The Cathedral and the Bazaar

Eric Raymond

I anatomize a successful open-source project, fetchmail, that was run as a deliberate test of some theories about software engineering suggested by the history of Linux. I discuss these theories in terms of two fundamentally different development styles, the "cathedral" model, representing most of the commercial world, versus the "bazaar" model of the Linux world. I show that these models derive from opposing assumptions about the nature of the software-debugging task. I then make a sustained argument from the Linux experience for the proposition that "Given enough eyeballs, all bugs are shallow," suggest productive analogies with other self-correcting systems of selfish agents, and conclude with some exploration of the implications of this insight for the future of software.

The Cathedral and the Bazaar

Linux is subversive. Who would have thought even five years ago that a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet?

Certainly not I. By the time Linux shimmered onto my radar screen in early 1993, I had already been involved in Unix and open-source development for ten years. I was one of the first GNU contributors in the mid-1980s. I had released a good deal of open-source software onto the net, developing or co-developing several programs (nethack, Emacs VC and GUD modes, xlife, and others) that are still in wide use today. I thought I knew how it was done.

Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping, and evolutionary programming for years. But I also believed there was a certain critical complexity above which

---

Eric Raymond is the co founder of the Chester County InterLink (CCIL), which provides free Internet access to the residents of Chester County, Pennsylvania. He is the editor of *The New Hacker's Dictionary* (MIT, 1991, 1993) and the author of a book of essays *The Cathedral and the Bazaar*. He is a member of the Merrill Lynch Technology Advisory Board and has hacked much widely used open source software. He has pursued undergraduate studies in philosophy and mathematics at the University of Pennsylvania but has never had a course in computer stuff. His home page is at: <*http://www.tuxedo.org/~esr/*>.

Knowledge, Technology, & Policy, Fall 1999, Vol. 12, No. 3, pp. 23–49.

a more centralized, *a priori* approach was required. I believed that the most important software (operating systems and really large tools like Emacs) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.

Linus Torvalds's style of development—release early and often, delegate everything you can, be open to the point of promiscuity—came as a surprise. No quiet, reverent cathedral-building here—rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who would take submissions from anyone) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only did not fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders.

By mid-1996 I thought I was beginning to understand. Chance handed me a perfect way to test my theory, in the form of an open-source project that I could consciously try to run in the bazaar style. So I did—and it was a significant success.

In the rest of this article, I will tell the story of that project and use it to moot some aphorisms about effective open-source development. Not all of these are things I first learned in the Linux world but we will see how the Linux world gives them particular point. If I am correct, they will help you understand exactly what it is that makes the Linux community such a fountain of good software—and help you become more productive yourself.

The Mail Must Get Through

Since 1993 I had been running the technical side of a small free-access ISP called Chester County InterLink (CCIL) in West Chester, Pennsylvania (I cofounded CCIL and wrote our unique multi-user bulletin-board software—you can check it out by telnetting *to locke.ccil.org <telnet://locke.ccil.org>*. Today it supports almost 3,000 users on thirty lines.) The job allowed me 24-hour-a-day access to the net through CCIL's 56K line—in fact, it practically demanded it.

Accordingly, I had gotten quite used to instant Internet email. For complicated reasons, it was hard to get SLIP to work between my home machine (*snark.thyrsus.com*) and CCIL. When I finally succeeded, I found having to periodically telnet over to locke to check my mail annoying. What I wanted was for my mail to be delivered on snark so that I would be notified when it arrived and could handle it using all my local tools.

Simple sendmail forwarding would not work, because my personal machine is not always on the net and does not have a static IP address. What I needed was a program that would reach out over my SLIP connection and pull across my mail to be delivered locally. I knew such things existed, and that

most of them used a simple application protocol called POP (Post Office Protocol). And sure enough, there was already a POP3 server included with locke's BSD/OS operating system.

I needed a POP3 client. So I went out on the net and found one. Actually, I found three or four. I used pop-perl for a while, but it was missing what seemed an obvious feature, the ability to hack the addresses on fetched mail so replies would work properly.

The problem was this: suppose someone named "joe" on locke sent me mail. If I fetched the mail to snark and then tried to reply to it, my mailer would cheerfully try to ship it to a nonexistent "joe" on snark. Hand-editing reply addresses to tack on *"@ccil.org"* quickly got to be a serious pain.

This was clearly something the computer ought to be doing for me. But none of the existing POP clients knew how. And this brings us to the first lesson:

1. Every good work of software starts by scratching a developer's personal itch.

Perhaps this should have been obvious (it has long been proverbial that "necessity is the mother of invention") but too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world—which may explain why the average quality of software originated in the Linux community is so high.

So, did I immediately launch into a furious whirl of coding up a brand-new POP3 client to compete with the existing ones? Not on your life. I looked carefully at the POP utilities I had in hand, asking myself "which one is closest to what I want?" Because...

2. Good programmers know what to write. Great ones know what to rewrite (and reuse).

While I do not claim to be a great programmer, I try to imitate one. An important trait of the great ones is constructive laziness. They know that you get an "A" not for effort but for results, and that it is almost always easier to start from a good partial solution than from nothing at all.

Linus Torvalds <*http://www.tuxedo.org/~esr/faqs/linus*>, for example, did not actually try to write Linux from scratch. Instead, he started by reusing code and ideas from Minix, a tiny Unix-like OS for PC clones. Eventually all the Minix code went away or was completely rewritten—but while it was there, it provided scaffolding for the infant that would eventually become Linux. In the same spirit, I went looking for an existing POP utility that was reasonably well coded, to use as a development base.

The source-sharing tradition of the Unix world has always been friendly to code reuse (this is why the GNU project chose Unix as a base OS, in spite of serious reservations about the OS itself). The Linux world has taken this tradition nearly to its technological limit; it has terabytes of open sources generally available. So spending time looking for some else's almost-good-enough is more likely to give you good results in the Linux world than anywhere else.

And it did for me. With those I had found earlier, my second search made up a total of nine candidates—fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail, and upop. The one I first settled on was "fetchpop" by Seung-Hong Oh. I put my header-rewrite feature in it, and made various other improvements that the author accepted into his 1.9 release.

A few weeks later, though, I stumbled across the code for "popclient" by Carl Harris, and found I had a problem. Though fetchpop had some good original ideas in it (such as its daemon mode), it could only handle POP3 and was rather amateurishly coded (Seung-Hong was at that time a bright but inexperienced programmer, and both traits showed). Carl's code was better, quite professional and solid, but his program lacked several important and rather tricky-to-implement fetchpop features (including those I had coded myself).

Stay or switch? If I switched, I would be throwing away the coding I had already done in exchange for a better development base. A practical motive to switch was the presence of multiple-protocol support. POP3 is the most commonly used of the post-office server protocols, but not the only one. Fetchpop and the other competition did not do POP2, RPOP, or APOP, and I was already having vague thoughts of perhaps adding IMAP *<http://www.imap.org>* (Internet Message Access Protocol, the most recently designed and most powerful post-office protocol) just for fun.

But I had a more theoretical reason to think switching might be as good an idea as well, something I learned long before Linux.

3. "Plan to throw one away; you will, anyhow." (Fred Brooks, The Mythical Man-Month, *Chapter 11).*

Or, to put it another way, you often do not really understand the problem until after the first time you implement a solution. The second time, maybe you know enough to do it right. So if you want to get it right, be ready to start over at least once.

Well (I told myself), the changes to fetchpop had been my first try. So I switched.

After I sent my first set of popclient patches to Carl Harris on 25 June 1996, I found out that he had basically lost interest in popclient some time before. The code was a bit dusty, with minor bugs hanging out. I had many changes to make, and we quickly agreed that the logical thing for me to do was take over the program.

Without my actually noticing, the project had escalated. No longer was I just contemplating minor patches to an existing POP client. I took on maintaining an entire one, and there were ideas bubbling in my head that I knew would probably lead to major changes.

In a software culture that encourages code-sharing, this is a natural way for a project to evolve. I was acting out this:

4. If you have the right attitude, interesting problems will find you.

But Carl Harris's attitude was even more important. He understood that...

5. When you lose interest in a program, your last duty to it is to hand it off to a competent successor.

Without ever having to discuss it, Carl and I knew we had a common goal of having the best solution out there. The only question for either of us was whether I could establish that I was a safe pair of hands. Once I did that, he acted with grace and dispatch. I hope I will do as well when it comes my turn.

The Importance of Having Users

And so I inherited popclient. Just as importantly, I inherited popclient's user base. Users are wonderful things to have, not just because they demonstrate that you are serving a need, but to prove that you have done something right. Properly cultivated, they can become co-developers.

Another strength of the Unix tradition, one that Linux pushes to a happy extreme, is that a lot of users are hackers too. Because source code is available, they can be effective hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your users will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.

6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.

The power of this effect is easy to underestimate. In fact, pretty well all of us in the open-source world drastically underestimated how well it would scale up with number of users and against system complexity, until Linus Torvalds showed us differently.

In fact, I think Linus's cleverest and most consequential hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model. When I expressed this opinion in his presence once, he smiled and quietly repeated something he has often said: "I'm basically a very lazy person who likes to get credit for things other people actually do." Lazy like a fox. Or, as Robert Heinlein famously wrote of one of his characters, too lazy to fail.

In retrospect, one precedent for the methods and success of Linux can be seen in the development of the GNU Emacs Lisp library and Lisp code archives. In contrast to the cathedral-building style of the Emacs C core and most other FSF tools, the evolution of the Lisp code pool was fluid and very user-driven. Ideas and prototype modes were often rewritten three or four times before reaching a stable final form. And loosely-coupled collaborations enabled by the Internet, à la Linux, were frequent.

Indeed, my own most successful single hack previous to fetchmail was probably Emacs VC mode, a Linux-like collaboration by email with three other people, only one of whom (Richard Stallman, the author of Emacs and founder of the FSF <*http://www.fsf.org*>) I have actually met (per 8/99). It was a front-end for SCCS, RCS and later CVS from within Emacs that offered "one-

touch" version control operations. It evolved from a tiny, crude sccs.el mode somebody else had written. And the development of VC succeeded because, unlike Emacs itself, Emacs Lisp code could go through release/test/improve generations very quickly.

8. Release early, release often.

Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects, because early versions are almost by definition buggy versions and you don't want to wear out the patience of your users.

This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, then you would only release one every six months (or less often), and work like a dog on debugging between releases. The Emacs C core was developed this way. The Lisp library, in effect, was not—because there were active Lisp archives outside the FSF's control, where you could go to find new and development code versions independently of Emacs's release cycle.

The most important of these, the Ohio State elisp archive, anticipated the spirit and many of the features of today's big Linux archives. But few of us really thought very hard about what we were doing, or about what the very existence of that archive suggested about problems in FSF's cathedral-building development model. I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library. I ran into political trouble and was largely unsuccessful.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there. Linus's open development policy was the very opposite of cathedral-building. The Sunsite (now Metalab <*http://metalab.unc.edu*>) and tsx-11 archives were burgeoning, multiple distributions were being floated. And all of this was driven by an unheard-of frequency of core system releases.

Linus was treating his users as co-developers in the most effective possible way:

8. Release early. Release often. And listen to your customers.

Linus's innovation was not so much in doing this (something like it had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it was not unknown for him to release a new kernel more than once a day! Because he cultivated his base of co-developers and leveraged the Internet for collaboration harder than anyone else, this worked.

But how did it work? And was it something I could duplicate, or did it rely on some unique genius of Linus Torvalds?

I did not think so. Granted, Linus is a damn fine hacker (how many of us could engineer an entire production-quality operating system kernel?). But Linux did not represent any awesome conceptual leap forward. Linus is not

(or at least, not yet) an innovative genius of design in the way that, say, Richard Stallman or James Gosling (of NeWS and Java) are. Rather, Linus seems to me to be a genius of engineering, with a sixth sense for avoiding bugs and development dead-ends and a true knack for finding the minimum-effort path from point A to point B. Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

So, if rapid releases and leveraging the Internet medium to the hilt were not accidents but integral parts of Linus's engineering-genius insight into the minimum-effort path, what was he maximizing? What was he cranking out of the machinery?

Put that way, the question answers itself. Linus was keeping his hacker/users constantly stimulated and rewarded—stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even daily) improvement in their work.

Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable. Linus was behaving as though he believed something like this:

9. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

Or, less formally, "Given enough eyeballs, all bugs are shallow." I dub this: "Linus's Law."

My original formulation was that every problem "will be transparent to somebody." Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. "Somebody finds the problem," he says, "and somebody else understands it. And I'll go on record as saying that finding it is the bigger challenge." But the point is that both things tend to happen rapidly.

Here, I think, is the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you have winkled them all out. Thus, the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena—or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new release. Accordingly, you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

And that is it. That is enough. If "Linus's Law" is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the Linux kernel, should at some point have collapsed under the weight of unforeseen bad interactions and undiscovered "deep" bugs. If it is true, on

the other hand, it is sufficient to explain Linux's relative lack of bugginess and its continuous uptimes spanning months or even years.

And maybe it shouldn't have been such a surprise, at that. Sociologists years ago discovered that the averaged opinion of a mass of equally expert (or equally ignorant) observers is quite a bit more reliable a predictor than that of a single randomly-chosen one of the observers. They called this the "Delphi effect." It appears that what Linus has shown is that this applies even to debugging an operating system—that the Delphi effect can tame development complexity even at the complexity level of an OS kernel.

One special feature of the Linux situation that clearly helps along the Delphi effect a lot is the fact that the contributors for any given project are self-selected. An early respondent pointed out that contributions are received not from a random sample, but from people who are interested enough to use the software, learn about how it works, attempt to find solutions to problems they encounter, and actually produce an apparently reasonable fix. Anyone who passes all these filters is highly likely to have something useful to contribute.

I am indebted to my friend Jeff Dutky <*dutky@wam.umd.edu*> for pointing out that Linus's Law can be rephrased as "Debugging is parallelizable." Jeff observes that although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus, it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

In practice, the theoretical loss of efficiency due to duplication of work by debuggers almost never seems to be an issue in the Linux world. One effect of a "release early and often policy" is to minimize such duplication by propagating fed-back fixes quickly.

Brooks even made an off-hand observation related to Jeff's: "The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. More users find more bugs" (my emphasis).

More users find more bugs because adding more users adds more different ways of stressing the program. This effect is amplified when the users are co-developers. Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem. The "Delphi effect" seems to work precisely because of this variation. In the specific context of debugging, the variation also tends to reduce duplication of effort.

So adding more beta-testers may not reduce the complexity of the current "deepest" bug from the developer's point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow to that person.

Linus coppers his bets, too. In case there are serious bugs, Linux kernel version are numbered in such a way that potential users can make a choice either to run the last version designated "stable" or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet formally imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive.

10. When is s rose not a rose?

Having studied Linus's behavior and formed a theory about why it was successful, I made a conscious decision to test this theory on my new (admittedly much less complex and ambitious) project.

But the first thing I did was reorganize and simplify popclient a lot. Carl Harris's implementation was very sound, but exhibited a kind of unnecessary complexity common to many C programmers. He treated the code as central and the data structures as support for the code. As a result, the code was beautiful but the data structure design ad-hoc and rather ugly (at least by the high standards of this old LISP hacker).

I had another purpose for rewriting besides improving the code and the data structure design, however. That was to evolve it into something I understood completely. It is no fun to be responsible for fixing bugs in a program you do not understand.

For the first month or so, then, I was simply following out the implications of Carl's basic design. The first serious change I made was to add IMAP support. I did this by reorganizing the protocol machines into a generic driver and three method tables (for POP2, POP3, and IMAP). This and the previous changes illustrate a general principle that is good for programmers to keep in mind, especially in languages like C that do not naturally do dynamic typing:

11. Smart data structures and dumb code works a lot better than the other way around.

Brooks, Chapter 9: "Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious."

Actually, he said "flowcharts" and "tables." But allowing for thirty years of semantic/cultural shift, it's almost the same point.

At this point (early September 1996, about six weeks from zero) I started thinking that a name change might be in order—after all, it was not just a POP client any more. But I hesitated, because there was as yet nothing genuinely new in the design. My version of popclient had yet to develop an identity of its own.

That changed, radically, when fetchmail learned how to forward fetched mail to the SMTP port. I will get to that in a moment. But first: I said above that I had decided to use this project to test my theory about what Linus Torvalds had done right. How (you may well ask) did I do that? In these ways:

1. I released early and often (almost never less often than every ten days; during periods of intense development, once a day).
2. I grew my beta list by adding to it everyone who contacted me about fetchmail.
3. I sent chatty announcements to the beta list whenever I released, encouraging people to participate.
4. I listened to my beta testers, polling them about design decisions and stroking them whenever they sent in patches and feedback.

The payoff from these simple measures was immediate. From the beginning of the project, I got bug reports of a quality most developers would kill for, often with good fixes attached. I got thoughtful criticism, I got fan mail, I got intelligent feature suggestions. Which leads to:

12. If you treat your beta-testers as if they are your most valuable resource, they will respond by becoming your most valuable resource.

One interesting measure of fetchmail's success is the sheer size of the project beta list, fetchmail-friends. At time of writing it has 249 members and is adding two to three a week.

Actually, as I revise in late May 1997 the list is beginning to lose members from its high of close to 300 for an interesting reason. Several people have asked me to unsubscribe them because fetchmail is working so well for them that they no longer need to see the list traffic! Perhaps this is part of the normal life-cycle of a mature bazaar-style project.

Popclient Becomes Fetchmail

The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port. I realized almost immediately that a reliable implementation of this feature would make all the other delivery modes next to obsolete.

For many weeks I had been tweaking fetchmail rather incrementally while feeling like the interface design was serviceable but grubby—inelegant and with too many exiguous options hanging out all over. The options to dump fetched mail to a mailbox file or standard output particularly bothered me, but I could not figure out why.

What I saw when I thought about SMTP forwarding was that popclient had been trying to do too many things. It had been designed to be both a mail transport agent (MTA) and a local delivery agent (MDA). With SMTP forwarding, it could get out of the MDA business and be a pure MTA, handing off mail to other programs for local delivery just as sendmail does.

Why mess with all the complexity of configuring a mail delivery agent or setting up lock-and-append on a mailbox when port 25 is almost guaranteed to be there on any platform with TCP/IP support in the first place? Especially when this means retrieved mail is guaranteed to look like normal sender-initiated SMTP mail, which is really what we want anyway.

There are several lessons here. First, this SMTP-forwarding idea was the biggest single payoff I got from consciously trying to emulate Linus's methods. A user gave me this terrific idea—all I had to do was understand the implications.

13. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.

Interestingly enough, you will quickly find that if you are completely and self-deprecatingly truthful about how much you owe other people, the world

at large will treat you like you did every bit of the invention yourself and are just being becomingly modest about your innate genius. We can all see how well this worked for Linus!

(When I gave this paper at the Perl conference in August 1997, Larry Wall was in the front row. As I got to the last line above he called out, religious-revival style,"Tell it, tell it, brother!"The whole audience laughed, because they knew it had worked for the inventor of Perl too.)

After a very few weeks of running the project in the same spirit, I began to get similar praise not just from my users but from other people to whom the word leaked out. I stashed away some of that email; I will look at it again sometime if I ever start wondering whether my life has been worthwhile.

But there are two more fundamental, non-political lessons here that are general to all kinds of design.

14. Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.

I had been trying to solve the wrong problem by continuing to develop popclient as a combined MTA/MDA with all kinds of funky local delivery modes. Fetchmail's design needed to be rethought from the ground up as a pure MTA, a part of the normal SMTP-speaking Internet mail path.

When you hit a wall in development—when you find yourself hard put to think past the next patch—it is often time to ask not whether you have got the right answer, but whether you are asking the right question. Perhaps the problem needs to be reframed.

Well, I had reframed my problem. Clearly, the right thing to do was (1) hack SMTP forwarding support into the generic driver, (2) make it the default mode, and (3) eventually throw out all the other delivery modes, especially the deliver-to-file and deliver-to-standard-output options.

I hesitated over step 3 for some time, fearing to upset long-time popclient users dependent on the alternate delivery mechanisms. In theory, they could immediately switch to forward files or their non-sendmail equivalents to get the same effects. In practice the transition might have been messy.

But when I did it, the benefits proved huge. The cruftiest parts of the driver code vanished. Configuration got radically simpler—no more groveling around for the system MDA and user's mailbox, no more worries about whether the underlying OS supports file locking.

Also, the only way to lose mail vanished. If you specified delivery to a file and the disk got full, your mail got lost. This can not happen with SMTP forwarding because your SMTP listener won't return "OK" unless the message can be delivered or at least spooled for later delivery.

Also, performance improved (though not so you would notice it in a single run). Another not insignificant benefit of this change was that the manual page got a lot simpler.

Later, I had to bring delivery via a user-specified local MDA back in order to allow handling of some obscure situations involving dynamic SLIP. But I found a much simpler way to do it.

The moral? Do not hesitate to throw away superannuated features when you can do it without loss of effectiveness. Antoine de Saint-Exupéry (who was an aviator and aircraft designer when he was not authoring classic children's books) said:

14. "Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away."

When your code is getting both better and simpler, that is when you know it is right. And in the process, the fetchmail design acquired an identity of its own, different from the ancestral popclient.

It was time for the name change. The new design looked much more like a dual of sendmail than the old popclient had; both are MTAs, but where sendmail pushes then delivers, the new popclient pulls then delivers. So, two months off the blocks, I renamed it fetchmail.

There is a more general lesson in this story about how SMTP delivery came to fetchmail. It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too. When your development mode is rapidly iterative, development and enhancement may become special cases of debugging—fixing "bugs of omission" in the original capabilities or concept of the software.

Even at a higher level of design, it can be very valuable to have the thinking of lots of co-developers random-walking through the design space near your product. Consider the way a puddle of water finds a drain, or better yet how ants find food: exploration essentially by diffusion, followed by exploitation mediated by a scalable communication mechanism. This works very well; as with Harry Hochheiser and me, one of your outriders might find a huge win nearby that you were just a little too close-focused to see.

Fetchmail Grows Up

There I was with a neat and innovative design, code that I knew worked well because I used it every day, and a burgeoning beta list. It gradually dawned on me that I was no longer engaged in a trivial personal hack that might happen to be useful to a few other people. I had my hands on a program every hacker with a Unix box and a SLIP/PPP mail connection really needs.

With the SMTP forwarding feature, it pulled far enough in front of the competition to potentially become a "category killer," one of those classic programs that fills its niche so competently that the alternatives are not just discarded but almost forgotten.

I think you can not really aim or plan for a result like this. You have to get pulled into it by design ideas so powerful that afterward the results just seem inevitable, natural, even foreordained. The only way to try for ideas like that is by having lots of ideas—or by having the engineering judgment to take other peoples' good ideas beyond where the originators thought they could go.

Andrew Tanenbaum had the original idea to build a simple native Unix for IBM PCs, for use as a teaching tool. Linus Torvalds pushed the Minix concept

further than Andrew probably thought it could go—and it grew into something wonderful. In the same way (though on a smaller scale), I took some ideas by Carl Harris and Harry Hochheiser and pushed them hard. Neither of us was "original" in the romantic way people think of genius. But then, most science and engineering and software development isn't done by original genius, hacker mythology to the contrary.

The results were pretty heady stuff all the same—in fact, just the kind of success every hacker lives for. And they meant I would have to set my standards even higher. To make fetchmail as good as I now saw it could be, I would have to write not just for my own needs, but also include and support features necessary to others but outside my orbit. And do that while keeping the program simple and robust.

The first and overwhelmingly most important feature I wrote after realizing this was multidrop support—the ability to fetch mail from mailboxes that had accumulated all mail for a group of users, and then route each piece of mail to its individual recipients.

I decided to add the multidrop support partly because some users were clamoring for it, but mostly because I thought it would shake bugs out of the single-drop code by forcing me to deal with addressing in full generality. And so it proved. Getting RFC 822 *<http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt>* parsing right took me a remarkably long time, not because any individual piece of it is hard but because it involved a pile of interdependent and fussy details.

But multidrop addressing turned out to be an excellent design decision as well. Here's how I knew:

15. Any tool should be useful in the expected way but a truly great tool lends itself to uses you never expected.

The unexpected use for multi-drop fetchmail is to run mailing lists with the list kept, and alias expansion done, on the client side of the SLIP/PPP connection. This means someone running a personal machine through an ISP account can manage a mailing list without continuing access to the ISP's alias files.

Another important change demanded by my beta testers was support for 8-bit MIME operation. This was pretty easy to do, because I had been careful to keep the code 8-bit clean. Not because I anticipated the demand for this feature, but rather in obedience to another rule:

16. When writing gateway software of any kind, take pains to disturb the data stream as little as possible—and never *throw away information unless the recipient forces you to!*

Had I not obeyed this rule, 8-bit MIME support would have been difficult and buggy. As it was, all I had to do is read RFC 1652 *<http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1652.txt>* and add a trivial bit of header-generation logic.

Some European users nudged me into adding an option to limit the number of messages retrieved per session (so they can control costs from their expensive phone networks). I resisted this for a long time, and I am still not entirely happy about it. But if you are writing for the world, you have to listen to your customers—this does not change just because they are not paying you in money.

A Few More Lessons from Fetchmail

Before we go back to general software-engineering issues, there are a couple more specific lessons from the fetchmail experience to ponder.

The rc file syntax includes optional "noise" keywords that are entirely ignored by the parser. The English-like syntax they allow is considerably more readable than the traditional terse keyword-value pairs you get when you strip them all out.

These started out as a late-night experiment when I noticed how much the rc file declarations were beginning to resemble an imperative minilanguage. (This is also why I changed the original popclient "server" keyword to "poll.")

It seemed to me that trying to make that imperative mini-language more like English might make it easier to use. Now, although I am a convinced partisan of the "make it a language" school of design as exemplified by Emacs and HTML and many database engines, I am not normally a big fan of "English-like" syntaxes.

Traditionally programmers have tended to favor control syntaxes that are very precise and compact and have no redundancy at all. This is a cultural legacy from when computing resources were expensive, so parsing stages had to be as cheap and simple as possible. English, with about 50 percent redundancy, looked like a very inappropriate model then.

This is not my reason for normally avoiding English-like syntaxes; I mention it here only to demolish it. With cheap cycles and core, terseness should not be an end in itself. Nowadays it is more important for a language to be convenient for humans than to be cheap for the computer.

There are, however, good reasons to be wary. One is the complexity cost of the parsing stage—you do not want to raise that to the point where it is a significant source of bugs and user confusion in itself. Another is that trying to make a language syntax English-like often demands that the "English" it speaks be bent seriously out of shape, so much so that the superficial resemblance to natural language is as confusing as a traditional syntax would have been. (You see this in a lot of so-called "fourth generation" and commercial database-query languages.)

The fetchmail control syntax seems to avoid these problems because the language domain is extremely restricted. It is nowhere near a general- purpose language; the things it says simply are not very complicated, so there is little potential for confusion in moving mentally between a tiny subset of English and the actual control language. I think there may be a wider lesson here:

17. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.

Another lesson is about security by obscurity. Some fetchmail users asked me to change the software to store passwords encrypted in the rc file, so snoopers would not be able to casually see them.

I did not do it, because this doesn't actually add protection. Anyone who has acquired permissions to read your rc file will be able to run fetchmail as you anyway—and if it is your password they are after, they would be able to rip the necessary decoder out of the fetchmail code itself to get it.

All .fetchmailrc password encryption would have done is give a false sense of security to people who do not think very hard. The general rule here is:

18. A security system is only as secure as its secret. Beware of pseudo-secrets.

Necessary Preconditions for the Bazaar Style

Early reviewers and test audiences for this paper consistently raised questions about the preconditions for successful bazaar-style development, including both the qualifications of the project leader and the state of code at the time one goes public and starts to try to build a co-developer community.

It is fairly clear that one cannot code from the ground up in bazaar style. One can test, debug and improve in bazaar style, but it would be very hard to originate a project in bazaar mode. Linus did not try it. I did not either. Your nascent developer community needs to have something runnable and testable to play with.

When you start community-building, what you need to be able to present is a plausible promise. Your program does not have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

Linux and fetchmail both went public with strong, attractive basic designs. Many people thinking about the bazaar model as I have presented it have correctly considered this critical, then jumped from it to the conclusion that a high degree of design intuition and cleverness in the project leader is indispensable.

But Linus got his design from Unix. I got mine initially from the ancestral popclient (though it would later change a great deal, much more proportionately speaking than has Linux). So does the leader/coordinator for a bazaar-style effort really have to have exceptional design talent, or can he get by on leveraging the design talent of others? I think it is not critical that the coordinator be able to originate designs of exceptional brilliance, but it is absolutely critical that the coordinator be able to recognize good design ideas from others.

Both the Linux and fetchmail projects show evidence of this. Linus, while not (as previously discussed) a spectacularly original designer, has displayed a

powerful knack for recognizing good design and integrating it into the Linux kernel. And I have already described how the single most powerful design idea in fetchmail (SMTP forwarding) came from somebody else.

Early audiences of this paper complimented me by suggesting that I am prone to undervalue design originality in bazaar projects because I have a lot of it myself, and therefore take it for granted. There may be some truth to this; design (as opposed to coding or debugging) is certainly my strongest skill.

But the problem with being clever and original in software design is that it gets to be a habit—you start reflexively making things cute and complicated when you should be keeping them robust and simple. I have had projects crash on me because I made this mistake but I managed not to with fetchmail.

So I believe the fetchmail project succeeded partly because I restrained my tendency to be clever; this argues (at least) against design originality being essential for successful bazaar projects. And consider Linux. Suppose Linus Torvalds had been trying to pull off fundamental innovations in operating system design during the development; does it seem at all likely that the resulting kernel would be as stable and successful as what we have?

A certain base level of design and coding skill is required, of course, but I expect almost anybody seriously thinking of launching a bazaar effort will already be above that minimum. The open-source community's internal market in reputation exerts subtle pressure on people not to launch development efforts they're not competent to follow through on. So far this seems to have worked pretty well.

There is another kind of skill not normally associated with software development that I think is as important as design cleverness to bazaar projects—and it may be more important. A bazaar project coordinator or leader must have good people and communications skills.

This should be obvious. In order to build a development community, you need to attract people, interest them in what you are doing, and keep them happy about the amount of work they are doing. Technical sizzle will go a long way towards accomplishing this, but it is far from the whole story. The personality you project matters, too.

It is not a coincidence that Linus is a nice guy who makes people like him and want to help him. It is not a coincidence that I am an energetic extrovert who enjoys working a crowd and has some of the delivery and instincts of a stand-up comic. To make the bazaar model work, it helps enormously if you have at least a little skill at charming people.

The Social Context of Open-Source Software

It is truly written: the best hacks start out as personal solutions to the author's everyday problems, and spread because the problem turns out to be typical for a large class of users. This takes us back to the matter of rule 1, restated in a perhaps more useful way:

19. To solve an interesting problem, start by finding a problem that is interesting to you.

So it was with Carl Harris and the ancestral popclient, and so with me and fetchmail. But this has been understood for a long time. The interesting point, the point that the histories of Linux and fetchmail seem to demand we focus on, is the next stage—the evolution of software in the presence of a large and active community of users and co-developers.

In *The Mythical Man-Month*, Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. He argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. This claim has since become known as "Brooks's Law" and is widely regarded as true. But if Brooks's Law were the whole picture, Linux would be impossible.

Gerald Weinberg's classic *The Psychology Of Computer Programming* supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of "egoless programming," Weinberg observed that in shops where developers are not territorial about their code, and encourage other people to look for bugs and potential improvements in it, improvement happens dramatically faster than elsewhere.

Weinberg's choice of terminology has perhaps prevented his analysis from gaining the acceptance it deserved—one has to smile at the thought of describing Internet hackers as "egoless." But I think his argument looks more compelling today than ever.

The history of Unix should have prepared us for what we are learning from Linux (and what I have verified experimentally on a smaller scale by deliberately copying Linus's methods). That is, that while coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities. 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 feedback exploring the design space, code contributions, bug-spotting, and other improvements come back from hundreds (perhaps thousands) of people.

But the traditional Unix world was prevented from pushing this approach to the ultimate by several factors. One was the legal constraints of various licenses, trade secrets, and commercial interests. Another (in hindsight) was that the Internet was not yet good enough.

Before cheap Internet, there were some geographically compact communities where the culture encouraged Weinberg's "egoless" programming, and a developer could easily attract a lot of skilled kibitzers and co-developers. Bell Labs, the MIT AI Lab, UC Berkeley—these became the home of innovations that are legendary and still potent.

Linux was the first project to make a conscious and successful effort to use the entire world as its talent pool. I do not think it is a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993–1994 that saw the takeoff of the ISP industry and the explosion of mainstream

interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet made possible.

While cheap Internet was a necessary condition for the Linux model to evolve, I think it was not by itself a sufficient condition. Another vital factor was the development of a leadership style and set of cooperative customs that could allow developers to attract co-developers and get maximum leverage out of the medium.

But what is this leadership style and what are these customs? They cannot be based on power relationships—and even if they could be, leadership by coercion would not produce the results we see. Weinberg quotes the autobiography of the 19th-century Russian anarchist Pyotr Alexeyvich Kropotkin's *Memoirs of a Revolutionist* to good effect on this subject:

> Having been brought up in a serf-owner's family, I entered active life, like all young men of my time, with a great deal of confidence in the necessity of commanding, ordering, scolding, punishing and the like. But when, at an early stage, I had to manage serious enterprises and to deal with [free] men, and when each mistake would lead at once to heavy consequences, I began to appreciate the difference between acting on the principle of command and discipline and acting on the principle of common understanding. The former works admirably in a military parade, but it is worth nothing where real life is concerned, and the aim can be achieved only through the severe effort of many converging wills.

The "severe effort of many converging wills" is precisely what a project like Linux requires—and the "principle of command" is effectively impossible to apply among volunteers in the anarchist's paradise we call the Internet. To operate and compete effectively, hackers who want to lead collaborative projects have to learn how to recruit and energize effective communities of interest in the mode vaguely suggested by Kropotkin's "principle of understanding." They must learn to use Linus's Law.

Earlier I referred to the "Delphi effect" as a possible explanation for Linus's Law. But more powerful analogies to adaptive systems in biology and economics also irresistibly suggest themselves. The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. Here, then, is the place to seek the "principle of understanding."

The "utility function" Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. (One may call their motivation "altruistic" but this ignores the fact that altruism is itself a form of ego satisfaction for the altruist.) Voluntary cultures that work this way are not actually uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom explicitly recognizes "egoboo" (the enhancement of one's reputation among other fans) as the basic drive behind volunteer activity.

Linus, by successfully positioning himself as the gatekeeper of a project in

which the development is mostly done by others, and nurturing interest in the project until it became self-sustaining, has shown an acute grasp of Kropotkin's "principle of shared understanding."This quasi-economic view of the Linux world enables us to see how that understanding is applied.

We may view Linus's method as a way to create an efficient market in "egoboo" —to connect the selfishness of individual hackers as firmly as possible to difficult ends that can only be achieved by sustained cooperation. With the fetchmail project I have shown (albeit on a smaller scale) that his methods can be duplicated with good results. Perhaps I have even done it a bit more consciously and systematically than he.

Many people (especially those who politically distrust free markets) would expect a culture of self-directed egoists to be fragmented, territorial, wasteful, secretive, and hostile. But this expectation is clearly falsified by (to give just one example) the stunning variety, quality and depth of Linux documentation. It is a hallowed given that programmers hate documenting; how is it, then, that Linux hackers generate so much of it? Evidently Linux's free market in egoboo works better to produce virtuous, other-directed behavior than the massively-funded documentation shops of commercial software producers.

Both the fetchmail and Linux kernel projects show that by properly rewarding the egos of many other hackers, a strong developer/coordinator can use the Internet to capture the benefits of having lots of co-developers without having a project collapse into a chaotic mess. So to Brooks's Law I counterpropose the following:

20. Provided the development coordinator has a medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

I think the future of open-source software will increasingly belong to people who know how to play Linus's game, people who leave behind the cathedral and embrace the bazaar. This is not to say that individual vision and brilliance will no longer matter; rather, I think that the cutting edge of open-source software will belong to people who start from individual vision and brilliance, then amplify it through the effective construction of voluntary communities of interest.

And perhaps not only the future of open-source software. No closed-source developer can match the pool of talent the Linux community can bring to bear on a problem. Very few could afford even to hire the more than two hundred people who have contributed to fetchmail!

Perhaps in the end the open-source culture will triumph not because cooperation is morally right or software "hoarding" is morally wrong (assuming you believe the latter, which neither Linus nor I do), but simply because the closed-source world cannot win an evolutionary arms race with open-source communities that can put orders of magnitude more skilled time into a problem.

On Management and the Maginot Line

The original "Cathedral and Bazaar" paper ended with the vision above—that of happy networked hordes of programmer/anarchists out competing and overwhelming the hierarchical world of conventional closed software.

A good many skeptics were not convinced, however; and the questions they raise deserve engagement. Most of the objections to the bazaar argument come down to the claim that its proponents have underestimated the productivity-multiplying effect of conventional management.

Traditionally-minded software-development managers often object that the casualness with which project groups form and change and dissolve in the open-source world negates a significant part of the apparent advantage of numbers that the open-source community has over any single closed-source developer. They would observe that in software development it is really sustained effort over time and the degree to which customers can expect continuing investment in the product that matters, not just how many people have thrown a bone in the pot and left it to simmer.

There is something to this argument, to be sure; in fact, I have developed the idea that expected future service value is the key to the economics of software production in "The Magic Cauldron" *<http://www.tuxedo.org/~esr/writings/magic-cauldron/>*.

But this argument also has a major hidden problem: its implicit assumption that open-source development cannot deliver such sustained effort. In fact, there have been open-source projects that maintained a coherent direction and an effective maintainer community over quite long periods of time without the kinds of incentive structures or institutional controls that conventional management finds essential. The development of the GNU Emacs editor is an extreme and instructive example; it has absorbed the efforts of hundreds of contributors over fifteen years into a unified architectural vision, despite high turnover and the fact that only one person (its author) has been continuously active during all that time. No closed-source editor has ever matched this longevity record.

This suggests a reason for questioning the advantages of conventionally-managed software development that is independent of the rest of the arguments over cathedral vs. bazaar mode. If it is possible for GNU Emacs to express a consistent architectural vision over fifteen years, or for an operating system like Linux to do the same over eight years of rapidly changing hardware and platform technology; and if (as is indeed the case) there have been many well-architected open-source projects of more than five years duration—then we are entitled to wonder what, if anything, the tremendous overhead of conventionally-managed development is actually buying us.

Whatever it is certainly does not include reliable execution by deadline, or on budget, or to all features of the specification; it is a rare "managed" project that meets even one of these goals, let alone all three. It also does not appear to be ability to adapt to changes in technology and economic context during the project lifetime, either; the open-source community has proven far more effective on that score (as one can readily verify, for example, by comparing

the thirty-year history of the Internet with the short half-lives of proprietary networking technologies—or the cost of the 16-bit to 32-bit transition in Microsoft Windows with the nearly effortless up-migration of Linux during the same period, not only along the Intel line of development but to more than a dozen other hardware platforms including the 64-bit Alpha as well).

One thing many people think the traditional mode buys you is somebody to hold legally liable and potentially recover compensation from if the project goes wrong. But this is an illusion; most software licenses are written to disclaim even warranty of merchantability, let alone performance—and cases of successful recovery for software nonperformance are vanishingly rare. Even if they were common, feeling comforted by having somebody to sue would be missing the point. You did not want to be in a lawsuit; you wanted working software.

So what is all that management overhead buying? In order to understand that, we need to understand what software development managers believe they do. A woman I know who seems to be very good at this job, says software project management has five functions:

1. To define goals and keep everybody pointed in the same direction.
2. To monitor and make sure crucial details don't get skipped.
3. To motivate people to do boring but necessary drudgework.
4. To organize the deployment of people for best productivity.
5. To marshal resources needed to sustain the project.

Apparently worthy goals, all of these; but under the open-source model, and in its surrounding social context, they can begin to seem strangely irrelevant. We will take them in reverse order.

My friend reports that a lot of resource marshalling is basically defensive; once you have your people and machines and office space, you have to defend them from peer managers competing for the same resources, and higher-ups trying to allocate the most efficient use of a limited pool.

But open-source developers are volunteers, self-selected for both interest and ability to contribute to the projects they work on (and this remains generally true even when they are being paid a salary to hack open source.) The volunteer ethos tends to take care of the "attack" side of resource-marshalling automatically; people bring their own resources to the table. And there is little or no need for a manager to "play defense" in the conventional sense.

Anyway, in a world of cheap PCs and fast Internet links, we consistently find that the only really limiting resource is skilled attention. Open-source projects, when they founder, essentially never do so for want of machines or links or office space; they die only when the developers themselves lose interest.

That being the case, it is doubly important that open-source hackers organize themselves for maximum productivity by self-selection—and the social milieu selects ruthlessly for competence. My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5 percent or so of the programming population. She spends most of her time

organizing the deployment of the other 95 percent, and has observed first-hand the well-known variance of a factor of one hundred in productivity between the most able programmers and the merely competent.

The size of that variance has always raised an awkward question: would individual projects, and the field as a whole, be better off without more than 50 percent of the least able in it? Thoughtful managers have understood for a long time that if conventional software management's only function were to convert the least able from a net loss to a marginal win, the game might not be worth the candle.

The success of the open-source community sharpens this question considerably, by providing hard evidence that it is often cheaper and more effective to recruit self-selected volunteers from the Internet than it is to manage buildings full of people who would rather be doing something else.

Which brings us neatly to the question of motivation. An equivalent and often-heard way to state my friend's point is that traditional development management is a necessary compensation for poorly motivated programmers who would not otherwise turn out good work.

This answer usually travels with a claim that the open-source community can only be relied on to do work that is "sexy" or technically sweet; anything else will be left undone (or done only poorly) unless it is churned out by money-motivated cubicle peons with managers cracking whips over them. I address the psychological and social reasons for being skeptical of this claim in "Homesteading the Noosphere." For present purposes, however, I think it is more interesting to point out the implications of accepting it as true.

If the conventional, closed-source, heavily-managed style of software development is really defended only by a sort of Maginot line of problems conducive to boredom, then it is going to remain viable in each individual application area for only so long as nobody finds those problems really interesting and nobody else finds any way to route around them. Because the moment there is open-source competition for a "boring" piece of software, customers are going to know that it was finally tackled by someone who chose that problem to solve because of a fascination with the problem itself—which, in software as in other kinds of creative work, is a far more effective motivator than money alone.

Having a conventional management structure solely in order to motivate, then, is probably good tactics but bad strategy; a short-term win, but in the long-term a surer lose.

So far, conventional development management looks like a bad bet now against open source on two points (resource marshalling, organization) —and also that it is living on borrowed time with respect to a third (motivation). And the poor beleaguered conventional manager is not going to get any succor from the monitoring issue; the strongest argument the open-source community has is that decentralized peer review trumps all the conventional methods for trying to ensure that details do not get slipped.

Can we save defining goals as a justification for the overhead of conventional software project management? Perhaps; but to do so, we will need good reason to believe that management committees and corporate roadmaps

are more successful at defining worthy and widely-shared goals than the project-specific "benevolent dictators" and tribal elders who fill the analogous role in the open-source word.

That is on the face of it a pretty hard case to make. And it is not so much the open-source side of the balance (the longevity of Emacs, or Linus Torvalds's ability to mobilize hordes of developers with talk of "world domination") that makes it tough. Rather, it is the demonstrated awfulness of conventional mechanisms for defining the goals of software projects.

One of the best-known folk theorems of software engineering is that 60 percent to 75 percent of conventional software projects are either never completed or rejected by their intended users. If that range is anywhere near true (and I have never met a manager of any experience who disputes it) then more projects than not are being aimed at goals which are either (a) not realistically attainable or (b) just plain wrong.

This, more than any other problem, is the reason that in today's software engineering world the very phrase "management committee" is likely to send chills down the hearer's spine—even (or perhaps especially) if the hearer is a manager. The days when only programmers griped about this pattern are long past; "Dilbert" cartoons hang over executives' desks now.

Our reply, then, to the traditional software development manager, is simple—if the open-source community has really underestimated the value of conventional management, why do so many of you display fear, loathing, and contempt for your own process?

Once again the existence of the open-source community sharpens this question considerably—because we have fun doing what we do. Our creative play has been racking up technical, market-share, and mind-share successes at an astounding rate. We are proving not only that we can do better software but that joy is an asset.

Two and a half years after the first version of this article, the most radical thought I can offer to close with is no longer a vision of an open-source-dominated software world; that, after all, looks plausible to a lot of sober people in suits these days.

Rather, I want to suggest what may be a wider lesson about software, (and probably about every kind of creative or professional work). Human beings generally take pleasure in a task when it falls in a sort of optimal-challenge zone; not so easy as to be boring, not too hard to achieve. A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction. Enjoyment tracks efficiency.

Relating to your own work process with fear and loathing (even in a displaced, ironic way) should therefore be regarded in itself as a sign that the process has failed. Joy, humor, and playfulness are indeed assets; it was not mainly for the alliteration that I wrote of "happy hordes" above, and it is no mere joke that the Linux mascot is a cuddly, neotenous penguin.

It may well turn out that one of the most important effects of open source's success will be to teach us that play is the smartest kind of work.

Further Reading

I quoted several bits from Frederick Brooks's classic *The Mythical Man-Month* because, in many respects, his insights have yet to be improved upon. I heartily recommend the 25th Anniversary edition from Addison-Wesley (ISBN 0-201-83595-9), which adds his 1986 "No Silver Bullet" paper.

The new edition is wrapped up by an invaluable twenty-years-later retrospective in which Brooks forthrightly admits to the few judgements in the original text which have not stood the test of time. I first read the retrospective after the first public version of this paper was substantially complete, and was surprised to discover that Brooks attributes bazaar-like practices to Microsoft! (In fact, however, this attribution turned out to be mistaken. In 1998 we learned from the "Halloween Documents" <*http://www.opensource.org/halloween/*> that Microsoft's internal developer community is heavily balkanized, with the kind of general source access needed to support a bazaar not even truly possible.)

Gerald Weinberg's *The Psychology Of Computer Programming* (New York: Van Nostrand Reinhold, 1971) introduced the rather unfortunately-labeled concept of "egoless programming." While he was nowhere near the first person to realize the futility of the "principle of command," he was probably the first to recognize and argue the point in particular connection with software development.

Richard Gabriel, contemplating the Unix culture of the pre-Linux era, reluctantly argued for the superiority of a primitive bazaar-like model in his 1989 paper "Lisp: Good News, Bad News, and How To Win Big." Though dated in some respects, this essay is still rightly celebrated among Lisp fans (including me). A correspondent reminded me that the section titled "Worse Is Better" adumbrates Linux. The paper is accessible on the World Wide Web at <*http://www.naggum.no/worse-is-better.html*>.

De Marco and Lister's *Peopleware: Productive Projects and Teams* (New York: Dorset House, 1987; ISBN 0-932633-05-6) is an under appreciated gem which I was delighted to see Fred Brooks cite in his retrospective. While little of what the authors have to say is directly applicable to the Linux or open-source communities, the authors' insight into the conditions necessary for creative work is acute and worthwhile for anyone attempting to import some of the bazaar model's virtues into a commercial context.

Finally, I must admit that I very nearly called this paper "The Cathedral and the Agora," the latter term being the Greek for an open market or public meeting place. The seminal "agoric systems" papers by Mark Miller and Eric Drexler, by describing the emergent properties of market-like computational ecologies, helped prepare me to think clearly about analogous phenomena in the open-source culture when Linux rubbed my nose in them five years later. These papers are available on the Web at <*http://www.agorics.com/agorpapers.html*>.

Epilog: Netscape Embraces the Bazaar

It is a strange feeling to realize you are helping make history. On 22 January 1998, approximately seven months after I first published "The Ca-

thedral and the Bazaar," Netscape Communications, Inc. announced plans to give away the source for Netscape Communicator <*http://www.netscape.com/newsref/pr/newsrelease558.html*>. I had no clue this was going to happen. Eric Hahn, Executive Vice President and Chief Technology Officer at Netscape, emailed me shortly afterwards as follows: "On behalf of everyone at Netscape, I want to thank you for helping us get to this point in the first place. Your thinking and writings were fundamental inspirations to our decision." The following week I flew out to Silicon Valley at Netscape's invitation for a day-long strategy conference (on 4 Feb. 1998) with some of their top executives and technical people. We designed Netscape's source-release strategy and license together.

A few days later I wrote the following:

> Netscape is about to provide us with a large-scale, real-world test of the bazaar model in the commercial world. The open-source culture now faces a danger; if Netscape's execution does not work, the open-source concept may be so discredited that the commercial world will not touch it again for another decade.
>
> On the other hand, this is also a spectacular opportunity. Initial reaction to the move on Wall Street and elsewhere has been cautiously positive. We are being given a chance to prove ourselves, too. If Netscape regains substantial market share through this move, it just may set off a long-overdue revolution in the software industry. The next year should be a very instructive and interesting time.

And indeed it was. As I write in mid-1999, the development of what was later named "Mozilla" has been only a qualified success. It achieved Netscape's original goal, which was to deny Microsoft a monopoly lock on the browser market. It has also achieved some dramatic successes (notably the release of the next-generation Gecko rendering engine).

However, it has not yet garnered the massive development effort from outside Netscape that the Mozilla founders had originally hoped for. The problem here seems to be that for a long time the Mozilla distribution actually broke one of the basic rules of the bazaar model; they did not ship something potential contributors could easily run and see working. (Until more than a year after release, building Mozilla from source required a license for the proprietary Motif library.)

Most negatively (from the point of view of the outside world) the Mozilla group has yet to ship a production-quality browser—and one of the project's principals caused a bit of a sensation by resigning, complaining of poor management and missed opportunities. "Open source," he correctly observed, "is not magic pixie dust."

And indeed it is not. The long-term prognosis for Mozilla looks rather better now (in July 1999) than it did at the time of Jamie Zawinski's resignation letter—but he was right to point out that going open will not necessarily save an existing project that suffers from ill-defined goals or spaghetti code or any of the software engineering's other chronic ills. Mozilla has managed to simultaneously provide an example of how open source can succeed and how it could fail.

In the meantime, however, the open-source idea has scored successes and found backers elsewhere. 1998 and late 1999 saw a tremendous explosion of interest in the open-source development model, a trend both driven by and driving the continuing success of the Linux operating system. The trend Mozilla touched off is continuing at an accelerating rate.

Acknowledgments

This paper was improved by conversations with a large number of people who helped debug it. Particular thanks to Jeff Dutky *<dutky@wam.umd.edu>*, who suggested the "debugging is parallelizable" formulation, and helped develop the analysis that proceeds from it. Also to Nancy Lebovitz *<nancyl@universe.digex.net>* for her suggestion that I emulate Weinberg by quoting Kropotkin. Perceptive criticisms also came from Joan Eslinger *<wombat@kilimanjaro.engr.sgi.com>* and Marty Franz *<marty@net-link.net>* of the General Technics list. Glen Vandenburg *<glv@vanderburg.org>* pointed out the importance of self-selection in contributor populations and suggested the fruitful idea that much development rectifies "bugs of omission"; Daniel Upper *<upper@peak.org>* suggested the natural analogies for this. I am grateful to the members of PLUG, the Philadelphia Linux User's group, for providing the first test audience for the first public version of this article. Paula Matuszek *<matusp00@mh.us.sbphrd.com>* enlightened me about the practice of software management. Finally, Linus Torvalds's comments were helpful and his early endorsement very encouraging.

Notes

1. In *Programming Pearls*, the noted computer-science aphorist Jon Bentley comments on Brooks's observation with "If you plan to throw one away, you will throw away two." He is almost certainly right. The point of Brooks's observation, and Bentley's, is not merely that you should expect first attempt to be wrong, it is that starting over with the right idea is usually more effective than trying to salvage a mess.
2. John Hasler has suggested an interesting explanation for the fact that duplication of effort does not seem to be a net drag on open source development. He proposes what I will dub "Hasler's Law": the costs of duplicated work tend to scale sub-quadratically with team size—that is, more slowly than the planning and management overhead that would be needed to eliminate them.
3. This claim actually does not contradict Brooks's Law. It may be the case that total complexity overhead and vulnerability to bugs scales with the square of team size, but that the costs from duplicated work are nevertheless a special case that scales more slowly. It is not hard to develop plausible reasons for this, starting with the unquestioned fact that it is much easier to agree on functional boundaries between different developers' code that will prevent duplication of effort than it is to prevent the kinds of unplanned bad interactions across the whole system that underlie most bugs.
4. The combination of Linus's Law and Hasler's Law suggests that there are actually three critical size regimes in software projects. On small projects (I would say one to at most three developers) no management structure more elaborate than picking a lead programmer is needed. And there is some intermediate range above that in which the cost of traditional management is relatively low, so its benefits from avoiding duplication of effort, bug-tracking, and pushing to see that details are not overlooked actually net out positive.

5. Above that, however, the combination of Linus's Law and Hasler's Law suggests there is a large-project range in which the costs and problems of traditional management rise much faster than the expected cost from duplication of effort. Not the least of these costs is a structural inability to harness the many-eyeballs effect, which (as we have seen) seems to do a much better job than traditional management at making sure bugs and details are not overlooked. Thus, in the large-project case, the combination of these laws effectively drives the net payoff of traditional management to zero.

6. Examples of successful open-source, bazaar development predating the Internet explosion and unrelated to the Unix and Internet traditions have existed. The development of the PKZIP compression utility during 1990–1992, primarily for DOS machines, was one such. Another was the RBBS bulletin board system (again for DOS), which began in 1983 and developed a sufficiently strong community that there have been fairly regular releases up to the present (mid-1999) despite the huge technical advantages of Internet mail and file-sharing over local BBSs. While the PKZIP community relied to some extent on Internet mail, the RBBS developer culture was actually able to base a substantial on-line community on RBBS that was completely independent of the TCP/IP infrastructure.