Setting Up Shop: The Business of Open-Source Software

Frank Hecker
hecker@hecker.org

Originally published May 1998, revised 20 June 2000
Revision 0.8
DRAFT

For the latest version of this paper see <http://www.hecker.org/writings/setting-up-shop.html>. This is a work in progress; I welcome comments and criticisms and will respond to them as I have time, either individually or in future versions. Note that this draft version of the paper is incomplete; I'll release the remaining sections in future versions of the paper as I complete them.

Disclaimer: This document represents my personal opinions only.

Abstract

Commercial software companies face many challenges in growing their business in today's fast-moving and competitive industry environment. Recently many people have proposed the use of an open-source development model as one possible way to address those challenges. This document investigates the business of commercial open-source software, including why a company might adopt an open-source model, how open-source licensing works, what business models might be usable for commercial open-source products, what special considerations apply to commercial products released as open source, and how various objections relating to open source might be answered. The target audience is commercial software and hardware companies and individual software developers considering some sort of open-source strategy or just curious about how such a strategy might work.

Executive summary

If your company is in the software business either directly (as an independent software vendor or ISV) or indirectly (for example, you produce software as a key component of other goods or services) then it's likely that you face several challenges in growing your business. Among them may be

- to continue to create new products and bring in new incremental revenue;
- to improve new product quality at first release;
- to do a better job of sustaining engineering in supporting current and older releases while still driving innovation in new releases;
- to more effectively recruit third-party developer and integrator support for your company's products and platform; and (last but by no means least)
- to motivate and retain your current employees, and to recruit and energize the next generation of your company's employees.

I believe these challenges are interconnected, for two reasons. First, most if not all of them are functions of constrained resources: few companies have enough people, money, or time to do everything that needs doing, especially when competing against larger companies with greater resources. Second, I believe that your company has at least one possible strategy available that may help address all these issues together -- turning some (or in exceptional cases even all) of your software products into "open-source" products, i.e., products for which you make the source code freely available under liberal licensing terms and with no licensing fees, so that others may take that software, make changes to it, and use or distribute the resulting modified versions as they see fit.

You've no doubt read about Netscape's recent release of the source code for Netscape Communicator; you may also have heard about earlier open-source projects like the Linux operating system kernel and read papers like Eric Raymond's "The Cathedral and the Bazaar" that make a case for open-source development within an extended developer community as a way to create better software. In this paper I discuss why and how a commercial company might build a business or extend an existing business through the creation and distribution of open-source software -- in other words, how to "set up shop" in the "bazaar."

I first discuss particular problems and potential opportunities which might lead you to consider doing something different from "business as usual." I then explore the business proposition for open source software, starting from the proposition that your company is in business to provide customers with something of value and to receive money (or other things of value) in return. Moving to open source for a product can potentially provide better value to your customers, including in particular the ability for your customers or third parties to improve that product through bug fixes and product enhancements. In this way you can create better and more reliable products that likely will more truly reflect your customers' requirements.

However the real key to building a business on open-source software is going beyond that to provide greater value to your customers than your competitors can, and ultimately turning that increased value into increased revenue and profits for your company. In the traditional software business model your company provides all (or almost all) of the value to customers, and you realize revenues and profits in return for that value through traditional software

license fees. In an open-source business model much of the value provided to customers will not be provided solely by you, but rather by other developers who are attracted to working on your open-source products and who will thus help augment your resources as opposed to your competitors'. These "outside" developers may be motivated by the prospect of working with software that solves important problems for them and for others, by the possibility of future gain providing related services and creating related products, by the opportunity to increase their own personal knowledge, or by the ego satisfaction of building an enhanced reputation among their peers.

Thus a significant part of your potential success will depend on the work of others who will be working "for free," i.e., open source developers who will contribute their work back to your company and to the developer community at large without demanding or receiving any money or other tangible payment in return. However open-source developers will not (and should not) do this work unless you treat them fairly. This is in part a function of your company's attitudes and actions towards developers working with its products, but is also formalized in the company's choice of an open-source license, specifying the terms and conditions under which the company's open-source products can be used, modified, and redistributed.

There have been several standard license agreements published for use with open-source software. All of them share some common features, most notably making software "free" to users both in terms of being no-cost and in terms of minimizing restrictions on use and redistribution; these features are explainable as necessary for developers to feel fairly treated. If possible you should use one of the existing open-source licenses, or modify one of those licenses to meet your needs; some licenses will work better than others for particular business models. Possible license choices include

- no license at all (i.e., releasing software into the public domain)
- licenses like the BSD License that place relatively few constraints on what a developer may do (including creating proprietary versions of open source products)
- the GNU General Public License (GPL) and variants which attempt to constrain developers from "hoarding" code, i.e., making changes to open-source products and then not contributing those changes back to the developer community, but rather attempting to keep them proprietary for commercial purposes or other reasons
- the Artistic License, which modifies various of the more controversial aspects of the GPL
- the Mozilla Public License (MozPL) and variants (including the Netscape Public License or NPL) which go further than the BSD and similar licenses in discouraging "software hoarding" but which still allow developers to create proprietary add-ons if they wish

Since you can't use traditional software licenses and license fees with

open-source software, you must find other ways of generating revenues and profits based on the value you are providing to customers. Doing this successfully requires selecting a suitable business model and executing it well; I discuss several business models potentially usable by companies creating or leveraging open-source software products (names and descriptions for the first four models are courtesy of OpenSource.Org):

- "Support Sellers," in which revenue comes from media distribution, branding, training, consulting, custom development, and post-sales support instead of traditional software licensing fees
- "Loss Leader," where a no-charge open-source product is used as a loss leader for traditional commercial software
- "Widget Frosting," for companies that are in business primarily to sell hardware but which use the open-source model for enabling software such as driver and interface code
- "Accessorizing," for companies which distribute books, computer hardware and other physical items associated with and supportive of open-source software
- "Service Enabler," where open-source software is created and distributed primarily to support access to revenue-generating on-line services
- "Brand Licensing," in which a company charges other companies for the right to use its brand names and trademarks in creating derivative products
- "Sell It, Free It," where a company's software products start out their product life cycle as traditional commercial products and then are continually converted to open-source products when appropriate
- "Software Franchising," a combination of several of the preceding models (in particular "Brand Licensing" and "Support Sellers") in which a company authorizes others to use its brand names and trademarks in creating associated organizations doing custom software development in particular geographic areas or vertical markets, and supplies franchises with training and related services in exchange for franchise fees of some sort

I also discuss another class of business models, which I call "hybrid" models, in which the constraints surrounding open source are relaxed in one way or another. For example, a company might use both traditional licensing and open-source-like licensing "side by side" for the same product, differentiating between different users (e.g., for-profit organizations vs. not-for-profit organizations vs. individuals) and/or between different types of use (e.g., intranet vs. extranet use, use on one platform vs. another, and so on); alternately, a company might license source widely to any and all users, and even allow "evaluation" licensing at no charge, but still charge "right-to-modify" license fees and restrict re-distribution of modified versions in some way. Although these business models are not true open-source models based on a strict definition of the term, they may be workable models for some companies in some cases.

Beyond selecting an appropriate license and business model, what else might your company have to do in order to implement an open-source strategy? I discuss various issues your company will have to consider and address when converting one or more products to open source, including

- Code-sharing. For historical reasons your open source product may share a common source code base with other of your products that will remain proprietary. If so, you will need to make sure that open source development can proceed without complicating your other internal development efforts; this may require both special considerations in licensing and appropriate modularization to enforce a clean separation between your open source and your proprietary source.
- Third-party technology. Your product may include technology licensed from third parties and present in your existing source code base. Such third-party code will need to be treated specially in order to create a releasable open-source product; typical options are to remove the code entirely, to seek permission for inclusion of third-party code (perhaps under some special arrangement), or to replace such code with open-source code providing equivalent or similar functionality. The presence of third-party technology also can influence your choice of open-source license.
- Code sanitization. You will need to do certain other things in order to ensure your source code is ready for public distribution; these include removal or revision of inappropriate language, comments intended for internal viewing only, and so on.
- Export control. If your company is located in the US and your product contains security and cryptographic code, in order to obtain export approval you will almost certainly have to modify your product for release as open source, including removal of all cryptographic code and security code that calls that code.
- Product development processes. Releasing source for a product will in almost all cases significantly change the way you do product development; indeed, without such changes you will not realize most of the benefits of an open-source strategy. This problem is discussed in more detail elsewhere, but I discuss a few key things you need to know, including in particular the advisability of forming a dedicated team to be responsible for your open-source efforts, the need to provide infrastructure for external developers (newsgroups, source code repositories with revision control, special bug reporting systems, etc.), and the desirability of having your own developers be "customers" of that infrastructure, in principle no different than any other developers.

Finally, I address some frequently-raised objections to converting commercial software products to an open-source model:

- "If you make a product open source, does that mean you're no longer committed to it?" Even after you convert a product to open source, you can

and should continue to have ultimate responsibility for it in some sense; in particular, you can not only exercise influence and oversight over the evolution of the product, you can and should continue to release an "official" binary version of the product yourself, packaged for easy installation by end users and with full QA testing, product support, and branding.

- "But customers don't really want to have source code and can't take advantage of it anyway. Why would they prefer an open-source product over a pre-packaged binary product?" Some customers do indeed want source code and find it of value. Other customers who prefer not to deal with source code can use the pre-packaged "official" product binaries you'll continue to produce. Both sets of customers benefit from the improved product quality and enhancements resulting from open-source development.
- "Wouldn't this lead to fragmentation of the product into incompatible versions?" This is one of the most common objections, but there is ample reason to believe that this will not be an issue, both because of the particular dynamics of open-source development as they have evolved over time, and also because your company can exert a positive influence to minimize the possibilities of this happening. In particular, the open-source developer community has unwritten but historically-effective rules that assign control of an open-source project, including the right to designate "official" versions, to a single entity (an individual, an informal group, or a formal organization); as the original developer of the product you are the natural candidate to be that entity, assuming you do the necessary things to live up to your assigned role.
- "What about the risk to customer from 'rogue' versions?" Again, this has not proved to be a problem with open-source products, both because of public review and also because there is typically a single place (i.e., the original vendor) to get an "official" version that has undergone additional review and testing.
- "What about providing technical support to customers with modified versions?" If you wish you can choose not to support modified versions of your product (i.e., versions built from a different code base than your "official" releases), leaving such support to the general open-source community or to other companies providing such support as a business. You can also contract out to external developers to provide such support as part of your own support offerings.
- "Wouldn't customers be concerned with your open source code 'tainting' theirs if they used it for their own projects?" To allay such concerns you can use a non-tainting open-source license, such as a BSD-style license, or a license like the Mozilla Public License that does not taint other code that calls open-source code using a defined API.
- "What about embarrassing things that people might discover in our source code?" Some "bad" things in your source code you definitely *want* to be exposed, most notably bugs; open-source development increases the chances that both major and minor bugs will be found and fixed. Other

things in your source code that might cause embarrassment, such as inappropriate language in comments, can and should be removed during preparation of the source for public release.

- "Wouldn't releasing your source code expose confidential plans and strategies to competitors?" Moving to an open-source model implies to some degree sharing your product strategies with external developers and letting them influence those strategies; this implies sharing those strategies with competitors as well. However at the same time this can result in greater public support for your strategies (because the outside world is helping you create those strategies), helping to counter those of your competitors. Also, releasing source does not imply or require making all internal information publicly available; in particular you can continue to keep a close hold on confidential details of business plans and the like.
- "Wouldn't people just use our code and our expertise without our getting anything in return?" This objection is similar to the original objections to companies like Netscape allowing downloading of software over the Internet at a time when vendors saw "software piracy" as a major cause of lost revenue. The answer to the objection is also similar: that the benefits of a properly executed open-source strategy can well outweigh the costs. To take but one example, every bug discovered and fixed by a developer "out there" directly saves you money otherwise required for QA and software maintenance "in here;" it also increases the value of your software as a reliable product and can indirectly lead to increased revenues for other products and services associated with that software.
- "What about competitors who might try to 'hijack' an open-source product for their own purposes?" Open-source licenses such as the GPL and MozPL can be used to enforce public disclosure and sharing of source code modifications. In the open-source world competitors must play by the same rules as everyone else, and those rules have evolved to minimize the chances of one individual or organization exercising undue advantage.

When looked at closely there is nothing strange or magical about open-source development from a business point of view; it should neither be shunned as impractical nor embraced as a panacea. There is no one single model you must follow, and it is not an "all or nothing" proposition. Open source is simply a new (really, newly popularized) way of developing, distributing, and licensing software, and for companies which understand the economic, cultural, and political factors that go into implementing an effective open-source strategy, the open-source model offers the promise of helping your business better survive and thrive in an increasingly demanding environment.

Making the business case for open source

Open-source software and open-source development projects have existed for many years now under the general term "free software". The word "free" has traditionally led commercial software vendors to think "no revenue," and

customers of those companies to think "no support;" thus free software was assumed to be irrelevant to the commercial world, with free software developers often cast as idealistic and naive. Similarly, the writings of some free software advocates (in tone if not in substance) have seemed to portray commercial software companies as interested only in short-term profits at the expense of the long-term interests of users and the software development community as a whole.

Perceptions about free software (pro and con) have often been more absolute than reality would warrant. To take but one example, even Richard Stallman, often thought the free software advocate most hostile to commercial considerations, did not argue in his "GNU Manifesto" that software development should always be a unpaid or non-profit activity; rather he proposed the abandonment of for-profit business models based on treating software as intellectual property, arguing instead that for-profit software development be done as a professional service. This happens to be one of the open-source business models discussed below. Similarly many commercial software companies have used software originating in the free software community as the basis for commercial products, and in some cases have contributed to the development of free software through donations of money, hardware, or their employees' time.

Events like the growing success of the Linux operating system kernel and associated GNU and other utilities (developed under the free software model) and Netscape's recent release of Communicator source code have focused more public attention on the potential importance of free software to businesses both as users of software and as for-profit producers of it; these events have also led to the recasting of "free" software as "open-source" software, a term that emphasizes more the importance of making source code to software freely available, and also may (at least subliminally) remind people that a company can choose to make source code freely available and still serve its own business interests as a for-profit organization. (Note however that open-source software is still "free" software in the sense that no license fees are charged for use or redistribution of binaries or source code, as well as in the sense that users are free to modify the source and create and distribute derivative works.)

Still, this strategy may seem counter-intuitive or even self-destructive; it goes against years of tried and true commercial software practices. However these are unusual times in the commercial software industry, and may call for unusual measures. In considering a major change in strategy like moving to open source, it may help to consider an historical analogy: When Netscape first made the Navigator web browser available for unrestricted download over the Internet, many saw this as flying in the face of conventional wisdom for the commercial software business, and questioned how Netscape could possibly make money "giving the software away." Now of course this strategy is seen in retrospect as a successful innovation that was a key factor (if not *the* key factor) in Netscape's rapid growth, and rare is the software company today that does

not emulate this strategy in one way or another. It's possible that the current interest in open source may signal another such industry-changing event.

I've written this paper primarily for commercial software vendors considering whether to take a product open-source; however it may also be of interest to their customers considering why and how they might use such products, and to open-source developers curious about the business side of open-source products. My goal is to address some of the practical realities of the business of open-source software, writing as someone who lobbied for an open-source strategy within a major commercial software company (Netscape Communications Corporation) and was a close observer of and sometime participant in the implementation of that strategy.

Why should you care?

Adopting an open-source business model can be a major decision for a commercial company. Thus before doing anything else you should ask yourself a simple question: "Do I really need or want to do this?" If your software business is doing fine, and you have neither problems you need to solve nor potential opportunities you want to take advantage of, then I really can't recommend you consider an open-source strategy; adopting such a strategy incurs definite costs in money, time, and effort, and I see no point in urging it on you if there's no benefit to be gained.

However it may be that you do in fact have either some present problems or some future opportunities you are concerned about; here are some common areas where commercial software companies might have "pain" or see "gain:"

- Product line evolution. You want or need to continue to create new products (or new add-ons to existing products) and bring in new incremental revenue.
- Product quality. You want or need to improve new product quality at first release.
- Product maintenance. You want or need to do a better job of sustaining engineering in supporting current and older releases while still driving innovation in new releases.
- Third-party recruitment. You want or need to more effectively recruit third-party developer and integrator support for your company's products and platform, so that others can help increase the value of your product to customers, and thus in effect help "sell" the product for you.
- Employee retention and recruitment. You want or need to better motivate and retain your current employees, using something beyond standard incentives like stock options and beer busts. At the same time you want to recruit and energize the next generation of your company's employees, providing them interesting and exciting opportunities they likely won't be able to find elsewhere.

I believe all of these areas can potentially benefit from an open-source strategy of some type. As discussed below, properly implemented an open-source strategy can allow you to bring more developer resources to bear on the task of product development, QA, and maintenance; this can not only increase product functionality and quality but can also increase the value of the product as a platform for third-party developers and channel partners. At the same time an open-source strategy can potentially motivate your own developers by providing them more opportunity to have their skills recognized and their efforts make a difference in the wider world.

If these and other potential benefits are of interest to you, please read on and see if and how this might apply to you. Because open source is such a radical departure from traditional commercial software practice, I think it will help to spend some time first going over the basic ideas behind an open-source software business; to start, let's go back to square one and consider the basic principles behind *any* successful business.

Why be in business?

As Netscape CEO Jim Barksdale likes to say, there is more to being in business than simply making money; in particular, Barksdale often quotes Peter Drucker's dictum that the purpose of any business is *to create customers and keep them*. *Customers* are organizations or individuals who see value in the products and services that you provide, and who pay you good money in order to obtain them. You *create* customers by convincing people that you can provide something of value to them and then persuading them to give you money in exchange for that value. You *keep* customers by providing them not just one-time but continuing value, sufficient that they will continue to give you money in exchange for the value they continue to receive.

For customers, value is not necessarily associated with particular features of a product. (In fact, products with the same features offered by two different companies may have different value for customers in each case.) More generally, a product has value for customers to the extent that *it helps them solve problems*; the more problems it can help them solve or the more important the problem it helps them solve, the more value the product has. Customers may solve these problems with the help of the product's actual features (as web browsers like NCSA Mosaic and Netscape Navigator originally helped solve the problem of accessing the Internet without complicated utilities) or they may solve their problems with the help of other product attributes (as a company's brand name and reputation for quality might help solve the problem of making a "safe" buying decision).

To remain successful over time a company can attempt to add additional value to its product line by adding new features to existing products, adding new products, adding new ways to use the products (e.g., services associated with the products), and so on. For your company to continue to grow you need to find

new ways to provide value to customers, and new ways to convert that value into money.

But your competitors also can add new features, new products, and new ways to use them, and in many cases they have greater resources with which to do this. If you want to be successful in a competitive market you can attempt to do so by offering particular product value earlier than your competitors, by matching or exceeding the value that your competitors' resources create for their own products, and by creating value for your products and services in ways that your competitors cannot easily duplicate.

In a time of rapid innovation competing successfully means continually having to pursue advantages in relative value deriving primarily from product features (advantages which are often temporary), combined with more strategic attempts to alter the competitive balance in ways with which your competitors are less prepared to cope. Seen in this light a move to more of an open-source business model is *not* simply a tactical move to solve a particular business problem, but rather can be part of an overall strategy to *change the rules* of competition in your market space and perhaps in the software industry as a whole.

Giving away the 'crown jewels'?

Let's assume for now that you agree with my comments above, and that you're at least willing to consider that open source offers a possible way for your company to better compete in the market. However I suspect that at some point in your experience with the concept of open source you, someone with whom you've been talking, or a news article you've read has raised the question "Why would a software company want to give away its crown jewels to the world?"

The idea of source code as "crown jewels" is simply a commonly-used metaphor, and metaphors can be powerful forces for good or ill depending on the context. In this case the metaphor implies that a software company's source code is a precious resource that must be protected at all costs from anyone outside the company who would seek to obtain it or (mis)use it. Thus most commercial software companies release their products' source code only in very special situations, and do so only when accompanied with an array of legal arrangements (including non-disclosure agreements and very tightly-worded terms and conditions regarding redistribution) and a large amount of money changing hands.

But arguably possessing source code *in and of itself* is irrelevant. Source code acquires true commercial value only if you can use it to create products and/or services that you sell for money, and you can typically do this successfully only in the context of a company with a known brand name and a sustainable market position. (For example, even if I had the source code to Windows it still wouldn't make me Bill Gates.) Similarly, releasing source code for one of your products

would *not* necessarily mean that other companies could duplicate your successes or even adversely affect your business. Those successes are typically a result of much more than the product features implemented in your source code; other factors might include the celebrity of and trust placed in your brand name, the quality and reputation of your services, the effectiveness of your sales and distribution channels, and so on. Those factors could still be very much influenced by you, even in the absence of complete control over the source code itself.

What is the value of source code?

Once you've accepted the possibility that releasing source code for one or more of your products might be a component of your overall business strategy, two questions then arise: How does distributing source code create value for customers, and how can your company convert that value into revenue and profits (i.e., money)?

First note that by "source code" for a given product I mean the underlying programming language statements (and related information, e.g., make files, error message files, etc.) which can be read by humans, modified to make changes and additions, and then used to build a functional version of the product or another derivative product reusing some or all of the same source code. For purposes of this discussion we assume that for a given product you would distribute all relevant source code. (In actual practice this may not be possible in all cases, as discussed below; for example, initially you might have to distribute some components of a product in binary form only, or not distribute certain components at all.)

Much software has traditionally been distributed in source code form, and there are several commonly-accepted ways in which having source code for a product can directly increase the value of that product for a customer by helping them solve any of a number of problems:

- The customer can better protect their investment in a software product in the event that the software vendor goes out of business or decides to discontinue a product that is critical to the customer's operations. (Source code escrow provisions in contracts provide similar protection, but typically only for the case where a vendor goes out of business. Source code escrow also does not provide the additional indirect benefits discussed below.)
- The customer can better understand how the software works in the event that the vendor's documentation is incomplete or confusing.
- The customer can look for and correct potential security flaws in the product that might otherwise adversely impact the customer's operations.
- The customer can fix bugs themselves if the vendor is unable or unwilling to do so.
- The customer can subject the product to independent audit for proper Year

2000 support and other requirements and can correct any problems found.
- The customer can port the software to new operating systems and/or hardware platforms not otherwise supported by the vendor.
- The customer can use the source code to create customized versions of the original software product, extended and improved versions, or whole new applications, thus avoiding the need to write those applications "from scratch."

Distributing source code for a product can also more indirectly increase the product's value to a customer:

- Writers and trainers not associated with the software vendor can create more complete and correct documentation and training material, because they do not have to rely often on the vendor's (sometimes incomplete) original documentation. This makes learning about the product easier, so that customers can more easily and cheaply train their end users and developers.
- Consultants and systems integrators can become more familiar with the technology and techniques underlying the product, and can become more skilled in implementing systems based on or incorporating it. This creates a wider base of people able to implement complex applications using the product, and this in turn means that customers can implement systems more cheaply (because there is more competition among people possessing the minimum necessary expertise) and/or can implement more functional systems for the same price (because the most skilled implementors possess greater expertise than they might otherwise).
- Third parties can find bugs and security flaws and create fixes for them; the fixes can then be incorporated into the original product. This makes the product more bug-free and secure than it might otherwise be, increasing the product's value by helping customers lower software operational and maintenance costs.
- Other software suppliers can reuse the source code (assuming that this is permitted and even encouraged) to create modifications and add-ons to the original product that provide features that the original product does not provide. Customers can use these new products to implement more functional systems than they might be otherwise, and this increases the overall value of the original product considered as a platform on which to build.

These ways to increase value are again quite familiar to anyone acquainted with products such as the Linux operating system kernel, the GNU C++ compiler, or the Apache web server that are normally distributed with source code (or for which source code is available separately). Together they help to make open-source products *whole products*, in the sense used by Geoffrey Moore (in *Crossing the Chasm* and *Inside the Tornado*) and others: products accompanied with all the additional things necessary to ensure that the value experienced by the customer matches the value promised by the vendor. These things include

support, documentation, training, third-party consulting and system integration, a thriving developer community to use the product in innovative ways and create add-ons to the product, a set of de jure or de facto standards upon which the vendor and others can build, and so on.

If you compete with larger companies, this is exactly where their greater resources and experience serve them well; to compete successfully with such companies as time goes on, you must not only create products that are more functional than your competitors' in a generic sense, but must also find ways to approach or match your competitors' whole product value, ways that do not require your company to take on the entire burden of doing so itself (given that your own resources do not equal those of your competitor).

However, for as a commercial company it is not enough just to increase the value of your products; you must also find sustainable ways to have customers reflect that value back to you in the form of increased revenues and profits. Before we consider possible ways to do that as part of an open-source strategy, it's worth reviewing how commercial software companies have traditionally approached this problem, and how this approach might need to change in the open-source case.

Building a business model

The standard software business model

Today the conventional way to get revenue in exchange for the value of a proprietary software product is to sell the customer the *right to use* the software product as opposed to selling them actual ownership of the product (as they might own a physical object); even a customer buying shrinkwrapped software sold in stores is technically buying a right-to-use license rather than possession of the software itself. The legal basis for right-to-use licensing is that the specific code and techniques underlying the product are considered to be intellectual property of the developer protected using legal constructs such as copyrights and patents; as the owner of that intellectual property the developer can control its distribution and use through legally-binding and enforceable contracts, and can charge a fee to individuals and organizations wishing to enter into such contracts.

Considered strictly from a business point of view right-to-use licensing has several advantages, especially for the software vendor but in many cases for the customer as well. First, right-to-use licenses can be tailored to work in a wide variety of ways; for example, software can be licensed per-user, per-machine, per-CPU (for multiprocessor systems), per-concurrent-user, or for an entire organization or part of an organization (site licensing). Different licensing schemes can also be employed based on the use to which software will be put; for example, using software in support of a particular end use may be done

using a separate license from using that same software in support of a different end use, even though the actual users of the software may be the same in both cases. This allows the vendor to offer preferential pricing schemes for certain customers or end uses as appropriate. (For example, the vendor might allow no-charge unrestricted use by all or some noncommercial customers, or might charge less for software used only internally versus software used to provide services to external users.)

Second, software license fees are independent of amounts charged to the customer for services such as technical support, consulting, and systems integration. For US software companies operating under SEC rules this means that software license fees can normally be recognized as revenue immediately at the time the associated product is shipped; by contrast, service fees can be recognized as revenue only over time as the services are actually delivered to customers. This can be an important consideration, especially for small software startup companies that need to grow revenues quickly in order to satisfy investors and fund growth in operations.

Finally, since they are not directly tied to services provided by people, software license fees can be independently negotiated between vendor and customer based on the perceived value embodied in the software itself. (For example, the perceived value might be roughly proportional to the amount of money the software can save the customer. If that amount is large then the perceived value of the software will be high.) At the same time the incremental cost of selling another software license is relatively low (since software can be easily reproduced). Thus assuming that the customer and the overall market judge the perceived value of a software product to be relatively high then (all other things being equal) a software company's gross margin on software licenses will be higher than its gross margin on services, and this will translate rather directly into increased profits for the software company.

Of course these characteristics of traditional right-to-use licenses have their potential downsides as well. Licensing arrangements can be overly complicated and difficult for both vendor and customer to administer. The immediate recognition of revenue from software license fees can produce undesired peaks and valleys in a software company's revenue stream, especially when the number of customers is relatively small and the average revenue per customer relatively large, so that the difference of even a day or two in concluding a few deals may shift a fair amount of revenue into one quarter from the next, or vice versa. And higher gross margins for software licenses are susceptible to price pressure from competitors willing and able to discount software heavily or even to give software away on its own or by bundling it with other software.

My purpose here is neither to argue for the superiority of the traditional software business model nor to claim that it can and must be completely abandoned. Rather I want to emphasize these important points:

- For a company considering adopting an open-source strategy, open source needs to be evaluated from a business point of view and not just as "a good thing to do." That requires being clear on the advantages and disadvantages of open source relative to the traditional model.
- How software is licensed has important implications for the software business; this is no less true for open-source business models than it is for traditional software business models.
- Any for-profit company must set prices for its products and services, even when using an open-source business model. (Of course that price could be zero for certain products and/or services.) All other things being equal a software company will be more financially successful setting prices based primarily on perceived value rather than based primarily on cost, in cases where it can legitimately do so (and assuming of course that the perceived value exceeds the cost). Thus it's worth looking at various open-source business models to see where goods and/or services might justifiably be priced using such "value-driven" pricing as opposed to "cost-driven" pricing.

Why do I use the terms "legitimately" and "justifiably" in connection with value-driven pricing? If the customer feels that a product has value to warrant the price and if the seller is not coercing the buyer in any way, then surely any price can be justified, no matter how high, and no pricing scheme is morally "better" than another. Yes, but in the case of open-source software there are other parties involved in the transaction, namely those developers who contribute bug fixes, minor enhancements, and (in some cases) major additions in functionality, usually without any direct monetary compensation. Although such developers freely choose whether to do this or not, both pragmatic considerations and simple fairness dictate that you take their opinions and interests into consideration when determining your desired business model and how to price your products and services; I discuss this in more detail below.

"Why would outside developers work on open-source products 'for free'?"

I've discussed above how by distributing source code your company could be able to take advantage of the talents of "outside" software developers. But why would those developers want to work with your source code, especially those developers who are doing it noncommercially and thus in a sense are working "for free?"

First, developers are in large part attracted to working on software that holds the promise of helping them solve problems they themselves consider significant. If you create software that addresses important problems, even if the problems are important only to a subset of the overall developer community, then it is a reasonably safe bet that at least some other developers will find this software promising themselves and worthy of working with, especially if they have access to the underlying source code. (As Eric Raymond notes in the "The

Cathedral and the Bazaar," your software need not completely solve a problem, it need only offer a "plausible promise" that it could do so as it is evolved.)

Once attracted to working with your source code, what would keep developers interested in doing so? The answers will vary depending on whether the developers are doing commercial or noncommercial development. Commercial developers will obviously be interested in some way to profit financially from their work. Working with your source code they will likely have a variety of possible approaches open to them:

- They could specialize in customizing your software for particular customers or vertical markets.
- They could provide fee-based product and/or developer support for customers.
- They could potentially create and sell add-on products, either independently or as one of your channel partners. (Selling products themselves as opposed to support for such products assumes that the license you use permits proprietary add-ons; see below.)
- They could license derivative works and related technology back to you (where the open-source license permits this).
- They could be hired by you, either as contractors or as employees.
- They could found companies that end up being acquired by your company or others.

About the only thing commercial developers could *not* do with your source code is found new software businesses "from scratch" totally unencumbered by source code licensing restrictions.

For noncommercial developers the rewards would be different. They would likely be motivated more by the ego reward of seeing their software used by other people and praised by other developers. They may also be motivated by the ideals of code sharing and general openness found in the open-source developer community. If starting with your source allows them to build more interesting software and if they can distribute that software freely (thus spreading their reputation and/or upholding open-source ideals), then it is likely that some of them would be enthusiastic about working with your code.

Assuming that developers have sufficient positive incentive to work with your source code, you then also have to be concerned with removing obstacles to their participation. This goes back to my comment above about treating open-source developers fairly; licensing in particular is a key area where considerations of fairness are important.

Basic principles of open-source licensing

Historically open-source software has been "free software" in the sense that no right-to-use license fee or right-to-redistribute license fee is charged users for

the software, whether in source or binary form. Note that this does *not* mean that open-source products do not have software licenses associated with them; in fact they almost always do. The Open Source Definition (written by Bruce Perens and originally published as the Debian Free Software Guidelines) is an abstraction from various free software licenses in common use; it explicitly states that open-source software licenses "may not restrict any party from selling or giving away the software" and "may not require a royalty or other fee for such sale."

This has the effect of setting the effective license price at zero, since once a (binary or source) copy of the software is given to any user then that user may turn around and redistribute the software to others without charging them a license fee and without owing any license fee to the originator of the software. The Open Source Definition also contains a number of other provisions intended to prevent vendors from weakening or evading this basic guarantee, including the provision that all users and all types of end use be treated equally (so that, for example, businesses cannot be charged a license fee while noncommercial use is exempted).

Why should this be? Part of the reason lies in the origins of free software as a concept; charging for software licenses was (and is) seen by many as antithetical to the interests of users (who would be better off if provided free and unrestricted use of software) and to society as a whole. For example, in the GNU Manifesto Richard Stallman claimed that "[a]rrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for" and that

> Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program.

In this view open-source licenses are designed the way they are for essentially utilitarian reasons, in order to promote the use, distribution, and creation of useful software.

However the provisions of the Open Source Definition and of open-source licenses can also be seen not just as guarantees to users in general but also as guarantees to software developers in particular, that they will be treated fairly and given equal opportunity to succeed or fail based on their merits. The underlying problem here is that the original developers of the software are always privileged in a fundamental sense: under existing legal environments they "own" the original code (in the sense of holding copyright to it), they have the power to set license terms for its use and redistribution (as a consequence of copyright), and given that they know most about the software they will likely have the leading role in its continued development (at least initially). This is

obviously true in the case of a commercial company converting a proprietary product to open source; it is also true in the case of a noncommercial developer who creates the initial version of an open-source program. (In his paper Homesteading the Noosphere Eric Raymond argues that for all open-source software there will always be one person or organization that has "the exclusive right...to *re-distribute modified versions*" [emphasis in the original], and thus has a privileged position with respect to it; Raymond sees this right as being a kind of common-law property right.)

Consider the rest of the (actual or potential) developer community who are not thus privileged and who are expected to contribute their labor to the improvement of the open-source product, usually for no direct compensation. Why should they participate in what threatens to be an unfair exchange, unless the gap between privileged and non-privileged developers is minimized? One way to minimize the gap is to eliminate license fees; this prevents the privileged developer(s) from making a direct monetary profit from the unpaid labor of other developers. A second way to minimize the gap is to treat all developers equally in the context of the open-source licensing terms, with no developer having special privileges or advantages *as a direct consequence of the license*. This helps ensure that in theory all developers have equal opportunity to achieve success, subject only to the skill and resources they can bring to the task, and that neither you nor anyone else can leverage a privileged position to obtain unjust advantage over other developers.

Open source licenses

Over the years several licenses have been written for use with open-source software, so that companies can select an appropriate license to match their particular situations. If you wish you can also write a new license (as Netscape did), either by creating it from scratch or by modifying an existing license; however note that this is not as simple to do as it might appear, and if you are contemplating doing this I strongly recommend that you research the various open-source licenses in use today and acquaint yourself as much as possible with the long history of public discussions as to their respective merits.

Public domain software

The first possible choice for an open source license is simply using no license at all; this is the case for software that is released into the public domain. Although the term "public-domain software" is often used loosely to refer to open-source or free software in general, public-domain software is strictly speaking software which has no copyright (e.g., the copyright has expired, or the copyright holder has explicitly waived copyright); since there is no copyright there is no "owner" of the software to grant licenses, and hence anyone may use the software in any way without any restrictions.

Putting software into the public domain grants the maximum freedom possible

to end users and developers. However at the same time it opens the possibility that one or more developers may take the software and use it as a base to create proprietary programs; if those programs become dominant in the market then from a practical point of view the software is no longer open-source, even if one form of it remains available in the public domain. (In fact, users may not even be aware that the proprietary products use public-domain code.)

Because of this potential problem most open-source advocates recommend not making software public-domain; even developers who do not believe in the concept of "intellectual property" still advocate using the mechanism of copyright, if only to be able to use a formal open-source license to promote certain beliefs and practices. (See the GNU General Public License for the best example of this.)

BSD License and BSD-style licenses

The BSD License was originally used for the Unix distributions released by the University of California at Berkeley. ("BSD" stands for "Berkeley Software Distribution.") Since then the BSD License or licenses adapted from or similar to it (including the original MIT and X Consortium License) have been used for several other open-source projects, including FreeBSD, NetBSD, and OpenBSD (Unix-based operating systems), Apache (a web server), SLAPD (an LDAP-based directory service), XFree86 (an implementation of the X Window System), and SATAN (a network security analysis tool), to mention only a few.

The BSD License has the following main features:

- an explicit grant of the right to unlimited use in source or binary form
- a requirement that the developers' copyright notices (and related material) be retained
- a requirement that the developers be credited in "advertising material"
- legal boilerplate to limit the developers' liability

Some licenses derived from the BSD License (like the original X Consortium License) omit the advertising requirement (which some have criticized); others add a provision requiring that the software be provided "at cost."

As noted above, the original BSD License was developed in order to release non-commercial software developed as a byproduct of university research, and its legal provisions reflect this heritage: they affirm the academic tradition of giving proper credit to researchers (i.e., the developers) and safeguard the basic legal interests of the university (i.e., the organization employing the developers), but otherwise impose no real restrictions on use of the software. From the point of view of a commercial software company BSD-style licenses contain the minimum terms and conditions that an open-source license would need to have in order to be an effective license at all; from the point of view of open-source developers BSD-style licenses allow the maximum freedom in using

the source to create derivative works. This includes the freedom to take open-source software under a BSD-style license and use it to create a proprietary product for which source code is not made available. As a result many open-source advocates recommend not using BSD-style licenses, preferring licenses that require (to a greater or lesser degree) that derivative works of open-source software also be made available as open source.

GNU General Public License

The GNU General Public License (GNU GPL or plain GPL) was created by Richard Stallman and the Free Software Foundation. The GPL in many ways occupies a place at the opposite end of the spectrum from BSD-style licenses: Where BSD-style licenses permit essentially unlimited commercial use of open-source software and essentially unrestricted creation of proprietary derivative works, the GPL is explicitly designed to prevent open-source software from being used to create proprietary derivative works. It does this through "copyleft" provisions in the GPL that require

- that programs licensed under the GPL must be distributed without a license fee and with source code made available; and
- that derivative works of a program licensed under the GPL must also be licensed under the GPL.

The GPL has a quite broad definition of what constitutes a derivative work of a GPL-ed program: "any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof". Besides modified versions of GPL-ed programs, this clearly includes programs that incorporate code fragments from a GPL-ed program or whose executables include statically linked GPL-ed libraries. It does not matter what license the source code of the new program was originally licensed under; the GPL explicitly states that

> [W]hen you distribute the same sections [i.e., code not under the GPL] as part of a whole which is a work based on the Program [i.e., a work under the GPL], the distribution of the whole must be on the terms of this License [i.e., the GPL], whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. [notes added]

Thus if you use GPL-ed code in a proprietary program containing your own source code, you must make your source code available under the same terms as the original GPL-ed code. (This property of GPL-ed code has led many to compare it to a virus subverting the proprietary "host" program to create more GPL-ed code.)

Two interesting cases are where a proprietary program dynamically links to GPL-ed code or a GPL-ed program dynamically links to proprietary code. The

latter case arises when a GPL-ed program uses existing system facilities such as C run-time libraries or GUI toolkit libraries; the GPL contains a special provision (in paragraph 3) exempting proprietary libraries from source code disclosure provisions if they are "normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system."

As to the former case, Richard Stallman has claimed that if a developer uses GPL-ed code to write code designed to be dynamically linked to from a proprietary program (for example, a GPL-ed plug-in module for a proprietary image processing program) then that developer is violating the GPL; Stallman later clarified this to add that if the developer were writing the GPL-ed code "from scratch" then they could grant special permission to use the code in this way, or if using GPL-ed code from other developers could ask those developers to grant such an exemption. This does not however change the intent of the GPL: Stallman still claims that "a GPL-covered plug-in that is designed to be combined with a non-free master program is a form of combined work, and a violation of the GPL." (Of course in the end this is simply Stallman's personal opinion, albeit an opinion that carries considerable weight among many free software developers; what constitutes a "combined work" or "derived work" in general is ultimately a question for the courts, and if a court ever decides a case involving the GPL they may or may not agree with Stallman on this point.)

The fact that GPL-ed code can "taint" initially non-GPL-ed programs means that in effect you can use GPL-ed source code only to create other GPL-ed programs. This is a desirable feature if (like the Free Software Foundation) you wish to encourage the spread of the GNU philosophy. It is also somewhat desirable (or at least tolerable) if you are creating an open-source product from scratch or if you wish to leverage others' GPL-ed code for your product, because the GPL is widely used and accepted in the free software developer community and there is a large base of existing GPL-ed code. However it creates a problem (and deliberately so) for commercial software vendors who wish to build their own proprietary products using GPL-ed code in some way; it also causes many commercial organizations to shy away from using GPL-ed software.

In particular, if you are considering converting a proprietary software product into an open-source product, and if your product includes third-party technology or shares source code with other products of yours, then the GPL would almost certainly not be a good open-source license for you to use because it would be difficult to impossible to put only part of your code under the GPL and still use that code in products in conjunction with third-party code or your own remaining proprietary code. (You could evade this issue initially by releasing your code under both the GPL and another less-restrictive open-source license and then using the less restrictive license when combining your open-source code with other code. However once others began to add their own GPL-ed code to yours then you could no longer use that new code under the less-restrictive license, unless you obtained permission to do so from each

and every contributor; that could pose insurmountable problems if the contributed code were in turn based on previously GPL-ed code from others, and so on.)

The GNU Library General Public License (GNU LGPL or plain LGPL) attempts to better address the case of open-source products that are libraries intended to be used with other programs, as opposed to programs in their own right; a program that calls routines in a LGPL-ed library is not considered a derived work for licensing purposes in exactly the same sense that it would be if the library were licensed under the GPL. However other derived works of the LPGL-ed library fall under LGPL terms, exactly as they would if the GPL were used instead. The LGPL may or may not be a good license for you to use for an open-source product, again depending on whether the product is being converted from an existing proprietary product and how third-party and still-proprietary code is used in that product.

Artistic License

The Artistic License was originally created by Larry Wall for use with Perl (a scripting language interpreter); it has also been used for open-source Ada library software (as the Ada Community License). The Artistic License is perhaps best thought of an attempt to create an open-source license that eliminates or mitigates the more controversial aspects of the GPL. In particular the Artistic License differs from the GPL in the following ways (among others):

- The Artistic License encourages users to make modifications freely and publicly available, but allows exemptions from such a requirement in the cases where the derived works are used only within an organization and are not publicly distributed, or where the derived works are not represented as being the original work, i.e., the derived work's executables have different names and differences from the original work are documented. The GPL has no such exemptions.
- The Artistic License allows the original work or derived works to be embedded "invisibly" in a proprietary program, i.e., where the proprietary program does not expose direct interfaces to the functionality of the original or derived work. Under the GPL the proprietary program would be considered a derived work also subject to the GPL.
- The Artistic License explicitly declares that input and output data submitted to or produced by the original work (or derived works) does not fall under the terms of the license. This is presumably intended to address a feature of the GPL whereby output from such GPL-ed works as GNU flex (a program to generate lexical analyzer code) was considered to fall under the GPL, since such output contained program fragments embedded as data in the original GPL-ed source code and was thus considered a derived work under the GPL.

Unfortunately the Artistic License is not as general-purpose as it might be; in

particular, some of its provisions (e.g., Sections 5 and 7) seem to assume that the product being licensed is a language interpreter (like Perl) or similar program. However the Artistic License (or a variant thereof) is a possible candidate license for anyone looking for an open-source license more encouraging of code sharing than BSD-style licenses but less restrictive than the GPL or GPL-like licenses.

Mozilla Public License

The Mozilla Public License (MozPL or MPL) and the related Netscape Public License (NPL) were created by Netscape as part of the project to release Netscape Communicator source code. Where the BSD License was created by a university and the GPL and Artistic License were created by free software developers (albeit with somewhat different philosophies), the MozPL is notable as an open-source license created by a commercial software company. As one of the newest open-source licenses the MozPL was influenced by and to some extent incorporates features from a number of older licenses, including the GPL and LGPL; however the MozPL is a distinctive license in its own right and has a number of interesting and even innovative features not found in other open-source licenses.

First, the MozPL contains a general and relatively rigorous definition of when and how derived works fall under the MozPL license provisions (or are "Covered Code," using the terminology of the license). For MozPL-ed source code considered as a set of source files, modifications of the original source files are considered to also fall under the MozPL, as are new source files incorporating extracts from the original source files. Such modified or new files are required to be licensed under the same terms as the original files, and in particular must be made freely and publicly available in source form. In this way the MozPL is similar to the GPL in mandating sharing of code modifications and seeking to prevent open-source code from being converted to proprietary code.

However unlike the GPL the MozPL explicitly permits MozPL-ed code to be combined with separate proprietary code to create a proprietary program (a "Larger Work" in MozPL terminology) which does *not* fall under MozPL terms; such a program may be licensed for a fee and its (proprietary) source code need not be made publicly available. As implied above, the separation between the open-source code and the proprietary code is made at the source file boundaries; a given source file is either under the MozPL or under a different license, which may be a proprietary or an open-source license. (The other license must be compatible with the MozPL in the sense that its provisions may not conflict with those of the MozPL; among other things this rules out combining MozPL-ed and GPL-ed code, since the presence of the GPL-ed code would result in the application of GPL terms to the combined code. However the LGPL allows enough flexibility so that LGPL-ed code may be combined with MozPL-ed code.)

Thus an open-source product initially released under the MozPL may be extended with proprietary code to create new proprietary products, as long as the proprietary code is separate (i.e., in separate files) and interacts with the open-source code using a defined API. (This feature of the MozPL is to some extent reminiscent of the LGPL, although the corresponding LGPL terms are more restrictive.) If the enabling code for the API is not already in the open-source product then changes to the open-source product to create the API fall under the MozPL (because they would be modifications to the original MozPL-ed source files) and must be made freely and publicly available by the developer creating the API (which in this case would be the developer creating the proprietary product). This encourages open competition among both commercial and non-commercial developers by allowing others to create alternatives (whether open-source or proprietary) to any proprietary products based on MozPL-ed code.

The NPL is a variant of the MozPL designed specifically for use with the released Netscape Communicator source code. (In practice the NPL was drafted first and the MozPL then generalized from it.) The NPL exists because prior to being released as open source Netscape Communicator already existed as a proprietary commercial product that shared source code with other proprietary products (including Netscape's SuiteSpot servers) and for which source code had been licensed to third parties under existing contracts still in force; the MozPL by itself would not be sufficient to address the consequent legal complications. The NPL duplicates all MozPL provisions and in addition reserves to Netscape two rights not granted to other developers: the right to use NPL-ed code in other products (like the SuiteSpot servers) without having those products fall under the NPL, and the right to relicense NPL-ed source code to third parties on terms other than those in the NPL. The former right is limited in time (to two years), since the shared code in question can be reorganized over time to separate NPL-ed code and proprietary code into different files and libraries. The latter right is not limited in time, since the third-party contracts in question may exist in perpetuity.

In my (admittedly biased) opinion the MozPL (or a MozPL variant, such as an NPL-like license, if necessary) deserves serious consideration as the license of choice for any commercial company looking at creating an open-source product from scratch, converting a proprietary product to open source, or making open-source extensions or additions to proprietary products; it was designed specifically for the requirements of a commercial software company doing both proprietary and open-source development and was created by lawyers and others intimately familiar with commercial software practices. (Thus, for example, the MozPL incorporates legal language typically required in standard commercial software contracts relating to limitations of liability, responsibility for intellectual property claims, arbitration of disputes, and so on.) The major case where the MozPL is not appropriate is where you are building a business based on existing open-source software already under the GPL or a GPL-like license.

Even if you decide not to use the MozPL it is still worth consulting Netscape's material about the MozPL and NPL and the public discussions through which Netscape received advice on and criticisms of the draft licenses; these are especially valuable as background information if you decide to create your own open-source license.

How to make money with open source?

Now back to the question at hand: How can a software company convert a product to open source and have the increased value provided to customers lead to increased revenue and profits for the company; in simpler terms, how is it possible to make money with an open-source product?

As discussed above, you will *not* make money through traditional software licensing fees. Thus open-source software is a real-life example of a case considered by many industry visionaries, namely intellectual property that is "free" in the sense that anyone can copy it and use it at no charge. (Alternatively, one could consider open-source software as not constituting intellectual property at all, a view held by some open-source advocates; see also the discussion on licensing above.) For example, Esther Dyson (in *Release 2.0* and elsewhere) has assumed that intellectual property considered as "content" will be free or near-free in the future, and has proposed the use of other business models for making money off intellectual property, based generally on either providing services tied to the intellectual property (e.g., as in consulting) or on using the intellectual property to attract people's attention and then realizing money from that (e.g., as in advertising).

Open-source business models

More specifically, there are a number of business models that have been attempted with open-source source, and a number of others that are possible in theory and may be workable in practice. As discussed above, all of the models assume the absence of traditional software licensing fees. I discuss these models in more detail below, including:

- which vendors (if any) are using this model
- what general types of open-source license might be appropriate for the model
- what opportunities exist for vendors to differentiate themselves
- what opportunities exist to set prices based on perceived value rather than actual cost

(Note: The names and basic definitions for the first four models below are from the OpenSource.Org web site.)

Support Sellers

In the "Support Sellers" model software-related revenue comes from media distribution, branding, training, consulting, custom development, and post-sales support instead of from traditional software licensing fees. This is the original free software business model advocated by Richard Stallman in the GNU Manifesto and first implemented by Cygnus Solutions, a for-profit company formed to provide support services for GNU tools (most notably the GNU compilers). It is the most common model today for companies involved with open source and in the Linux market is used by several vendors, of which Red Hat Software and Caldera Software are perhaps best known. For historical reasons all these companies use the GNU General Public License (GPL), but most if not all open-source licenses would work for this model.

Revenue is generated in this model by selling two broad categories of items: physical goods, e.g., media and hard-copy documentation, and/or services, e.g., technical support. Vendors can differentiate themselves by providing more complete and easier-to-use software distributions, in essence simplifying and improving the user's experience with the open-source software in question; they can also differentiate themselves by the quality and pricing of their service offerings.

In this model there is only limited ability to use value-driven pricing; more typically the pricing is determined primarily by the cost to provide goods and services, as there is price competition with other vendors offering comparable goods and services and definite limits to what users are willing to pay for them. However if a vendor's reputation is good then that can be used to justify higher prices than for a "commodity" offering.

Loss Leader

In the "Loss Leader" model a no-charge open-source product is used as a loss leader for traditional commercial software; the open-source product generates little or no revenue, but providing the product makes it more likely that customers will buy other products that are sold using the traditional software business model. To some extent this is the model now being used by Netscape with the Netscape Communicator product (but see also the discussion of the "Service Enabler" business model below); it will also be used by Sendmail, Inc., which plans to release a commercial product based on the open-source version of the popular sendmail mail system..

If the open-source product shares any source code with the company's proprietary products (for example, they use common libraries) then the open-source license chosen must allow distribution of such source code for the open-source product while allowing the same source code to be used in proprietary products distributed using standard licenses. The company should therefore avoid the use of the GPL or other licenses that aggressively "taint" products incorporating code under the license; a BSD-style license would work fine, and the Mozilla Public License or variants thereof might be appropriate if

the common source code is accessed via a set of defined APIs.

Generation of revenue (if any) from the open source product could be done as in the "Support Sellers" model, i.e., by selling media and services. However typically the bulk of revenue generated would be through sales of other software products; having the open-source product could increase sales of such traditional products in various ways:

- by helping build the overall vendor brand and reputation
- by making the traditional products more functional and useful (in essence adding value to them)
- by increasing the overall base of developers and users familiar with and loyal to the vendor's total product line

Vendors have ample scope to differentiate themselves based on the products in the traditional product line, and can also employ value-driven pricing where appropriate with such products.

Widget Frosting

The "Widget Frosting" business model is intended for companies that are in business primarily to sell hardware ("widgets") but which use the open-source model for enabling software such as driver and interface code ("frosting") distributed at no charge along with the hardware. The hardware could be anything from a individual chipset to a controller board to a peripheral device to a complete computer system. The enabling software could be driver code (e.g., for a graphics board or peripheral), compilers and linkers (e.g., for microprocessors and related chips), or complete applications or operating systems (e.g., for a workstation or network computer). Companies could use any of a number of open-source licenses.

Corel is using this model for the Netwinder product line to be offered by its Corel Computer subsidiary; Netwinder is essentially a network computer using Linux as its operating system kernel. Corel Computer is porting the Linux kernel to Netwinder and will release all Netwinder driver code as open source under the GPL; it will also release a set of Netwinder development tools based on open-source applications. Another example of companies using this model to some degree is VA Linux Systems and other companies which sell PCs designed and preconfigured to run Linux; unlike Corel VA Linux and its competitors are selling standard PC hardware, but are still using Linux to maximize the utility of their systems (e.g., as network servers or high-end workstations).

In the "Widget Frosting" model most if not all revenue would be generated through sales of the hardware itself. Using open source for the enabling software could increase hardware sales as in the "Loss Leader" scenario, e.g., by increasing the base of developers familiar with the hardware and able to make it perform to full capacity, thus making the hardware more functional,

reliable, and useful to its end users.

In this model vendors can differentiate themselves based on the attributes of the underlying hardware, e.g., functionality, performance, reliability, flexibility, and cost; the function of the open-source software is to enable the hardware to fulfill its full potential with regard to these qualities. Since the vendor is selling physical goods for which competitive products exist in most cases, the pricing is typically much more cost-driven than value-driven.

Accessorizing

The "Accessorizing" business model is for companies which distribute books and other physical items (as opposed to software and services) associated with and supportive of open source software. Here the company does not typically participate in open-source development per se, but rather for the most part piggybacks on open-source software developed and maintained by others. A good example of a company pursuing this model is O'Reilly & Associates, which publishes books documenting and explaining various open software products (including Linux, Perl, GNU Emacs, etc.). The only license feature required for this model is the ability to bundle free software with the item being sold (where this makes sense); by definition this is permitted by any true open-source license.

Vendors can differentiate themselves based on the quality of the goods themselves, and can also build some brand loyalty among people who use and like open-source software and who are appreciative of vendors that support it in some way. Since what is being sold here is a physical item as opposed to intellectual property of some sort, pricing will tend to be cost-driven rather than value-driven for the most part, although in some cases brand reputation can justify somewhat higher prices than otherwise possible.

Service Enabler

In the "Service Enabler" business model a company creates and distributes open-source software primarily to support access to revenue-generating on-line services. (The services may generate revenue, e.g., through subscription fees or advertising.) To a certain extent Netscape can be seen as pursuing this model in addition to the "Loss Leader" model, given that Netscape Communicator can be used as a front-end to access Netscape's Netcenter services (from which Netscape derives revenue from advertising and related sources). Another example would be a company providing a fee-based on-line game service which created special-purpose software to access the service and then distributed the software as open source so that users could not only download the software at no charge but could also modify the software, e.g., to customize it for their use or to support special input or output devices.

For this model a company would likely use an open-source license that

minimized the possibility that its open source software could be turned directly into proprietary software by competitors; examples of such licenses include the GPL and the Mozilla Public License.

Vendors can differentiate themselves based on the attributes of the services themselves; some of these attributes are a function of the open-source software serving as the front end, but as much or more would be a function of the back-end systems. Those back-end systems could be based on proprietary software or on a combination of open-source and proprietary software; in either case, by creating unique and useful services which could not be easily duplicated by competitors a company could justify pricing the service more commensurate with the value experienced by the user. (For example, an on-line gaming service could price itself based on the entertainment value it provides to users relative to alternatives like traditional video games, movies, and cable TV.)

The following business models are more theoretical, in the sense that no companies appear to be actively using them today. However they appear to be at least theoretically feasible, and it may be that in the future one or more companies will be able to implement them successfully.

Sell It, Free It

The "Sell It, Free It" model is essentially the "Loss Leader" model repeated and extended through time. In this model a company would deliberately structure its development and licensing practices so as to release software products first as traditional commercial products and then convert them to open-source products when they reach an appropriate point in their life cycle where the benefits of developing them in an open-source environment outweigh the direct software license revenue they produce. The newly freed open-source products would still add value to the remaining proprietary products, as in the "Loss Leader" model; in fact, if the proper open-source license were chosen (e.g., a BSD-style license or the Mozilla Public License) then newer proprietary products could be based in part on code from older products now released as open source.

There would be two main tactical problems to overcome in successfully implementing a "Sell It, Free It" strategy. The first would be deciding exactly the right time in the product's life cycle to convert it to open source; doing it too early might mean unnecessarily forgoing significant license revenues, while doing it too late might lessen the interest of open source developers in contributing to the product's further development. The second problem is related to the first: If customers think the product is likely to be converted to open source at some point then it is more difficult to justify to them why they should buy it now rather than waiting. The problem becomes more tractable if the product's sales cycle is short relative to the product's life cycle (otherwise price reductions or conversion to open source are likely before customers

complete their buying decisions) and if product pricing is managed so as to plan for scheduled reductions in license price and at least partially compensate for such reductions with revenue from other sources, such as technical support or professional services.

In a fully-realized "Sell It, Free It" model customers buying the product near the beginning of its life cycle are in essence paying a premium for the value of having the software earlier rather than later, and license pricing can reflect this value. Once the product converts to open source it becomes in essence a commodity, but it can still add value to newer proprietary products, whose license pricing can reflect that added value.

Brand Licensing

In the "Brand Licensing" model a company makes the software product itself open source but retains the rights to its product trademarks and related intellectual property, and charges other companies for the right to use those trademarks in creating derivative products distributed under the exact same brand name. This of course requires that the product exist in at least two different forms with two different names: the "official" product referred to by a trademarked name, and the "unofficial" product referred to by a separate name.

Using this model was not possible with traditional free software products because the product "brand names" (as it were) were typically not formally registered as trademarks. (In the case of Linux this caused a dispute requiring legal action to resolve.) However a commercial product being converted to open source will almost always have associated with it trademarks in the form of product names and logos, and these can remain under the control of the company. For example, even though Netscape released source code for the Netscape Communicator product, only Netscape (or authorized Netscape licensees) can use the source code to create a product called "Netscape Communicator;" products built by others from the source are typically referred to by the name "Mozilla" (from the original code name for Netscape Navigator 1.0).

In general the two product versions (with and without associated trademarks) could be built from identical or near-identical source code; however from the perspective of the market they would be two different products with possibly different perceived value. (For example, the branded product may have undergone additional testing and validation not done with the non-branded product.) If you convert a product to open source then a third party wishing to distribute a product based on that source (for example, for distribution as part of a special on-line service) may also wish to separately pay you for a license to use your trademarks in association with that product. The price of that license can reflect the brand value and reputation that your company (and by association, your trademarks) have in the marketplace.

(I should note that even though I've used Netscape as an example here, Netscape has not publicly announced any plans to license its trademarks as described above.)

Software Franchising

"Software Franchising" is a possible business model based on taking to their logical conclusion the ideas of some of the preceding models, in particular "Brand Licensing" and "Support Sellers." The "raw materials" underlying a support seller's business are available to anyone, but it's reasonable to assume that some may be better at the business than others and as a result may build value in a particular brand associated with the company's services. If the company then wishes to expand, one possibility would be to grow not through direct hiring and acquisition but rather through franchising; in other words, the company would authorize other developers to use its brand names and trademarks in creating associated organizations doing open-source support and custom software development in particular geographic areas or vertical markets.

In this model the "software franchisor" would not only license brands and trademarks but also supply franchisees with training (e.g., in specific aspects of running an open-source development effort and support seller business) and services (e.g., centralized support for contracting and procurement, advertising and marketing campaigns). Revenues would come from sources such as sales of franchises and royalties based on franchisees' revenues.

Hybrid business models

The business models discussed in the previous section are all based on open-source software in one way or another, with that software licensed under a true open-source license along the lines of that recommended by the Open Source Definition; if a business also involves software that is not open-source (as for example in the "Loss Leader" model), then the software is assumed to be totally proprietary or "closed-source," i.e., the company does not make its source available except under very restrictive licenses.

However there are possible business models that involve software distributed under licenses that are not quite open-source in the strict sense, but are also not as restrictive as traditional proprietary licenses. One way to explore the space of possible "hybrid" business models is to go back to the Open Source Definition and look at relaxing one or more of its requirements; here are some changes that might be made:

- change the availability of source code
- change the treatment of different users
- change the treatment of different types of use

Let's consider these in more detail, along with two examples of organizations that have pursued such hybrid models: Troll Tech and The Open Group.

Changing availability of source code

An open-source license grants at least four separate rights with regard to source code:

- the right to view (i.e., to see the code in the first place and possess a copy of it)
- the right to use (i.e., to compile the source into executable form and run the resulting application)
- the right to modify (i.e., to make changes to the source code)
- the right to redistribute (i.e., to give the source code to a third party, potentially in either modified or unmodified form)

All these rights come bundled together and are available "for free;" i.e., the developer does not charge the user any license fees to have them.

However at least in theory these rights could be separated and could be made conditional on paying some sort of license fee. (Some might object that there are no technical means to enforce such separation; for example, if someone has a copy of the code and can use it to build an executable, there is no way to prevent them from modifying it. This is true, but in practice such considerations have not been a major problem in commercial software licensing; for example, although many software products are now available over the Internet for downloading (e.g., for evaluation use) and thus there is no technical way to enforce payment for a right-to-use license, nevertheless almost all commercial and government organizations do in fact "pay up" and officially acquire licenses if their users end up using the product -- particularly if the organizations are prompted to do so.)

Thus, for example, users could be given a low-cost or even no-cost license to possess a copy of product source code, compile it for internal use, and make bug fixes as necessary, but would be charged a higher license fee if they wished to redistribute modified versions. (Returning bug fixes to the original vendor could be exempted from this restriction.) Such licensing terms would likely not be attractive to many if not most noncommercial developers, but might be acceptable and of interest to many commercial organizations. If the product serves a very specialized market then it is quite possible that only commercial users would be interested in the source code anyway, so that a lack of participation by noncommercial developers would not necessarily always be a drawback.

As a real-life example, Troll Tech distributes a free version of its Qt GUI toolkit , including source code; the Qt Free Edition license originally allowed developers to view, use, and modify the source code, but prohibited them from distributing

modifications. (As discussed below, Troll Tech later changed licensing of the Qt Free Edition to loosen this restriction somewhat.)

Changing treatment of different users

An open-source license is available to all potential users no matter who they are. (The Open Source Definition specifies that an open source license may not discriminate between users based on "fields on endeavor," which in essence is saying the same thing.) However in general a source license could be offered on different terms to one group of users versus another; the most common and probably the most useful distinction is made between commercial and noncommercial users. (Note that this distinction is not always easily made in general; for example, there are many instances of large organizations which are technically nonprofits but which are in many ways indistinguishable from for-profit entities performing similar functions. In practice it would be up to the software vendor to decide to which class a particular user would be assigned.)

Thus, for example, noncommercial users could be given a full set of rights to the source (rights to view, use, modify, and redistribute), but the vendor might charge commercial users license fees to obtain all or some of those rights. A real-life example of this was the licensing originally adopted for the X11R6.4 release of the X Window System from The Open Group; X11R6.4 source code was made available under a noncommercial license at no charge; however users wishing to distribute binaries for commercial purposes were required to pay a fee for a separate commercial license. (As discussed below, the Open Group later changed the licensing of X11R6.4, and dropped the distinction between commercial and noncommercial use.) Troll Tech also originally prohibited commercial use of the Qt Free Edition, and sold a separate Qt Professional Edition for that purpose. (As discussed below, Troll Tech subsequently loosened this restriction somewhat.)

Changing treatment of different types of use

An open-source license does not discriminate between different uses of the software; a given user may use the software in any context and for any purpose they wish. However in general a source license could have restrictions limiting the use of the software for certain purposes, or could be offered on different terms (including different prices) depending on the end use.

Thus, for example, a source license could allow personal use or internal use within an organization at low cost or no cost, but prohibit or require higher license fees for use of the software to provide a service to other users (e.g., on the Internet or as part of an extranet). Alternatively, the license could allow no-charge use of the software on particular operating system and/or hardware platforms (e.g., Linux) but require that a license fee be paid to use the software on other platforms (e.g., Windows 95 or Solaris). As a real-life example, Troll

Tech originally restricted the Qt Free Edition to being used only under the X Window System (e.g., on Linux and various Unix-based systems); developers wishing a version of Qt for Windows 95 or other platforms had to license Qt Professional Edition. (The current Qt Free Edition license does not contain this restriction, but as a practical matter there is still only an X version of Qt Free Edition.)

Viability of hybrid models

A company using a hybrid business model can derive revenue and profits directly from the sale of licenses for the software in question as opposed to deriving revenue and profits only through more indirect means, as in an open-source business model. The question is whether such a company can also gain the other benefits of open-source business models such as leveraging outside developers to help improve its software and drive its adoption in the market. Although commercial developers presumably would have no qualms about working with hybrid software, many noncommercial developers would likely object to the more restrictive terms of hybrid licenses versus open-source licenses.

For example, The Open Group aroused widespread controversy when they changed the licensing terms for X11R6.4 to use separate commercial and noncommercial licenses as opposed to the single BSD-style license previously used; among other things this resulted in the threat of a split in X11 development, with the XFree86 Project deciding to not use X11R6.4 or other future releases from The Open Group, instead committing to continuing development based on X11R6.3. Subsequently The Open Group decided to change the licensing of X11R6.4; the current X11R6.4 license is a true open-source license and explicitly allows commercial sale of X11R6.4 without paying royalties. In return the XFree86 project decided to employ X11R6.4 as the basis for their 4.0 release, and later formally joined X.Org, the group established by The Open Group to oversee X development.

Similarly, although Troll Tech attempted to justify its business model to the open-source developer community and established a foundation to help ensure that Qt Free Edition would never become a proprietary product, nevertheless many in that community still saw Troll Tech's licensing policies as inconsistent with the goals of the free software movement. This resulted in a rivalry of sorts between proponents of Qt and the Qt-based K Desktop Environment (KDE) and proponents of the GTK+ toolkit and GNOME desktop environment, which are distributed under the LGPL and GPL. It also resulted in the formation of a separate project (known as Harmony) to build a Qt-compatible library to be distributed under the LGPL. Subsequently Troll Tech changed its licensing of the Qt Free Edition to use a new Q Public License (QPL). The QPL is a true open-source license, and eliminates some of the restrictions of the original Qt Free Edition license; for example, the QPL now allows distribution of modified versions, as well as commercial use of the library. However Troll Tech does

require that modifications to Qt Free Edition source code be distributed separately from the base Troll Tech distribution, and also still attempts to make some distinctions between use of Qt in noncommercial and commercial contexts in an effort to support Troll Tech's chosen business model. For example, the QPL requires that applications developed to link to the Qt Free Edition should themselves be open-source applications; development of traditional proprietary software still requires licensing of the Qt Professional Edition.

Issues and tactics

Beyond selecting an appropriate license and business model, what else might your company have to do in order to implement an open-source strategy? This depends to a large extent on whether you will be converting an existing proprietary product to open source or writing an open-source product from scratch; the former is more likely, since even if your open-source product is not simply a new version of an existing product it is quite likely that you'll want to at least reuse code that already exists in other of your products.

When converting an existing product or reusing existing code it's likely you'll have to address the following issues:

- sharing existing code among open-source products and proprietary products
- what to do about code in your products licensed from third parties
- for US companies, how to comply with relevant export control regulations on cryptography
- what other sorts of "sanitization" you will have to do on your source code

I discuss possible tactics to resolve such issues. In addition, I briefly discuss the open-source development process and how you might organize software development for your open-source product or products.

Source code sharing among products

If you have only one software product and that product is or will be open source, or if you are writing an open-source product totally from scratch, then you can skip this section. However if you have multiple software products and especially if those products currently share source code in any way (e.g., they call common libraries or even just have a few routines or header files in common) then you will have to consider the issue of how your open-source product can "coexist" with your other products and how you might have to change your products and your development process to accomodate this. (This section assumes that you hold copyright to the source code in question; if some of the source code is licensed from third parties please see the next section.)

Unfortunately some open-source licences make it difficult to impossible to share

source code between an open-source product and a proprietary product; the presence of code from the open-source product in the proprietary product can trigger licensing provisions that in theory force disclosure of source code from the proprietary product as well. (As discussed previously in relation to the GPL, this "tainting" effect is a feature, not a bug, of the open-source licenses in question.) On the face of it this would preclude (for example) maintaining common libraries used by both sets of products. Short of enforcing a strict separation of code between the products, there are several ways to approach this issue; in either case you must first identify all the source files which are or will be shared, and single them out for special treatment.

This first approach is to use an open-source license that does not taint in any way; a BSD-style license is the obvious choice. If you release all the common code under such a license then it can included in any type of product without any untoward consequences arising from the licensing provisions. If someone outside your company contributes changes under the same license then you are free to incorporate those chances into your common code and still use that code with your proprietary products. (For that matter, you could incorporate such changes directly into a proprietary product.)

If for other reasons you want or need to use a strongly tainting open-source license like the GPL, a second approach is to put your common code under two licenses: your existing proprietary license and the separate open-source license. (Since you hold copyright to the code, as assumed above, you have the right to license it how you please.) When the source files in question are included in your open-source product then the open-source license is in effect; when included in your proprietary products the proprietary license is in effect. The major drawback of this approach is that if someone outside your organization contributes changes to one of the shared source files under the open-source license then you will either have to avoid using the contributed open-source code in the shared routine (because of the possibility of tainting your proprietary products) or ask special permission of the contributor to use the code under the proprietary license as well. Either alternative imposes a burden on your development and legal staff to track changes, secure permissions, and so on. It also raises the possibility of open-source development efforts around your product splitting into two incompatible "forks:" one based on the code as released by you (with only your changes or changes for which you have permission to do dual licensing) and one based on the code with all changes contributed under the open-source license.

If the shared code is in the form of a common library or libraries accessed through a set of defined APIs, then a third approach is to release the common code under an open-source license like the LGPL or MozPL which does not taint across library or source file boundaries. In this case the common library itself is a pure open-source product in and of itself (no dual licensing is needed, and the library can be released as a separate stand-alone open-source product); your proprietary products can call that library under the specific exemptions granted

by the license. (The MozPL is somewhat more liberal than the LGPL in this respect, as previously discussed.) Note that the rest of the open-source product (the part that calls the common library) can be under any open-source license compatible with the LGPL or MozPL; this includes all the licenses previously discussed: GPL, MozPL, BSD-style licenses, and the Artistic License.

However it may be that you do not wish such a common library to be open source but rather want to keep it proprietary. (For example, the library may include code you do not have rights to release or do not wish to release, e.g., to protect trade secrets.) This poses no problem for your proprietary products calling that library, and you could use the same (proprietary) license for both sets of code. However if you wish to call the proprietary library from the open-source product, then you'll need to license the open-source code under a license that permits this usage; the MozPL or BSD-style licenses would work here, the GPL would not.

You would also have to face the issue of how open-source developers would be able to create a complete product on their own as part of the open-source development process: If the proprietary library is critical to the functioning of your open-source product, then you would have to supply binary versions of the library for them to link to. Such a library would have to be provided at no charge, in order that the final open-source product (including the library) be licensable at no charge in source or binary form (as specified by the Open Source Definition). You would also have to provide the binary version of the library on all platforms of interest to open-source developers, because they're not going to be able to port it themselves in the absence of library source code. (On the other hand, given that the API to the library is known, they can in the long run simply create a true open-source equivalent to the proprietary library.)

A final note: Many developers and corporate legal staff are not familar with the nuances of open-source licenses and may be very concerned just about the presence of open-source code "within the firewall," and in particular on the same systems or in the same source code repositories as proprietary code. These concerns while exaggerated are nonetheless understandable, given the theoretical possibility of tainting; however these concerns can be easily addressed by familiarizing people with how open-source licenses work and enforcing some minimal policies in dealing with open source. In particular, no open-source license in use today (including the GPL) causes tainting when open-source code resides on the same storage medium with proprietary code (or is "aggregated with" such code, as the GPL puts it). Thus it is perfectly safe, for example, to maintain code for a GPL-ed or other open-source product in the same source repository used to maintain proprietary code, if the open-source code is in a separate part of the repository and is not linked with or otherwise combined with proprietary code.

Third-party technology

Many of your products probably incorporate technology from other companies, licensed in either source or binary form; how can you handle such technology in the context of an overall strategy of distributing open source code for a product? This depends on the particular licensing terms that have been or could be negotiated with the third-party technology supplier, and in particular whether the third party is already pursuing or is supportive of an open-source strategy.

The first and hardest case is where your license with the technology supplier constrains you to shipping only the binary form of their code, and only embedded in the binary form of your product; the license may also prohibit you from disclosing the APIs by which your product interfaces to the third-party product, with such APIs being considered proprietary to the third-party. If the third-party technology is critical enough to your product and if the license terms and conditions are too restrictive then it is quite possible that you may not be able to release your own source code at all, either for the entire product or for a major component of it. In this case it is probably worth asking the third party to see if their restrictions could be relaxed in some way; it is possible that some middle way exists that would allow you to distribute at least partial source code while still complying with relevant license requirements.

The second case is where the supplied technology is already or could be encapsulated in a wholly separate binary component intended to be accessed by a public API, and the license reflects this usage. One example would be a shared library for a graphical user interface toolkit such as Motif; another more specific example is an add-on module like a browser plug-in. Here you have the option of releasing your own source code (since you are using public APIs), and the end user or developer can separately acquire and license the separate component from the third party. (For example, the required library or libraries might already be included as part of a user's operating system.)

The third case is where the third-party component(s) cannot be made available at all (for whatever reason), but the APIs are still public. Here you have the option of releasing your own code but leaving it up to the customer to find a substitute for any omitted components. Assuming a clever and active open-source developer community working with your source code, this could quite possibly lead to the creation of substitute technology for the original supplier's technology, a substitute that would almost certainly be available under more favorable licensing terms. If you wish you could then potentially adopt the substitute technology instead, or simply accept the existence of two possible versions: a public version using the open-source technology and your own version using the proprietary technology. (This assumes that the open-source license you're using permits you to link your code with proprietary code; for example, the GNU General Public License prohibits this.)

The final case is where the third-party supplier is willing to release their own source code. In the longer term, if the open-source movement proves

successful, there may be technology suppliers willing to use it, especially if they come to believe that this is ultimately in their interest by providing them with new ways to make money as well. This should be a major focus of the open-source movement as a whole, because (among other things) outside developers adding features to open-source products could in turn become technology suppliers to the vendors of those products, and to encourage this those vendors should work to ensure that their suppliers would share in some way in the overall rewards of the business.

Other source code sanitization

Most companies developing proprietary closed-source products treat the contents of their source code as a purely internal matter and have assumed that it will forever remain so (with the possible exception of tightly-controlled release to a partner under NDA). If you have traditionally operated this way and are now considering a move to open source (even if only partially) then you will quite likely be concerned with releasing source code for fear of inadvertently exposing material which might embarrass you in some way or be otherwise damaging to your company should someone happen to come across it. Examples include bugs both minor and major, security flaws, and embarrassing comments by developers; the latter could potentially include comments disparaging the code itself (e.g., "what a brain-dead hack"), the quality of your products ("fix #1,000,001 for this turkey"), or particular of your customers ("patch for the idiots at Acme Widgets"), as well as the standard "seven dirty words" and other language not suitable for material intended to be in universal public distribution.

Bugs and security flaws you need worry less about; if you don't know about them already (and can fix them prior to open-source release) then they will sooner or later be found (and possibly fixed) by developers outside your company. This is by no means a bad thing; rather it's one of the claimed benefits of open source and such investigation and bug fixing should only be encouraged. As for embarrassing comments and the like, these can and should be removed during the work to prepare the source for initial release; you will need to do such "sanitization" work for every source file to be released, with someone assigned to review and if necessary modify each file. (This need not be a single person; rather for each file it should be the person or persons most responsible for the source code in that file.) This sanitization review serves other purposes as well, since (as discussed elsewhere) you will also need to identify source code licensed from third parties, source code affected by export control or other regulations, and source code not intended for release for other reasons.

However you still need to address the sanitization issue for new code developed later. The most straightforward approach to address the issue of public releasability in the long-term is probably to have developers understand that they are "writing for publication" and that each and every line of their code will

likely be viewed both by customers and (even more important) by their peers in the developer community. Within the context of an open-source code strategy you can no longer treat source code as an internal-only matter, but will have to exercise just as much care as you (ideally should) exercise with released binaries and documentation. This might require extra editing steps and code reviews, greater adherence to consistent formatting conventions (at least within particular sets of modules), inclusion of proper licensing notices, and related actions; the immediate responsibility for ensuring this happens rests with the module owners (whether they are employed by you or not).

US export control issues

(This section is addressed specifically to organizations and individuals in the US; if you are based in another country please consult your local laws and regulations. This discussion is for informational purposes only and is not intended as legal advice. If you wish to develop and distribute cryptographic software then you should consult an attorney with expertise in the particular laws and regulations that apply in your jurisdiction.)

So far I've assumed that you would have compete discretion over whether you wished to release your own source code or not, as well as over how much of that source you would choose to release. However there is one key area, namely security-related and cryptographic code, where you are and will almost certainly continue to be constrained in major ways regarding distribution of binary and source code; given activity in the US Congress and Administration over the past year or two, it is even possible that those constraints will become even more restrictive in the future than they are today. It's therefore useful to review how much freedom of action you are likely to have in this area, and how that might affect your open-source distribution strategy.

Under the current regulatory regime, US companies and individuals are prohibited from exporting from the US (except to Canada) software that implements strong encryption (e.g., using greater than a 40-bit symmetric encryption key). Thus at a minimum you would be prohibited from exporting source code that implemented such cryptography. (However you could distribute such source code within the US and Canada, at least for the present, as long as you take appropriate precautions and implement appropriate controls in your software distribution mechanisms to comply with the relevant regulations.)

However the restrictions are actually more onerous than that. The US government has traditionally refused to permit the export of products in which the supplied weak cryptography could be easily replaced with strong cryptographic implementations developed outside the US. For example, the source code for the S/MIME Freeware Library (SFL) is considered to be an export-controlled item with controlled distribution, even though the source does not actually include any cryptographic code. It's therefore reasonable to

conclude that even under the current regulatory regime you will not be able to export source code to security libraries implementing functions such as S/MIME, SSL, PKCS#7, IPSec, etc., and will also be required to remove from the released source code calls to routines in such libraries.

Finally, it's possible that the US government may decide to impose further restrictions in the future as part of attempts to regulate domestic use of cryptography within the US. For example, it may be that US software vendors will be prohibited in the future from shipping any product with encryption functionality unless it implements a key escrow scheme of some type. Assuming that the escrow scheme is enforced by your security library or libraries, you may be prohibited from distributing such security-related source code even within the US and Canada, on the grounds that others (within the US or not) could easily create a derivative work that disabled key escrow.

The ultimate question is where the U.S. government will "draw the line" in terms of attempting to exercise control over the use of cryptography by imposing restrictions on commercial software vendors and noncommercial developers. Given that US software vendors and developers have only limited influence over the future of cryptography regulations, if you decide to pursue a open-source strategy then the best that you can do is simply to distribute your security-related source code to the maximum extent permitted by relevant laws and regulations both in the US and worldwide. For a real-life example of the complications inherent in doing so, see the Mozilla Crypto FAQ.

Open-source development processes

Suppose that you release one of your products as open source (whether an existing product or new). How could you possibly coordinate the efforts of potentially hundreds or even thousands of developers worldwide who might be creating bug fixes, customized versions, and add-on products all based on your source code?

At first glance this would seem to be impossible. However there is at least one example of a successful project involving distributed and relatively loosely-coordinated development of software products arguably at least as complex and functional as any commercial products, namely the project to create the Linux operating system kernel and the body of free software that runs on top of it. The Linux project offers many lessons on how you might potentially organize software development in the context of an open-source strategy. These are discussed at greater length in the paper "The Cathedral and the Bazaar" by Eric Raymond, but some key points can be summarized here as follows, together with implications for you should you wish to pursue such a strategy:

- Properly organized and coordinated, distributed development can produce more products faster and with higher quality than would be possible in an isolated effort. As Raymond puts it, "The developer who uses only his or

her own brain in a closed project is going to fall behind the developer who knows how to create an open, evolutionary context in which bug-spotting and improvements get done by hundreds of people." Substitute "software company" for "developer" and you have a basic theme of this paper; my suggestion is that this approach offers you one way to compete successfully with larger competitors without having their internal resources.

- Distributed development is jump-started and proceeds most rapidly when there is an existing body of code in which developers can see the promise of solutions to problems which interest them, and which developers can use a base for their own work to solve those problems. One of your major roles would be to provide this existing source code base initially, and to continue to seed it with new contributions in the form of new product features with accompanying source code.
- Higher-quality code can be generated faster when you can enlist other people to do not only bug detection and reporting, but bug fixing as well; bug fixers are in turn more motivated when you do frequent releases that incorporate their fixes and enable them to see the fruits of their efforts. Here you would need good mechanisms to ensure that bug fixes from other developers are in fact incorporated back into your source tree. This is both easier and harder than using the Internet as your beta test site, as many software companies are now doing: On the one hand, developers with access to source would be much more likely to not only find problems but also reproduce, diagnose, and fix them. On the other hand, they would have a much higher expectation of having their contributions taken seriously by you; they would request or even demand access to the actual developers working on the components, and would be even more turned off than regular users if they received little or no response to their communications with you.
- The Internet and the collaborative tools built on top of it (email, newsgroups, etc.) form an indispensable infrastructure for the coordination of distributed developers but you can realize the true promise of distributed development only if you provide the proper social context for it, including having development coordinators with the necessary social and communications skills to "lead without coercion" and focus developers' energies into productive channels. Thus to make an open-source strategy work you would have to put the proper effort into selecting the teams and team leaders chosen to develop products according to this strategy; in particular it would be very helpful if your developers had previous experience with open-source projects.

It is important to note that there is no "free lunch" here: You cannot simply release source code, put a few newsgroups up, and expect distributed development to magically self-organize; you would need to make a sustained effort involving various people's time and attention in order to get a distributed open-source development effort to the point where it would produce results. However I believe that the potential rewards are worth the effort, and that you

could not necessarily achieve comparable results by continuing your current development practices.

What about ...?

[This section addresses various objections and concerns relating to open source, especially for a commercial business. To be written.]

Conclusion

[Summary and conclusion. To be written.]

Appendices

Acknowledgments

This paper recycles a fair amount of material from a paper I originally wrote in the late summer and fall of 1997 for internal distribution at Netscape. The original paper was inspired by discussions on various Netscape internal newsgroups over the years about the possibility of releasing Netscape source code; in particular two independent postings from Jamie Zawinski and Eric Krock provided the immediate impetus for me to start writing down my thoughts on this subject. Special thanks go to the two main reviewers of that original paper: John Menkart of Netscape's government sales group, with whom I discussed business models and licensing issues, and Jamie Zawinski, who provided useful information on software development according to the GNU and Linux models, including a pointer to Eric Raymond's paper "The Cathedral and the Bazaar."

Further material and inspiration for the current paper came from the public discussions on the Mozilla Public License and Netscape Public License, from the material created by Eric Raymond, Bruce Perens, and other for the OpenSource.Org web site, and from articles, editorials, and user discussions on the slashdot.org web site.

Finally, full credit must be given to two groups of people at Netscape: to the executive management team (Jim Barksdale, Marc Andreessen, Eric Hahn, Mike Homer, etc.) that committed Netscape to an open-source strategy, and especially to those in the Netscape client engineering group and elsewhere who worked long hours to implement that strategy and succeeded in making their 31 March 1998 deadline for source code release, creating an innovative pair of licenses in the MozPL and NPL, and putting mozilla.org and Mozilla source development on a firm footing for the future; all the theorizing in the world would have been fruitless without their efforts.

Revision history

0.8
: Corrected URL for "Homesteading the Noosphere." Revised references to Linux. Updated URL and name for VA Linux. Updated for new email address.

0.7
: Corrected URL for "The Cathedral and the Bazaar." Corrected URLs and text concerning The Open Group and Troll Tech.

0.6
: Changed contact information.

0.5
: Really completed draft of the section "Issues and tactics."

0.4
: (Mostly) completed draft of the section "Issues and tactics."

0.3
: Completed draft of sections "Making the business case" and "Building a business model."

0.2
: First public release, of executive summary only.

0.1
: First draft, not yet complete.

Revised 20 June 2000
Frank Hecker - hecker@hecker.org