funded by charitable contributions from parties that in general have no pecuniary interest in *Firefox.*

11. http://www.infonomics.nl/FLOSS/report/

12. Studies by Stanford and CMU researchers show that the Linux 2.6 kernel has 0.17 bugs per 1000 lines of code while non-Open-Source commercial software generally scores 20-30 bugs per 1000 lines. The studies are reported in *Wired* magazine at http://www.wired.com/news/linux/0,1411,66022,00.html, I do not presently have a reference to the actual studies.

13. The *replicator* is a fictional device that manufactures many different physical objects on demand, presumably from designs stored on a computer. In the *Star Trek: The Next Generation* television series, a character walked up to the device and gave it a voice command: *tea, earl gray, hot.* A cup and the tea were manufactured instantly.

14. The Linux Standard Base was meant to make it possible for proprietary application vendors to support all standard-compliant Linux distributions. It has not been commercially effective at this so far.

15. Analysis of pro-bono counsel, *Free Software Foundation*, in private communication.

16. http://www.catb.org/~esr/writings/cathedral-bazaar/ Raymond followed *The Cathedral and the Bazaar* with two additional papers, *Homesteading the Noosphere* (1998) and *The Magic Cauldron* (1999), which continue his discussion of the economics of Open Source. They can be found at the same link. His last revision to *The Magic Cauldron* appeared in 2002.
17. The GNU project was announced in September 1983, see http://www.gnu.org/gnu/initial-announcement.html, although the project didn't actually start until January 1984, and it was not until sometime later that the Free Software Foundation was formed around it. See http://www.gnu.org/gnu/gnu-history.html for an overview of the GNU project.
18. There was also the fact that the GNU project had developed and integrated a lot of software, but had not been able to produce its own operating system kernel. This left the door open for Linus Torvalds to do the last part of the project. Generally an operating system is referred to by the name of its kernel, dispite the fact that it contains many components other than the kernel. This caused an injustice to Stallman that persists today. Although Stallman did a great deal of the work that made *Linux* possible, Torvalds' team of kernel contributors was not closely allied with Stallman, and announcements of Linux were not attributed to FSF.

    Linux is in truth only the kernel. The rest of what comes in a Linux distribution is what Stallman intended to be the *GNU System*, and Stallman should be credited with its concept, with directly developing the C compiler (surely as important as the kernel) and the Emacs editor, and with inspiring the development of much of the rest of the system.

19. There is a cost-of-participation for retail software as well. Part of that is discussed here as the *shelfware* problem. The after-purchase cost is not discussed, but includes such things as the cost of documenting problems to the vendor (often an all-day task for a programmer) and going through service escalation procedures with the vendor's service staff until those problems get attention.