伽藍とバザール
(*The Cathedral and the Bazar*)

Eric S. Raymond　　山形浩生訳

1998/08/11 22:48:58　2000/05/02 翻訳更新

概要

この論文[*1] ではまず、大成功したフリーソフト/オープンソースプロジェクト fetchmail を分析する。このソフトは、Linux の歴史から導かれる、ソフト工学についての意外な理論を試すという意図で実施されたプロジェクトである。本論ではその理論を、二種類の根本的にちがった開発スタイルという形で論じている。一つは FSF やそのまねっ子たちの「伽藍」モデルで、それに対するのが Linux 界の「バザール」モデルだ。この 2 つのモデルが、ソフトのデバッグ作業の性質に関する、正反対の前提からそれぞれ生じていることを示す。続いて Linux 体験に基づき、「目玉の数さえ十分あれば、どんなバグも深刻ではない」という仮説を支持する議論を展開し、利己的エージェントによる自己修正システムとの有益な対比をしてみる。そしてこの洞察がソフトウェアの未来に対して持つ意味について、いくつか考察を行って結論としている。

目次

*1 原文の最新版は `http://www.catb.org/~esr/writings/cathedral-bazaar/` にて各種フォーマットで入手可能。

翻訳:`http://cruel.org/freeware/cathedral.html`

続編 「**ノウアスフィアの開墾**」 (***Homesteading the Noosphere***) も是非参照されたし。

原文:`http://www.catb.org/~esr/writings/homesteading/`

翻訳:`http://cruel.org/freeware/noosphere.html`

続々編 「**魔法のおなべ**」 (***The Magic Cauldron***) も是非参照されたし。

原文:`http://www.catb.org/~esr/writings/magic-cauldron/`

翻訳:`http://cruel.org/freeware/magicpot.html`

パロディ版 ***The Circus Midget and the Dinosour Turd*** （翻訳『**サーカス小人と恐竜うんち化石**』）も一興かと。

原文:`http://missoula.bigsky.net/oxymoron/midgetturd.html`

翻訳:`http://cruel.org/freeware/midgetj.html`

1 伽藍方式とバザール方式

Linux は破壊的存在なり。インターネットのかぼそい糸だけで結ばれた、地球全体に散らばった数千人の開発者たちが片手間にハッキングするだけで、超一流の OS が魔法みたいに編み出されてしまうなんて、ほんの 5 年前でさえだれも想像すらできなかったんだから。

ぼくもできなかった。Linux がぼくのレーダー画面に泳ぎ着いたのは 1993 年の頭だったけれど、その頃ぼくはすでに Unix やフリーソフト開発に 10 年以上も関わってきていた。1980 年代半ば、ぼくは最初期の GNU 協力者の一人だったし、ネット上にかなりのフリーソフトもリリースして、いまでも広く使われているようなプログラムをいくつか（nethack、Emacs VC モードと

GUD モード、xlife など）単独または共同で開発してきた。だから、もうやり方はわかってるもんだと思いこんでいた。

Linux は、ぼくがわかっているつもりでいたものを、大幅にひっくりかえしてくれた。それまでだって、小さなツールや高速プロトタイプ作成、進化的プログラミングといった Unix の福音は説き続けてはいた。でももっと上のレベルでは何かどうしようもない複雑な部分がでてきて、もっと中央集権的で、アプリオリなアプローチが必要になってくるものだとも思っていた。一番だいじなソフト（OS や、Emacs みたいな本当に大規模なツール）は伽藍のように組み立てられなきゃダメで、一人のウィザードか魔術師の小集団が、まったく孤立して慎重に組み立てあげるべきもので、完成するまでベータ版も出さないようでなくちゃダメだと思っていた。

だからリーヌス・トーヴァルズの開発スタイル——早めにしょっちゅうリリース、任せられるものはなんでも任して、乱交まがいになんでもオープンにする——にはまったく驚かされた。静かで荘厳な伽藍づくりなんかない——Linux コミュニティはむしろ、いろんな作業やアプローチが渦を巻く、でかい騒がしいバザールに似ているみたいだった（これをまさに象徴しているのが Linux のアーカイブサイトで、ここはどこのだれからでもソフトを受け入れてしまう）。そしてそこから一貫した安定なシステムが出てくるなんて、奇跡がいくつも続かなければ不可能に思えた。

このバザール方式がどういうわけかまともに機能するらしく、しかもみごとな結果を生むなんて、衝撃以外の何物でもなかった。この世界の様子を学ぶにあたって、ぼくは個別のプロジェクトだけでなく、なぜ Linux 界が混乱のうちに崩壊しないのか、それどころかなぜ、伽藍建設者たちの想像を絶するスピードで、続々と強みを発揮し続けられるのかを理解しようとしてきた。

1996 年半ばには、答がわかりかけてきたような気がした。そしてその頃まったくの偶然から、自分の理論を試してみる完璧な機会がやってきた。意識的にバザール方式で運営できるようなフリーソフトプロジェクトという形で。そこでバザール方式を試してみた——大成功。

というわけでこれから、そのプロジェクトの話をしようではないの。そしてそれを使って、上手なフリーソフト開発についていくつかアフォリズムを提案してみよう。全部が全部、Linux の世界で学んだことばかりではないけれど、そういうものでも Linux 界がすごくいい例になってることがわかるはず。ぼくが正しければ、なぜ Linux コミュニティがこんなにいいソフトを続々と生み出せるのか、みんなにもずばりわかるはず——そしてみんなももっと生産的になれるはずなんだ。

2　なにはともあれメールは通せ

1993 年以来、ぼくはペンシルバニア州ウェストチェスターにある、Chester County InterLink (CCIL) という小さなフリーアクセス ISP の技術面を担当してた (ぼくは CCIL の共同創設者で、ぼくたちが使ってるユニークなマルチユーザ BBS ソフトを書いた。興味があれば、`locke.ccil.org`

に telnet してみてほしい。いまでは 19 回線で 3,000 人弱のユーザをサポートしてる) 。この仕事のおかげで、CCIL の 56K 回線を通じて 1 日 24 時間ネットにアクセスし続けられるようになった——というより、仕事柄そうせざるをえなかったというのが実状かな！

そういうわけで、インターネットの電子メールがすぐに手放せなくなった。なんかややこしい理由で自宅のマシン（`snark.thyrsus.com`）と CCIL とで SLIP 接続するのに手間取って、それがうまくいくと、今度はしょっちゅう `locke` に telnet してメールをチェックするのがえらく面倒になってきた。メールを `snark` に配信してもらって、新しいメールがきたら biff(1) が報せてくれるようにしたかったわけ。

Sendmail の転送機能は使えない。snark はネットにつないでないときもあって、IP アドレスも固定されてないからね。SLIP 接続経由で手をのばして、メールをローカルマシンに引っ張ってきてくれるようなソフトがほしかったわけだ。そういうソフトがあるのは知ってたし、そのほとんどが POP (Post Office Protocol) っていう簡単なアプリケーション・プロトコルを使ってるのも知ってた。そして、確かに locke の BSD/OS には、ちゃんと POP3 サーバソフトが含まれているではないの。

あとは POP3 のクライアントがあればいい。そこでネットで探してみると、3 つか 4 つ見つかった。しばらくは pop-perl を使ってたけれど、これには当然あるべき（と思われる）機能が欠けていた。とってきたメールのアドレスをいじくって、返信がうまくいくようにするのができなかったんだ。

つまりこういうことだ。locke 上の「joe」という人が、ぼくにメールを出したとする。このメールを snark にとってきて、それに返信したら、メールソフトは snark 上の「joe」にそれを送って悦に入っちゃうわけ。そんな人物はいないのに。返信アドレスを手でなおして、`@ccil.org` を最後にくっつけてたんだけれど、これはすぐにえらい手間になってきた。

こんなのどう見ても、コンピュータがやるべきことだよね (実は RFC1123 のセクション 5.2.18 によれば、これは sendmail が処理しなきゃいけないんだけど)。でも、既存の POP クライアントはどれ一つとしてこいつがこなせなかった！　というわけで、教訓その 1:

1. **よいソフトはすべて、開発者の個人的な悩み解決から始まる。**

これは自明のことかもしれない（昔から「必要は発明の母」って言うし)。でも実際のソフト開発者ってのは、お金で横っ面はられて自分では要りもしないし好きでもないようなソフトを一日中シコシコ書いてることがあまりに多いんだ。でも、Linux の世界ではちがう——Linux 界出身ソフトの質が、平均してすごく高いのはこのせいかもしれないね。

そこでぼくは、既存のものとタメを張るようなまっさらの POP3 クライアントを書き上げるべく、即座にコード書きの渦中へ猛然ととびこんだ——かな？　ご冗談を！　ぼくはまず、手元にあ

る POP ユーティリティをじっくりながめてこう考えた。「ぼくの欲しいものにいちばん近いのはどれかな？」　というのも：

2. **何を書けばいいかわかってるのがよいプログラマ。なにを書き直せば（そして使い回せば）いいかわかってるのが、すごいプログラマ。**

——だからね。すごいプログラマを気取るつもりはないけど、でもそのまねくらいはしたい。すごいプログラマの大事な特徴の一つが、建設的な面倒くさがりってヤツなんだ。評価してもらえるのは結果であって、そのための努力じゃないってのがわかってるってこと。そして白紙から始めるよりは、よくできた部分解からはじめたほうがほぼ絶対に楽。

たとえばリーヌス・トーヴァルズ[*2]は、Linux をゼロから書き始めたわけじゃない。386 マシン用の、小さな UNIX っぽい OS だった Minix のコードやアイデアを再利用するところから始めてる。やがて Minix のコードは全部落とされたか、あるいは完全に書き直された——でも最初のうちは、やがて Linux となるべき赤ん坊のための簡単な囲いを提供してくれてたんだ。

同じ精神から、ぼくは既存の POP ユーティリティを探しに出た。そこそこ上手にコーディングされてて、開発のベースに使えるようなヤツを。

Unix 界では、ソース共有の伝統のおかげでコードの再利用が昔からとってもやりやすかった（このせいで GNU プロジェクトは、Unix という OS そのものついては、かなり不満を持ってたんだけれど、ベース OS には Unix を選んだ）。Linux 界は、この伝統を技術的な極限にまでつきつめてる。だれにでも使えるオープンなソースコードが、何テラバイトもある。だからだれかほかの人の、ほとんど使いものになりそうなコードを探すのは、Linux の世界ではほかのどこよりもすごくいい結果をうみやすい。

ぼくの場合もそうだった。もう一度探しに出た結果、最初に見つけたのとあわせて候補が 9 個あがってきた—— fetchpop、PopTart、get-mail、gwpop、pimp、pop-perl、popc、popmail、upop。最初に落ち着いた先は呉承洪 (オー・スンホン）の fetchpop だった。ぼくは自前のヘッダ変更機能をそれに加えて、その他いろいろ改良を入れた。作者はそれを受け入れて、1.9 リリースに含めてくれた。

でも数週間たって、Carl Harris の popcleint のコードに出くわして、困ったなと思った。fetchpop はなかなかいい独創的なアイデア（たとえば daemon モードとか）が入ってたんだけれど、POP3 しか扱えないし、コードもいささか素人くさかった（承洪（スンホン）はプログラマとして才能はあるけれどまだ駆け出しで、その両方の特徴が fetchpop には出ていた）。Carl のコードのほうが優れていて、プロ級のしっかりしたものだったけれど、大事なのに実装が面倒な fetchpop

[*2] http://www.catb.org/~esr/faqs/linus/

の機能がいくつか欠けていた（ぼくがコーディングした機能も含め）。

このままいくか、乗り換えるか？　乗り換えたら、開発ベースはよくなるけれど、かわりにこれまでのコーディングは全部捨てることになる。

実際問題として、乗り換える理由の一つに複数のプロトコルが扱える点があった。POP3 は post-office サーバプロトコルで一番普及はしているけれど、唯一無二ってわけじゃない。Fetchpop をはじめとする競合ソフトは POP2 も RPOP も APOP も扱えなかったし、ぼくのほうでは IMAP[*3] (Internet Message Access Protocol、一番最近に設計された、最強の post-office プロトコル) のサポートを入れたらいいかもしれないな、なんてことをすでにおもしろ半分で考え始めていた。

でも、乗り換えたほうがいいかもしれない理論的な根拠もあった。これはぼくが、Linux よりずっと前に学んだことでもある。こういうことだ：

3. **捨てることをあらかじめ予定しておけ。どうせいやでも捨てることになるんだから（**Fred Brooks『**人月の神話**』[*4]**第** 11 **章）**

あるいは言い換えると、一回とりあえず解決策を実装してみるまでは、問題を完全には理解しきれないってこと。二回目くらいになってやっと、正しい解決法がわかるくらいの理解が得られるかもしれない。だからちゃんとした問題解決をしたいなら、**少なくとも**一回くらいはやりなおす覚悟はしておくこと。[JB]

ま、（と独り言）fetchpop の改良が一回目だったわけだ。というわけで、ぼくは乗り換えた。

最初の popclient 用パッチを 1996 年 6 月 25 日に Carl Harris に送ったんだけれど、実はかれはしばらくまえに、popclient に興味をなくしていることがわかった。コードもほこりをかぶってる状態で、ちょっとしたバグも残ったままだったし。こっちとしては、いろいろ変えたいところもあった。だから、ぼくがこのプログラムを任されるのがいちばん理にかなってるということで、両者はすぐに合意した。

自分でも気がつかないうちに、プロジェクトは拡大したわけだ。もはやぼくは、既存の POP クライアントにちょっとパッチをあてるような話をしてるわけじゃない。プログラムをまるごとメンテする作業を引き受けてたんだ。そして頭の中ではいろいろアイデアも湧いてきていて、これをやったら大幅な変更が必要になるな、というのもはっきりしてた。

コード共有を奨励するソフト文化にあって、これはプロジェクト発展の自然な道筋ではある。ぼくは次の原理を実践していたことになる：

[*3] http://www.imap.org

[*4] 邦訳は巻末文献リスト参照

4. **まともな行動をとってれば、おもしろい問題のほうからこっちを見つけだしてくれる。**

でも Carl Harris の行動のほうがもっと大事だ。かれが理解していたこと：

5. **あるソフトに興味をなくしたら、最後の仕事としてそれを有能な後継者に引き渡すこと。**

なにも相談なんかしなくても、カールもぼくも、自分たちがこの世で最高の問題解決方法を実現するという共通の目標を持っていることがわかっていた。二人にとって唯一の問題は、ぼくがこれを安心して任せられる人物だってことを証明できるかということだけだった。それを実証してみせたら、かれはすぐさま優雅なふるまいを見せて、ソフトをゆだねてくれた。ぼくの順番がきたときにも、Carl と同じくらいの鷹揚さを示したいなと思う。

3　ユーザは大事な財産

というわけで、ぼくは popclient をひきついだ。そして同じくらい大事なことだけれど、ぼくは popclient のユーザベースもひきついだわけだ。ユーザを持つのはすばらしいことで、それは単に、自分が何かニーズに対応してるんだな、なにか役に立つことをしたんだな、ということを実証してくれるからというだけじゃない。きちんと育てれば、ユーザは共同開発者になってくれるんだ。

これまた Unix の伝統の強みで、これまた Linux がみごとに極限までおしすすめる強みでもあるんだけれど、ユーザの中にもハッカーがたくさんいるわけだ。そしてソースコードが公開されてるから、かれらは同じハッカーでも**役に立つ**ハッカーになってくれる。これはデバッグ時間短縮にはすごく役に立つ。ちょっと励ますだけで、ユーザが問題を診断し、直し方を提案してくれて、一人でやるよりずっとはやくコードを改善できるようにしてくれる。

6. **ユーザを共同開発者として扱うのは、コードの高速改良と効率よいデバッグのいちばん楽ちんな方法。**

この効果の力はすごく見落としがち。はっきり言って、ぼくらフリーソフト界のほとんどだれもが、この効果がユーザの数の増加とともにどれほどすごくなるか、そしてそれがシステムの複雑さに対してどれほど有効に機能するかについて、まったく見えてなかった。リーヌスが目を開いてくれるまでは。

はっきり言って、ぼくはリーヌスのいちばん賢い、影響力あるハッキングというのは、Linux のカーネル構築そのものではないと思う。むしろ Linux 開発モデルの発明だと思う。本人の前でこの意見を述べてみたら、かれはにっこりして、これまで何度か言ったことを静かに繰り返した。「ぼくは基本的に怠け者で、ほかの人のしてくれた作業を自分の仕事だと称するのが好きなんだよ」。キツネのようなずるがしこい怠けぶり。あるいはロバート・ハインラインが自作の登場人物の一人

について書いた有名な表現にならえば、「失敗するには怠惰すぎる」。

ふりかえってみると、Linux の手法や成功の前例は GNU Emacs の Lisp ライブラリと Lisp コードアーカイブの開発にみることができる。Emacs の C のコア部分やその他 FSF ツールみたいな、伽藍建築方式にくらべると、Lisp コードのプールの進化は流動的で、すごくユーザ主導で行われた。アイデアやプロトタイプ・モードは、安定した最終形に落ち着くまで 3 回も 4 回も書き直されるのがしょっちゅうだった。そして Linux と同じく、インターネットが可能にしたゆるい協力体制もしばしばとられていた。

確かに、ぼく自身でも fetchmail 以前でいちばんうまくいったハッキングは、Emacs の VC モードだと思う。これは Linux みたいに、電子メールで 3 人と共同作業して開発した。今日にいたるまで、その中で実際に顔をあわせたことがあるのは一人（リチャード・ストールマン、Emacs の作者で FSF[*5]の創始者）だけだ。これは SCCS、RCS、そして後には CVS となったもののフロントエンドで、ワンタッチのバージョンコントロール機能を Emacs の中から使えるようにするものだった。もとにしたのは、だれかが書いた、いい加減でちっちゃな sccs.el モード。そして VC の開発が成功したのは、Emacs 本体とはちがって、Emacs Lisp のコードはリリース/テスト/改良のサイクルをすごくはやく回せるからだった。

4 はやめのリリース、しょっちゅうリリース

はやめにしょっちゅうリリースするのは、Linux 開発モデルの重要な部分だ。ほとんどの開発者（含ぼく）は、プロジェクトがちょっとでも大きくなったらこいつはまずいやり方だと考えていた。初期バージョンはその定義からいってバグだらけだし、ユーザの我慢にも限度があるだろうから。

この信念のおかげで、伽藍建設式の開発への関与も深まった。もし最優先課題が、できるだけ少ないバグしかユーザにお目にかけないということだったら、うん、それならリリースは半年に一度とかにして（あるいはもっと間をおいて）、リリースの間は犬みたいにひたすらバグ取りに専念するだろう。Emacs の C の核部分はこういう形で開発された。Lisp ライブラリは、事実上ちがっていた。FSF のコントロールのきかない活発な Lisp アーカイブがあって、そこにいけば Emacs のリリースサイクルとはまったく関係ない、新しい開発コードが手に入ったから。[QR]

こういうアーカイブのいちばん重要なものの一つは、オハイオ州立大の elisp アーカイブでここは今日の大きな Linux アーカイブの精神や特徴の多くを先取りしたところだった。でも、自分たちがなにをしているのかしっかり考えてみた者はほとんどいなかったし、このアーカイブの存在自体が、FSF 式の伽藍建設型開発モデルの問題点についてなにを示唆しているのかについてもあまり考えなかった。1992 年頃、ぼくはオハイオのコードの相当部分を正式に公式 Emacs Lisp ライ

[*5] http://www.fsf.org/

ブラリに組み込もうとして、かなりまじめに取り組んだ。でも政治的な問題にぶちあたって、ほとんどうまくいかなかった。

でもそれから一年たたないうちに、Linux がかなり目に見えて広まってくると、なにかちがった、ずっと健全なことが起こっているのははっきりしてきた。リーヌスのオープンな開発方針は、伽藍建設の正反対のものだった。Sunsite（現 metalab [*6]や tsx-11 のアーカイブははちきれそうで、パッケージもどんどん登場してきた。そしてそのすべてが、前代未聞の頻度でリリースされるコアシステムに動かされていた。

リーヌスはいちばん効果的なやりかたで、ユーザたちを共同開発者として扱っていたことになる：

7. **はやめのリリース、ひんぱんなリリース。そして顧客の話をきくこと**

リーヌスの革新は、これをやったということじゃない（似たようなことは、もうながいこと Unix の世界の伝統になっていた）。それをスケールアップして、開発しているものの複雑さに見合うだけの集中した取り組みにまでもっていったということだった。開発初期のあの頃だと、リーヌスが新しいカーネルを**一日に**何回もリリースすることだって、そんなに珍しくはなかった。そしてかれは、共同開発者の基盤をうまく育てて、インターネットでうまく共同作業をする点で、ほかのだれよりも上をいっていた。それでうまくいったわけだ。

でも、**具体的にどういうふうに**うまくいってるんだろう。そしてそれはぼくでもまねできるものなんだろうか、それともリーヌスだけにしかない独特な才能に依存したものなんだろうか？

そうは思えなかった。そりゃもちろん、リーヌスはまったく大したハッカーだ（完全な製品レベルの OS カーネルをつくりあげられる人間が、ぼくたちのなかでどれだけいるね？）。でも、Linux はとんでもないソフトウェア思想上の進歩を取り込んだりはしていない。リーヌスは、たとえばリチャード・ストールマンとかジェームズ・ゴスリング（NeWS と Java で有名）のような、設計面での革新的天才ではないんだ（少なくともいまのところは）。むしろリーヌスはエンジニアリングの天才なんじゃないかと思う。バグや開発上の袋小路を避ける第六感と、A 地点から B 地点にたどりつく、いちばん楽な道を見つけだす真の直感もある。Linux の設計はすべて、この特徴が息づいているし、リーヌスの本質的に地道で単純化するような設計アプローチが反映されている。

じゃあ、もし急速リリースと、インターネットの徹底的な使い倒しが偶然ではなくて、労力を最小限ですまそうとするリーヌスのエンジニアリング上の天才的洞察の不可欠な部分だったんなら、かれが最大化しているのは何だったんだろう。この仕組みからかれがひねりだしているのはなんだったんだろう。

[*6] http://metalab.unc.edu/

こういう問題のたてかたをすれば、質問自体が答になる。リーヌスは、ハッカー/ユーザたちをたえず刺激して、ごほうびを与え続けたってことだ。刺激は、全体の動きの中で一員となることでエゴを満足させられるという見込みで、ごほうびは、自分たちの仕事がたえず（まさに毎日のように）進歩している様子だ。

リーヌスは、デバッグと開発に投入される人・時間を最大化することをずばり狙っていたわけだ。コードの安定性が犠牲になったり、なにか深刻なバグがどうしようもなくなったら、ユーザ基盤に見放されるかもしれないという危険をおかしてまでそれをやっていた。リーヌスの行動を見ていると、次のような信念を持っていたんじゃないかと思える：

8. **ベータテスタと共同開発者の基盤さえ十分大きければ、ほとんどすべての問題はすぐに見つけだされて、その直し方もだれかにはすぐわかるはず。**

あるいはもっとくだけた表現だと、「目玉の数さえ十分あれば、どんなバグも深刻ではない」。これをぼくはリーヌスの法則と呼んでる。

はじめにこの法則を書いたときは、どんな問題も「だれかには明白だ」という書き方をしていた。リーヌスはこれに異議を唱えて、問題を理解してそれをなおす人物は、必ずしもどころかふつうは、その問題を最初に記述する人間ではないと言った。「だれかが問題を見つける。そしてそれを理解するのはだれか**別の人**だよ。そしてぼくは、問題を見つけることのほうがむずかしいと述べたことは記録しておいてね」。でも肝心なのは、見つけるのもなおすのも、だいたいすごく短期間で起きるってことだ。

ここに、伽藍建築方式とバザール式のちがいの核心部分があるんだと思う。伽藍建設者的なプログラミングの見方では、バグや開発上の問題はややこしく、潜伏した深い現象だ。問題を全部ほじくりだしたと確信できるようになるには、少数の人が何ヶ月も専念してチェックしなきゃならない。だからリリースの間隔も開いてくるし、長く待たされたリリースが完璧じゃないときには、どうしても失望も大きくなる。

一方のバザール的見方だと、バグなんてほとんどは深刻な現象じゃないという前提にたつことになる——少なくとも、リリースを一つ残らず、千人の熱心な共同開発者が叩いてくれるような状況にさらされたら、どんなバグも早々に浮上してくると考える。よって、たくさんなおしてもらうためにリリースも増やすし、有益な副作用としては、ときどきヘマが出回っちゃっても、あんまり失うものは大きくないってわけ。

そして、これがすべてだ。これだけで必要十分。もしリーヌスの法則がまちがってるなら、Linux カーネルほど複雑なシステム、Linux カーネルくらいみんながよってたかってハッキングしてるようなシステムは、どこかの時点でまずい相互作用や、発見できない「深い」バグのせいで崩壊してたはずなんだ。一方、もしリーヌスの法則が正しければ、これで Linux が相対的にバグが少ないこ

とを十分説明できる。

そしてこれは、そんなに驚くべきことでもなかったのかもしれない。社会学者たちは何年も前に、同じくらいの専門家（あるいは同じくらい無知な人たち）の意見の平均は、そういう観察者の一人をランダムに選んで意見をきくよりも、予測精度がかなり高いことを発見している。これをかれらは「デルファイ効果」と呼んだ。どうやらリーヌスが示したのは、これが OS のデバッグにも適用できるってことみたいだ。つまりデルファイ効果は、OS カーネル級の複雑なものでも、開発上の複雑さをおさえることができるんだ。

Linux の場合の特別な性格で、デルファイ効果的な形でとても役にたっているのは、どんなプロジェクトでもその貢献者は自薦だということだ。初期にコメントをくれた人が指摘してくれたことだけれど、貢献は、ランダムなサンプルから出てくる訳じゃなくて、そのソフトを使うだけの興味を持って、その仕組みを学び、出くわした問題への解決を探そうとして、まあまともそうな解決策を作るだけのことをした人から寄せられる。これだけのフィルタを全部突破してくる人は、貢献できるだけのものは持っている可能性がかなり高い。

Jeff Dutky <dutky@wam.umd.edu>は、リーヌスの法則は「デバッグは並列処理可能だ」と言い換えることもできると指摘してくれた。感謝したい。Jeff の観察では、デバッグするにはデバッガは開発コーディネータと多少のやりとりは必要だけれど、デバッガ同士では大した調整は必要ない。だから、開発者を追加えることで発生する、幾何級数的な複雑性と管理コスト増大という問題には直面しないですむというわけだ。

実際問題として、デバッガたちの作業重複によって生じる理論的な無駄は、Linux の世界ではほとんど問題にされないようだ。「はやめしょっちゅうのリリース」の効果の一つとして、すでにフィードバック済みのバグフィックスをすばやく広めることでそういう重複をなくせるということがある。[JH]

ブルックスは、すでに Jeff の見解に関連したような観察をなにげなく述べてる。「広範に使われるプログラムをメンテナンスするコストは、おおむねその開発コストの 40% だ。驚いたことに、このコストはユーザ数に大きく左右される。**ユーザが増えると見つかるバグも増える**のだ」（強調筆者）。

ユーザが増えると見つかるバグも増えるのは、ユーザを追加することで、プログラムをもっといろんな方法で叩いてみることができるからだ。この効果は、そのユーザたちが共同開発者でもある場合にはさらに増幅される。各人が、ちょっとずつちがったものの見方と分析用ツールキットをもって、その任に当たる。「デルファイ効果」はまさにこの多様性のためにうまく機能するらしい。デバッグという分野に限った話をすると、この多様性のおかげで試みが重複する機会も減るらしい。

だからベータテスタの数を増やしても、**開発者側の**立場からすれば目下の「一番深い」バグの複

雑さが減るわけではないけれど、でもだれかのツールキットがその問題にうまくマッチして、**その人にとっては**そのバグが深刻ではないという可能性を増してくれるわけだ。

リーヌスも、そこらへんは抜け目なくやってる。万が一**本当に**深刻なバグがあったときのために、Linux カーネルのバージョンのナンバリングには工夫がある。ユーザ候補は、「安定」とされたカーネル最新版を使うか、最先端にいって、新しい機能を使うかわりにバグの危険をおかすか、という選択ができるようになってる。この戦術は、ほかの Linux ハッカーたちはまだ正式に採用していないけれど、でも採用されるべきかもしれない。選択肢があるというのは、魅力を増すから。

5　バラがバラでないのは？

リーヌスの行動を研究して、それが成功している理由について理論ができたので、この理論を自分の（確かにずっと単純で小規模な）プロジェクトで試してみようとぼくは意識的に決めた。

でも、まずやったのは popcleint を再構成してすごく単純化することだった。Carl Harris の実装はすごくしっかりしていたけれど、C のプログラマにありがちな、無用な複雑さが見られた。かれはコードを中心に考えていて、データ構造はコードのサポートとして扱っていた。結果として、コードは美しかったけれど、データ構造のデザインはいきあたりばったりで、いささか醜かった（少なくともこの老いぼれ LISP ハッカーの高い基準で見れば）。

でも、書き直しをやったのは、コードやデータ構造の設計を改善する以外にも目的があった。それは、このソフトを進歩させて、自分が完全に理解してるものにすることだった。自分でもわかってないプログラムのバグをなおす責任をしょいこむなんて、おもしろくもないからね。

そして最初の 1 ヶ月かそこらは、単に Carl の基本的な設計の考え方を追いかけてただけだった。ぼくが加えた最初の大きな変更は、IMAP のサポートを加えることだった。これは、プロトコルマシンを、汎用ドライバとメソッドテーブル 3 つ（POP2、POP3、IMAP 用）に再構成することで実現した。これと、その前の変更は、プログラマとして頭にいれておくといい一般原則を示すものだ。特に、ダイナミックなタイプ処理をしない C みたいな言語では：

9. **賢いデータ構造と間抜けなコードのほうが、その逆よりずっとまし。**

またもやフレッド・ブルックス本の第 11 章から。「コードだけ見せてくれてデータ構造は見せてもらえなかったら、わたしはわけがわからぬままだろう。データ構造さえ見せてもらえれば、コードのほうはたぶんいらない。見るまでもなく明らかだから」

ほんとはかれが言ったのは「フローチャート」に「テーブル」だった。でも 30 年にわたる用語面・文化面での推移を考慮すれば、ほとんど同じことを言ってる。

この時点（1996 年 9 月頭、ゼロ時点から約 6 週間後）で、ぼくはそろそろ名前の変え時かな、と

考え出した。なんといっても、もう POP クライアントだけじゃなくなってたんだし。でも、ためらった。いまのところ、まだこのソフトにはまったく新しい部分が何もなかったからだ。ぼく版の popclient は、まだ独自のアイデンティティを確立するにいたってなかった。

これが派手に変わったのは、fetchmail がとってきたメールを SMTP ポートに転送する方法を身につけたときだった。この話はまたあとで。それよりまず：上で、このプロジェクトを使って、リーヌス・トーヴァルズがうまくやった点についての自分の理論を試すことにした、と書いた。試すって、どういうふうに？（という疑問は当然起こるだろう）。それは以下の通り：

1. はやめしょっちゅうのリリースを心がけた（間が 10 日以上開いたことはほとんどない。集中して開発しているときは、1 日 1 回）。
2. だれかが fetchmail の件で連絡してきたら、その人をベータリストに加えてリストを増やした。
3. リリースごとに騒々しいアナウンスをベータリストに送りつけて、みんなに参加をうながした。
4. そしてベータテスタたちの言うことをきいて、設計上の判断について意見を求め、パッチやフィードバックを送ってくれたら必ずほめた。

こういう単純な方法の見返りはすぐにやってきた。プロジェクトの始めから、ぼくは他の開発者なら死んでもいいと思うような質の高いバグレポートをもらったし、しかもそれになかなかいいフィックスまでついてきた。よく考えられたコメントももらったし、ファンレターもきたし、賢い機能の提案ももらった。これでわかるのが：

10. **ベータテスタをすごく大事な資源であるかのように扱えば、向こうも実際に大事な資源となることで報いてくれる。**

Fetchmail の成功をはかるおもしろい指標としては、このプロジェクトのベータリスト——fetchmail 友の会——のサイズを見るといい。執筆時点では 249 人で、毎週 2、3 人追加されている。

実は、1997 年 5 月に改訂している時点だと、このリストは人数が減りはじめてる。その理由がおもしろい。何人かがリストから外してくれといってきたんだけれど、それは fetchmail がかれらにはまったく文句なしに機能しているので、メーリングリストのトラフィックを見る必要がないと言うんだ。成熟したバザール形式のライフサイクルでは、これが自然なのかも知れない。

6 Popclient から Fetchmail へ

このプロジェクトの真のターニングポイントは、Harry Hochheiser がクライアント機の SMTP ポートにメールを転送するための書きかけのコードを送ってきてくれたときだった。ぼくはほとんど即座に、この機能を信頼できる形で実装できたら、ほかの配信モードはほとんど時代遅れ同然になるなと気がついた。

何週間にもわたって、ぼくは fetchmail にいろいろ追加する形でいじってきていた。でもその間、インターフェースのデザインが使えなくはないけれど、ちょっと野暮ったいなと感じだしていた——エレガントじゃないし、貧弱なオプションがそこらじゅうにぶらさがってるし。とってきたメールをメールボックスファイルや標準出力にダンプするオプションがことさら気に入らなかったけれど、その理由が自分でもわからなかった。

SMTP 転送について考えてみたときに気がついたのは、popclient はいろいろやろうとしすぎてるということだった。これはメール配送エージェント（MTA）とローカル配信エージェント（MDA）の両方をこなすよう設計されていた。SMTP 転送があれば、MDA の仕事からは足を洗って、メールのローカル配信はほかのソフトにまかせればいい。ちょうど sendmail がやってるように。

メール配信エージェントのややこしい設定なんか、しなくたっていいじゃないか。メールボックスをロックして追加なんて、しなくていいじゃないか。ポート 25 は、TCP/IP サポートのあるプラットホームなら、まずまちがいなくそこにあるんだから。特にこうすれば、とってきたメールは確実に、ふつうの送り手から送られてきた SMTP メールのように見えるはずなんだ。それがもともとぼくたちの求めているものだろう。

ここにはいくつか教訓がある。まず、この SMTP 転送のアイデアは、ぼくがリーヌスのやりかたを意識的に真似ようとした最大の見返りだった。あるユーザがすばらしいアイデアを提供してくれた——ぼくは単に、その意義を理解すればよかっただけ。

11. **いいアイデアを思いつく次善の策は、ユーザからのいいアイデアを認識することである。時にはどっちが次善かわからなかったりする。**

おもしろいことに、もし自分が他人に負うところがいかに大きいかについて、完全かつ謙虚なくらいに正直でいたら、世の中はその発明が全部あなた自身のもので、単に自分の天才ぶりについて謙遜しているだけだと見なしてくれる。これがリーヌスの場合にどんなにうまくいってるか、みんなもごらんの通り！

（1997 年 8 月にこの論文を Perl の会議で発表したとき、最前列に Larry Wall がいた。ぼくがこ

の直前のくだりにさしかかるとかれが野次をとばした。キリスト再来信者スタイルで、「そうだそうだ、言ってやれ、兄弟！」と。全聴衆が笑った。みんなこれが、Perl 発明者にとってもうまくいったのを知っていたからだ。）

そしてその同じ精神でプロジェクトを運営してほんの数週間もしないうちに、ユーザたちからだけでなく、それを伝え聞いたほかの人たちからも、同じような賞賛のメールが届くようになった。そういうメールはいくつかとってある。いつか、自分の人生なんて何の価値もなかったんじゃないかと思うようになったら、またそれを読みかえそうっと:-)。

でもここには、もっと本質的で政治的でない教訓があと 2 つある。これはあらゆるデザインに一般化して言えることだ。

12. **自分の問題のとらえかたがそもそも間違っていたと認識することで、もっとも衝撃的で革新的な解決策が生まれることはよくある。**

ぼくは popclient を、MTA/MDA の複合物として開発し、いろんな気の利いたローカル配信モードをくっつけたものにしようとし続けていた。これは解決すべき問題をまちがえていたんだ。Fetchmail の設計は、純粋な MTA として最初から考え直す必要があった。通常の SMTP を話すインターネットメールパスの一部として。

開発で壁にぶちあたったとき——次のパッチ以降何をしたらいいか、すごく悩んでしまうとき——というのは、自分が最終的な回答にたどりついたのかな、と考えるべきときではなく、むしろ自分が正しい質問をしているのかな、と考え直してみるべきときであることが多い。ひょっとして、問題をとらえなおしてみる必要があるのかもしれない。

というわけで、問題をとらえなおした。明らかに、やるべき正しいことというのは：

1. 汎用ドライバに SMTP 転送サポートをハックして入れ込む
2. それをデフォルトモードにする
3. やがてはその他の配信モードをすべてうっちゃる。特にファイルへの配信と標準出力への配信は始末する

ということだった。

この 3 番目については、しばらくためらった。ながいこと popclient を使ってきて、他の配信メカニズムに頼っている古参ユーザたちを怒らせるんじゃないかと思ったからだ。理論的には、かれらは即座に.forward ファイルか、sendmail 以外でもそれに相当する仕組みを使うことで、同じ結果を得ることができる。でも実際問題としては、この移行はえらく手間がかかることになるかもしれない。

でも実際にやってみたら、利点のほうが大きく出てきた。ドライバのコードのいちばんいやらし

い部分が消えた。設定もむちゃくちゃに簡単になった——システム MDA やユーザのメールボックスを探し回ったりして悩むこともなくなったし、下敷きになってる OS がファイルのロッキングをサポートしているか心配する必要もない。

さらに、メールをなくす唯一の方法も消え失せた。もしファイルへの配信を指定してディスクがいっぱいになったら、メールは消えてしまう。これは SMTP 転送では絶対に起きない。SMTP のリスナーは、メッセージが配信できるか、少なくとも後の配信用にスプールできる場合にしか OK を返してよこさないからだ。

さらに、速度も向上（もっともこれは一回走らせたくらいではわからないけれど）。もうひとつバカにできないメリットとして、man ページがすごくシンプルになった。

あとで、ダイナミック SLIP がらみの珍しい状況を処理できるようにするため、ユーザ指定のローカル MDA 経由の配信は復活させなきゃならなかった。でも、これももっと簡単にこなす方法を見つけた。

教訓は？　古びてきた機能は迷わず捨てること——有効性を下げずに捨てられる場合には。アントワーヌ・サンテグジュペリ（かれは児童書の古典を書いていないときには、飛行家で航空機設計家だった）曰く：

13. **「完成」（デザイン上の）とは、付け加えるものが何もなくなったときではなく、むしろなにも取り去るものがなくなったとき。**

コードがどんどんよくなって、しかも同時に単純になってきたら、それはもう**絶対に**自分が正しいのがわかる。そしてこの過程で、fetchmail のデザインは、先祖の popclient とは別の独自のアイデンティティを獲得してきた。

そろそろ名前を変える頃だった。新しいデザインは、以前の popclient よりずっと sendmail の双子みたいな感じになってきていた。どっちも MTA だけれど、sendmail がプッシュして配信するのに対して、新しい popclient はプルして配信する。だから引き継いで 2 ヶ月したところで、ぼくはこれを fetchmail と改名した。

SMTP 配信が fetchmail になったこのお話には、もっと一般的な教訓がある。並列処理可能なのは、デバッグだけじゃない。開発と、（かなり驚く規模で）デザイン空間の開拓も並列処理ができるってことだ。開発モードが高速なやりとりに基づくものになっていると、開発と拡張はデバッグの特殊なケースになってくる——つまり、もとのソフトの機能やコンセプトにおける「見過ごしというバグ」をなおすことになるわけ。

もっと高次のデザインでも、自分の製品近くのデザイン空間を、多くの共同開発者たちがランダムウォークしつついろいろ考えてくれていると、とても役にたつことが多い。水たまりがドブに流れていくやりかたとか、あるいはもっといいのは、アリが食物を見つけるやり方を考えてほしい。

基本的には、拡散による探索をして、その後で、スケーラブルな通信メカニズムによって、探索の結果を取り尽くす。これは実にうまく機能するんだ。そして Harry Hochheiser とぼくの場合と同じく、きみの周縁ライダーたちが、近くの巨大な勝利を見つけてくることは十分にありえる。自分では、近くばかり見過ぎていてちょっと気がつかなかったようなやつをね。

7　Fetchmail の成長

そういうわけで、きれいで革新的なデザインを手に入れ、毎日自分で使ってるから確実に動くのもわかってるコードもできたし、さらにベータリストは拡大する一方。だんだんわかってきたのは、自分がやっているのが、ほんの数人にたまたま便利に使えるような、自分だけのちょっとしたハッキングなんかではなくなってきた、ということだった。ぼくがいじっているのは、Unix マシンと SLIP/PPP メール接続を持ってるハッカーみんなが本当に必要としているプログラムだったんだ。

SMTP 転送機能のおかげで、このソフトは競合ソフトからぬきんでて、「カテゴリーキラー」になる可能性まで出てきた。カテゴリーキラーというのは、ニッチをあまりに見事に満たしているので、それ以外の選択肢は単に放棄されるどころか、ほとんど忘れ去られてしまうようなソフトのことだ。

こういう結果は、狙ってできるものではないし、計画して得られるものでもないと思う。ものすごく強力なデザイン上のアイデア、あとから考えると、その結果が当然きわまりなく、自然で、事前に約束されていたようにさえ思えるようなアイデアによって、そういう結果に引き込まれなくてはならないんだと思う。そんなアイデアを狙うには、たくさんアイデアを思いつくしかない——または、ほかのひとのいいアイデアをもらって、それをもとの発案者が思ってもみなかったところまでつきつめるというエンジニアリング上の判断を持つという方法か。

Andrew Tanenbaum は、386 用に簡単なネイティブの Unix をつくるというアイデアを最初におもいついた。これは教育用ツールとしてだった。リーヌス・トーヴァルズは Minix のコンセプトを、Andrew が予想もしなかったくらい遙か遠くにまで拡張した——そしてそれが、すばらしいものへと成長した。同じように（もっともスケールは小さいけれど）ぼくは Carl Harris と Harry Hochheiser のアイデアをもとにして、それをつきつめた。ぼくらのいずれも、みんなが天才というものを考えるロマンチックな意味では「独創的」じゃなかった。でも、科学も工学もソフト開発も、ほとんどは独創的な天才の手になるものではないんだ。ハッカー神話がなんと言おうとも。

でも結果はそれでもなかなか大した代物だった——それどころか、あらゆるハッカーが死んでもいいと思うような大成功！　そしてこれはつまり、ぼくは自分のねらいをさらに高く設定しなきゃいけないということを意味する。fetchmail を、いまの自分に見える可能性くらいにまで優れたも

のにするためには、自分のニーズのためだけにコードを書くのではなく、自分の守備範囲ははずれていても他人が必要としているような機能を加え、サポートしなくてはならなくなった。そして同時に、プログラムをシンプルで堅牢にしておく必要もあった。

これを認識してから書いた初の、そして圧倒的にいちばん重要な機能は、マルチドロップのサポートだった。これはつまり、複数ユーザ宛のメールが全部たまっているメールボックスからメールをとってきて、それぞれのメールを個別受信者に選り分ける機能だ。

マルチドロップのサポートを足すことにしたのは、一つには一部のユーザがしつこくせがんだこともある。でも最大の理由は、アドレッシングを最大限に一般化して対応せざるを得なくなることで、シングルドロップのコードのバグを全部ひねりつぶせるだろうと思ったからだ。そしてそれは立派に実証された。RFC 822[*7]の解析をきちんと実装するには、えらく長い時間がかかった。これは個別部分が特にむずかしかったからではなく、相互に依存しあった面倒な細部を山ほど片づけなくてはならなかったからだ。

でもマルチドロップアドレッシングは、ふたをあけてみたらこれまたすばらしい設計上の決定だった。なぜそう思ったかというと：

14. **ツールはすべて期待通りの役にたたなきゃダメだが、すごい ツールはまったく予想もしなかったような役にもたってしまう。**

——からだ。マルチドロップ fetchmail の予想もしない利用法は、メーリングリストを運用する際に、リストの管理や alias の展開を、SLIP/PPP 接続の**クライアント側で**できちゃえることだった。これはつまり、ISP のアカウント経由で個人マシンを走らせてる人でも、ISP の alias ファイルに絶えずアクセスすることなしにメーリングリストを運用できるってことだ。

もう一つ、ベータテスタたちが要求してきた重要な変更は、8 ビット MIME のサポートだった。これはすごく楽だった。ぼくは自分のコードを 8 ビットクリーンにするように心がけてきたからだ。これは、こういう機能への要望を予想してたからではない。こんな別のルールに従ったまでのことだった：

15. **ゲートウェイソフトを書くときはいかなる場合でも、データストリームへの干渉は最低限におさえるように必死で努力すること。そして受け手がわがどうしてもと言わない限り、絶対に情報を捨てないこと！**

この規則を守っていなかったら、8 ビット MIME サポートはむずかしくてバグだらけになっただろう。でも、ぼくは守っていたので、RFC 1652[*8] を読んで、ほんのちょっとしたヘッダ生成の

[*7] http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt
[*8] http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1652.txt

ロジックを加えるだけですんだ。

ヨーロッパのユーザの一部は、セッションあたりにとってこられるメッセージ数を制限するオプションをつけるようにしつこくせがんだ（電話代が高いので、それを抑えるためだ）。ぼくは長いことこれを拒否してきたし、いまでも完全に満足しているとはいいがたい。でも、世界のために書いているんなら、顧客には耳を傾けなきゃ——かれらがお金で支払ってるんじゃなくても、これは変わらない。

8 続・Fetchmail の教訓

一般的なソフトウェア工学の問題に戻る前に、fetchmail の経験からもう少し教訓を引っぱり出して考察しておこう。

rc ファイルの構文は、オプションの「ノイズ」キーワードを持っていて、これはパーサーに完全に無視される。このキーワードのおかげで英語に似た構文が使えるようになるので、それを全部はぎとってできるような、従来の無味乾燥なキーワードと値の対応表よりはずっと読みやすくなっている。

これはそもそも、ある晩遅くにちょっと始めた実験が発端だった。rc ファイルの宣言が、命令形のミニ言語にずいぶん似てきたのに気がついたのだ（そしてこのせいで、もとの popclient のキーワード「サーバ (server)」を「チェックせよ（poll)」に変えた)。

この命令形のミニ言語をもっと英語風にしてみたら、使いやすくなりそうだと思った。さて、ぼくは Emacs や HTML や各種データベースエンジンに代表される「なんでも言語化せよ」式デザイン派閥の意識的な急進派ではあるけれどふつうは「英語っぽい」構文はあまり好きではない。

伝統的にプログラマは、厳密でコンパクトで冗長性のまったくない制御構文を好む傾向にあった。これはコンピュータ資源が高価だった時代の文化的な名残だ。当時は構文解析段階は、できる限り安く単純でなきゃならなかったから。英語は 50% くらい冗長性を持っているので、当時はすごく不適切なモデルに見えた。

英語っぽい構文を避けるべき理由としてぼくが挙げたいのはこういうことではない。ここでこれを挙げたのは、単にそれを却下するためだ。サイクルやコアが安くなってきたら、無味乾燥がそれ自体で目的化してはならない。いまでは、言語はコンピュータにとって安あがりになるよりも、人間にとって便利なほうが大事なのだ。

でも、慎重になるべきまともな理由はある。一つは複雑さと、構文解析段階のコストだ——あんまり複雑にすると、それ自体がバグのもとになったりユーザの混乱を招いたりしかねない。別の理由として、言語の構文を英語っぽくしようとすると、しばしばその言語の使う「英語」はとんでもなく歪んだ代物になってりまい、これが高じると、そういうわざとらし自然言語との類似は伝統的

な構文と同じくらい混乱をまねくことになる（これはいろんな通称 4GL や商業データベースキュエリー言語で見かける）。

Fetchmail の制御構文は、どうやらこういう問題を免れている。それは、言語ドメインがすごく制限されているからだ。汎用言語にはほど遠い。それが表現していることは、とにかく大して複雑ではないので、英語の小さなサブセットと実際の制御言語の間を行き来するのに、精神的な混乱を起こす可能性があまりない。ここにはもっと広い教訓があるかもしれない。

16. **自分の言語がチューリング的完成からほど遠い場合には、構文上の甘さを許すといろいろ楽になるかもね。**

別の教訓は、隠すことでセキュリティを高めるという点についてのものだった。Fetchmail ユーザの一部は、ソフトの仕様を変えて、パスワードを暗号化して rc ファイルに保存するようにしてくれと要求してきた。のぞき屋たちが気軽にそれをのぞいたりできないようにしてほしいから、と言って。

これはやらなかった。これでは実は、セキュリティはぜんぜん高まらないからだ。rc ファイルを読む許可を与えられている人間なら、だれでも fetchmail をどのみちあなたと同じように好き勝手に動かせてしまうんだから——そしてそいつがあなたのパスワード目当てなら、fetchmail のコードそのものから必要なデコーダをぬきだして、ファイルを解読して盗むことができてしまう。

だから.fetchmailrc パスワード暗号化なんかしても、ものごとをつきつめて考えない人たちに、セキュリティが高まったかのようなまちがった幻想を与えるだけだ。ここでの一般原則は以下の通り：

17. **セキュリティシステムのセキュリティは、そこで使われてる秘密の安全性にかかっている。見かけだけの秘密は要注意。**

9 バザール方式の前提条件とは

この論文の初期レビューアーや、試験読者たちがたえず返してきた質問というのは、上手なバザール形式の開発に必要な条件は何か、というものだった。これはプロジェクトリーダーの資質と、共同開発者コミュニティをつくろうとしてコードを公開する時点での、コードの状態についての条件の両方についてのものだった。

バザール形式で最初からコードを書くのが無理だというのは、まあはっきりしているだろう[IN]。バザール形式でテストしたりデバッグしたり改善したりはできるけれど、プロジェクトを**最初からバザール式で始めるのは**すごくむずかしいだろう。リーヌスはそんなことはしなかったし、

ぼくもしなかった。あなたが生み出そうとしてる開発者コミュニティは、いじるために何か動いてテストできるものを必要としているんだ。

コミュニティ形成を始めるときには、まずなによりも**実現できそうな見込み**を示せなきゃならない。別にそのソフトは特によく書けてなくてもいい。雑で、バグだらけで、不完全で、ドキュメント皆無でもいい。でも絶対不可欠なのが、開発者候補たちに、それが目に見える将来にはなにか本当に使える代物に発展させられると説得できることだ。

Linux と fetchmail は、どちらも強力で魅力的な基本デザインをもって公開された。ぼくが提出したバザールモデルについて考えてきた人の多くは、これがきわめて重要だということを正しく認識し、そこからいきなり、だったらプロジェクトリーダーには高度なデザイン上の直感と才能が必要にちがいないという結論に一足飛びにとびついてしまった。

でもリーヌスはデザインを Unix からもらってる。ぼくはもともと先祖の popmail からアイデアを得てる（もっともそれは後に大きく変わった。割合から言えば Linux よりはずっと大きな変化だ）。ということは、バザール形式のリーダー/コーディネーターはずばぬけたデザインの才能が本当にいるんだろうか、それとも他人のデザインの才能をうまく活かすだけでやっていけるんだろうか。

コーディネーターが、とてつもないデザイン上のひらめきを自分で得る必要性は必ずしもないと思う。でも、絶対に必要なのは、その人物が**ほかの人たちのよいデザイン上のアイデアを認識できる**ということだ。

Linux も fetchmail も、この証拠を示している。リーヌスは（すでに述べた通り）、驚異的に独創的な設計者ではないけれど、よいデザインを認識してそれを Linux カーネルに組み込む強力な第六感を示した。そしてすでに述べたように、fetchmail 最大の強力なデザインアイデア（SMTP 転送）は他人にもらったものだった。

この論文を早い時期に読んだ人たちは、おまえはバザールプロジェクトでのデザイン上の独創性を過小評価している、自分にはいろいろアイデアがあるもんだから、それが当然のことだと思ってるんだろう、と誉めてくれた。確かにこれは一理ある。ぼくの最大の強みは確かに、コーディングやデバッグではなく、デザイン能力にある。

でもソフトの設計で、小利口で独創的になることの問題点は、それが習慣になってしまうことだ。ソフトは堅牢でシンプルにしておかなきゃダメなのに、反射的にそれを媚びた複雑なものにしてしまいがちになる。このまちがいのおかげでつぶれたプロジェクトもいくつかある。でも、fetchmail ではそういうことにならずにすんだ。

だから fetchmail プロジェクトが成功したのは、一部はぼくが小利口になりがちな自分の性格を抑えたからだと思う。これは（少なくとも）バザールプロジェクトの成功にデザイン上の独創性が不可欠という議論の反証になっている。そして Linux もそうだ。もしリーヌス・トーヴァルズが

開発途上で根本的な OS デザインの革新をやってのけようとしていたら、その結果のカーネルがいまのものほど安定してうまくいっていたかどうか？

一定レベルのデザインとコード書き能力は必要だけれど、でもバザール式のプロジェクトを始めようかと真剣に考えている人なら、ほとんどだれでもそんな最低限以上の能力はあるだろう。フリーソフト/オープンソースコミュニティ内における評判の市場は、最後まで面倒を見られないような開発プロジェクトを始めないように、みんなに微妙な圧力をかける。いまのところ、これはなかなかうまく機能してきたようだ。

別の才能で、ソフト開発とはふつうは関連づけられないけれど、でもバザールプロジェクトではデザイン上の才覚に匹敵するほど——あるいはそれ以上——重要なものがあると思う。バザールプロジェクトは、コーディネータやリーダの対人能力やコミュニケーション能力が優れていないとダメだ。

これは説明するまでもないだろう。開発コミュニティをつくるには、人を引きつける必要がある。自分のやっていることに興味を持たせて、各人のやっている仕事量についてみんなが満足しているように気を配る必要がある。技術的な先進性は、これを実現する役にはおおいに立つけれど、でもそれだけではぜんぜん足りない。その人が発する個性も大事だ。

リーヌスがナイスガイで、みんなかれを気に入って手伝いたくなってしまうのは、偶然ではない。ぼくがエネルギッシュで外向的で、大人数を動かすのが好きで、コメディアンの話術や本能をちょっと備えているのも偶然じゃない。バザールモデルが機能するためには、人を魅了する能力が少しくらいでもあると、きわめて役に立つのだ。

10　フリーソフトの社会的な意義

これはもう不動の真実だ。最高のハックは、作者の日常的な問題に対する個人的な解決策として始まる。そしてその問題が、実は多数のユーザにも典型的なものであるために広まる。これでルールその 1 の話に戻ってきた。ただしもう少し便利な形で言い直してみよう。

18. **おもしろい問題を解決するには、まず自分にとっておもしろい問題を見つけることから始めよう。**

Carl Harris とかれのかつての popclient もそうだったし、ぼくの fetchmail もそうだ。でもこれは長いこと理解されてきた。おもしろい点、つまり Linux と fetchmail の歴史がぼくたちの目をいやでも向ける点は、次の段階だ——ユーザと共同開発者たちの巨大で活発なコミュニティがある中で、ソフトがどう発展するかという話。

『人月の神話』でフレッド・ブルックスはプログラマの時間が代替不能だと看破している。遅れているソフト開発に開発者を加えても、開発はかえって遅れる。プロジェクトの複雑さとコミュニケーションコストは、開発者数の 2 乗で増大するのに対し、終わる作業は直線的にしか増加しないというのがかれの議論だった。この論はそれ以来「ブルックスの法則」と呼ばれるに至り、真実をついているものとだれもが考えている。でもブルックスの法則が唯一無二の真理なら、Linux はあり得なかっただろう。

数年後、ジェラルド・ワインバーグの古典『プログラミングの心理学』が、いまにして思えばブルックスに対する重要な訂正だったものを提供してくれた。「エゴのないプログラミング」を論じるなかで、ワインバーグが述べたのは、開発者たちが自分のコードを私物化せず、ほかのみんなにバグを探したり改良点を見つけたりするよう奨励するようなところでは、ソフトの改善がほかよりも劇的にはやく生じる、ということだった。

Weinberg の分析がしかるべき評価を得なかったのは、用語のせいかもしれない——インターネットのハッカーたちを「エゴがない」と呼ぶなんて、つい笑ってしまうではないの。でも、かれの議論は今やかつてない説得力を持っている。

Unix の歴史を見れば、Linux から学びつつあるもの（そしてぼくが意図的にリーヌスの手法を真似ることで、実験的に小規模に確認したもの [EGCS]）は見えていたはずなんだ。コーディングは基本的に孤独な活動だけれど、真に偉大なハックはコミュニティ全体の関心と脳力を動員することで実現されるってこと。閉ざされたプロジェクトの中で、自分の脳味噌だけを使う開発者は、オープンで発展的な文脈をつくりだして、デザイン空間の探索やコードの貢献、バグつぶしなどの改善をもたらすフィードバックが何百人も（いや何千人かも）から戻ってくるようにできる開発者に負けてしまうんだ。

でも従来の Unix の世界は、このアプローチをとことんまでつきつめることができなかった。要因はいくつかある。一つはいろいろなライセンスや商売上の秘密、商業的な利害からくる法律上の制約。そしてもう一つは（いまにして思えば）インターネットがまだ発達しきってなかったことだ。

安いインターネット以前には、いくつかの地理的に集中したコミュニティではワインバーグの「エゴのない」プログラミングが奨励されていた。そこでは開発者は、有能なチェック屋や共同開発者を楽にたくさん集めることができた。ベル研、MIT AI 研、UC バークレー——こういうところは伝説的な技術革新を生み出したし、いまでも強力だ。

Linux は、意識的かつ成功裏に**全世界**を才能プールとして使おうとした最初のプロジェクトだった。Linux 形成期が、World Wide Web の誕生と同時期なのは偶然ではないと思うし、Linux が幼年期を脱したのが 1993-1994 年という、ISP 産業がテイクオフしてインターネットへの一般の関心が爆発的に高まった時期と同じなのも偶然ではないだろう。リーヌスは、拡大するインターネットが可能にした新しいルールにしたがって活動する方法を見いだした、最初の人間だったわけだ。

安いインターネットは、Linux モデルの発展にとっての必要条件ではあったけれど、でもそれだけでは十分条件ではなかったと思う。もう一つの重要な要素は、開発者が共同開発者を集めて、インターネットというメディアを最大限に活かすためのリーダーシップのスタイルと、強力のための慣行が開発されたことだろう。

でもこのリーダーシップのスタイルとはなんで、その慣行ってのはどういうものだったんだろう。これは権力関係に基づくものではあり得ない——あり得たとしても、脅しによるリーダーシップは、いまぼくたちが目にするような結果を生み出しはしない。ワインバーグは、19 世紀ロシアのアナキストであるクロポトキンの『ある革命家の回想』を引用して、この点についていい議論を展開している。

> 「農奴を所有する一家に育ったわたしは、当時の若者たちみんなと同じように、命令したり指令したり、叱りつけたり罰したりといった行動の必要性について、まったく疑うことを知らぬままに成年に達した。しかしかなりはやい時期に、わたしは大がかりな企業を経営することになり、自由な人々と交渉することになった。そしてまちがい一つが重大な結果を招くような状況で、わたしは命令と規律という原理にしたがって活動するのと、共通の理解という原理に基づいて行動するのとの差をだんだん理解するに至った。前者は軍隊のパレードでは見事に機能するが、実生活において、目標が多くの重なり合う意志の真剣な努力によってしか達成できないような状況では何の価値もない」

この「多くの重なり合う意志による真剣な努力」は、まさに Linux のようなプロジェクトには必須——そして「命令という原理」は、ぼくたちがインターネットと呼ぶアナキスト天国のボランティアたちに対しては、実質的に適用不可能だ。効果的に活動して競争するには、共同プロジェクトを仕切りたいハッカーは、クロポトキンが「理解の原理」で漠然と示唆しているモードを使い、有益なコミュニティをリクルートしてやる気を起こさせる方法を学ばなくてはならない。つまり、リーヌスの法則を学ばなくてはならないんだ。[SP]

まえにリーヌスの法則の説明として「デルファイ効果」が考えられると述べた。でも、生物学や経済学に見られる適応型システムも、アナロジーとして強力だし魅力もある。Linux の世界はいろんな意味で、自由市場や生態系のような動きを見せる。自己中心的なエージェントがそれぞれ効用を最大化しようとして、その過程で自己調整的な自律的秩序を生み出し、それはどんな中央集権計画の何倍も複雑で効率が高くなる。だからこここそが「理解の原理」を探すべき場所だ。

Linux ハッカーたちが最大化している「効用関数」は、古典経済的なものではなく、自分のエゴの満足とハッカー社会での評判という無形のものだ（かれらの動機を「愛他精神」と呼ぶ人もいるけれど、でもそれは、愛他家にとっての愛他活動はそれ自体が一種のエゴの満足だという事実を見落としている）。こういう形で機能するボランタリー文化は、実はそんなに珍しいものじゃない。

ぼくが長いこと参加してきたもう一つの例は、SF ファンダムで、ここはハッカー界とちがってボランティア活動の基本的な動機をはっきり「エゴブー」（他のファンたちの間で自分の評判を高めること）だと認識している。

リーヌスは、開発そのものはほとんど他人にやらせつつ、うまいこと自分はプロジェクトの門番におさまった。そしてプロジェクトへの関心を育てて、それが自立するようにしてきた。クロポトキンの「共通の理解という原理」を鋭く把握していることがわかるよね。このように Linux の世界を準経済学的に見てやると、その理解がどのように適用されているか見て取れるだろう。

リーヌスのやり方は、「エゴブー」の効率的な市場をつくりだす方法として見るといいかもしれない。個々のハッカーたちの利己性を、協力体制を維持しないと実現不可能なむずかしい目標に、できるだけしっかり結びつける方法だ。Fetchmail プロジェクトで、ぼくは（もっと小規模にではあるけれど）かれの手法が再現できるものだということを示した。ぼくのほうが、リーヌスよりもそれをちょっと意識的かつ体系的に行ったとはいえるかもしれない。

多くの人（特に政治的な理由で自由市場を信用しない人たち）は、自己中心的なエゴイストの文化なんか断片的で、領土争いばかりで、無駄が多く、秘密主義的で、攻撃的にちがいないと考える。でもこの予想ははっきりと反証できる。数多い例の一つをあげると、Linux 関連文書の驚くべき多様性と品質と詳細さがある。プログラマたちはドキュメント作成が**大嫌い**というのは、ほとんど神聖化された周知の事実とされている。だったら、なぜ Linux ハッカーたちはこんなにもたくさんの文書を生み出すんだろう。明らかに Linux のエゴブー自由市場は、商業ソフト屋さんのものすごい予算をもらった文書作成業者たちよりも、気高さに満ちた他者をいたわる行動を生み出すうえでうまく機能するわけだ。

Fetchmail と Linux カーネルプロジェクトがどちらも示しているのは、ほかの多くのハッカーたちのエゴにきちんとごほうびをあげれば、強力な開発者/コーディネータはインターネットを使って、共同開発者がたくさんいるメリットを享受しつつ、プロジェクトが混乱しきった修羅場に陥って崩壊するのは避けられる、ということだ。というわけで、以下はブルックスの法則に対するぼくの反対提案：

19. **開発コーディネーターが、最低でもインターネットくらい使えるメディアを持っていて、圧力なしに先導するやりかたを知っている場合には、頭数は一つよりは多いほうが絶対にいい。**

フリーソフト（オープンソース・ソフト）の未来は、ますますリーヌスのやりかたを身につけた人たちのものになっていくと思う。つまり、伽藍を後にしてバザール方式を受け入れる人たちのものだ。これは別に、個人のビジョンと才能がもはやどうでもいいということではない。むしろ、フリーソフト/オープンソースの最先端は、個人のビジョンと才能を出発点としつつも、それをボラ

ンタリーな利害/興味コミュニティの構築によって増幅する人々のものだと思う。

そしてこれは、単に「フリー」ソフト（オープンソース・ソフト）だけの未来像ではないのかも知れない。問題解決にあたって、Linux コミュニティが動員できるほどの才能プールに太刀打ちできる商業デベロッパは存在しない。Fetchmail に貢献してくれた 200 人以上を雇える財力を持つようなデベロッパですら、ごくわずかしかいない！

もしかすると、最終的にフリーソフト/オープンソース文化が勝利するのは、協力が道徳的に正しいとかソフト「隠匿」が道徳的にまちがってるとかいう理由のためではなく（ちなみに後者については、リーヌスもぼくもそうは思わない)、単に商業ソフトの世界が、ある問題に有能な人々の時間を幾桁も多くそそぎ込めるフリーソフト/ オープンソース界と、進化上の軍事競争で張り合えなくなるからかもしれない。

11　マネジメントとマジノ線について

もともとの「伽藍とバザール」はいまのビジョンで終わっていた——ネットワーク化されたシヤワセな集団プログラマー／アナキストたちが、伝統的な閉鎖ソフトの階級社会を競争でうち負かして圧倒するという。

かなりの懐疑論者は、納得しなかった。そしてかれらの挙げる質問は、まともに相手をするだけの価値がある。バザール議論へのほとんどの反論は、バザール支持者たちは従来のマネジメントが持っている生産性倍増効果を過小評価している、という話にいきつく。

伝統的な考え方のソフト開発マネージャは、オープンソース世界ではいともあっさりプロジェクトグループが生まれ、姿を変えては消え去っていくので、オープンソース・コミュニティーがどのクローズドソースの開発者集団と比べても、数の上で圧倒的な優位性を持っているというメリットの大部分は消えてしまう、と反論することが多い。かれらの主張では、ソフト開発で重要なのは長期にわたり開発努力が継続することで、顧客が大事な製品に投資を続けてくれると期待できるとうにすることなのだ、何人が鍋に骨を投げ込んでそのまま煮立てておこうと、そんなのは関係ない、ということになる。

この議論には、確かに一理ある。実はぼくも、『魔法のおなべ』でソフト生産の鍵となるのは期待将来サービス価値だという考えを展開している。

でもこの議論にはまた、大きな問題が隠されている。それが暗黙に仮定しているのは、オープンソース開発はそういう継続的な努力を提供できないということだ。ところが実は、従来型のマネジメントが不可欠だと考えるような、インセンティブ構造や組織的なコントロールなんかまったくなしに、きちんと方向性を保って有能な管理者コミュニティをずいぶん長期にわたって維持してきたオープンソースプロジェクトはたくさんある。GNU Emacs エディタの開発は、その極端で示唆的

な例だろう。15 年にわたり何百もの貢献者たちの努力を吸収して、統合されたアーキテクチャの方向性を維持してきている。開発者の入れ替わりは激しくて、しかもその間ずっと活動を続けてきたのは、たった一人（開発者）だけだ。クローズドソースのエディタのどれ一つとして、これほどの長寿記録にかなうものはない。

ここから出てくるのは、伽藍 VS バザール方式の議論とは独立した、従来型のソフト開発マネジメントのメリットに対する疑問だ。もし GNU Emacs が一貫したアーキテクチャのビジョンを 15 年も維持できたり、Linux みたいな OS が同じように 8 年も、ハードやプラットホームの技術が変わり続ける中で維持できたのなら——そしてもし、ほかにもきちんとしたアーキテクチャを持ったオープンソースのプロジェクトが、5 年以上も続いている例があるなら——だったらわれわれとしては、そもそも従来型のマネジメント方式の開発というのは、あれだけすさまじいオーバーヘッドをかけて、いったい何を買っているんだろう、というのを当然聞いてみたくなるわけだ。

それはまちがいなく、締め切りを信頼できる形で守るということではないし、予算内にきちんとおさめるということでもないし、仕様書をすべて反映させるということでもない。こういう目標の一つでも達成できたら、それは珍しくきちんと「マネジメント」されたプロジェクトだと言っていい。また、プロジェクトの寿命の中で、技術的・経済的な環境変化にもすばやく適応できているとは思えない。この点では、オープンソース・コミュニティのほうが**はるかに**高い能力を見せてきた（これはたとえば、インターネットの 30 年にわたり歴史と、独占ネットワーク技術の短い半減期とを比べてみればすぐに確認できる——あるいはマイクロソフト・ウィンドウズの 16 ビットから 32 ビットへの移行がえらく高コストだったのにくらべて、同じ頃に同じことをやった Linux ではほとんど何の苦労もなかったことを考えてみるといい。しかも Linux はインテル専用ではなくて、64 ビットの Alpha チップも含む一ダースものハードウェア・プラットホームでそれを実現したんだぜ）。

伝統的な開発様式で買えるもんだと多くの人が考えているものとしては、プロジェクトがおかしくなったときに、法的に縛って責任をおわせ、可能性としては損害賠償金も得る相手ができる、ということだ。でもこんなのは幻想でしかない。ほとんどのソフトライセンスは、このソフトが商品として売り物になることすら保証しないような免責条項が書かれているし、まして性能のことなんかまるっきり保証しない——そしてソフトが期待性能に達しない場合に、損害賠償を勝ち取れたケースがあるか？まったくないといっていいくらい、ほとんどない。そしてそれがそんなに珍しいことでなかったとしても、訴える相手がいるから安心なんていうのは、そもそもがピントはずれだ。きみは訴訟がしたかったのか？ちゃんと動くソフトがほしかったんだろうに。

じゃあ、マネジメントのオーバーヘッドをあんなにかけて、いったい何が手に入るんだろう。

これを理解するには、ソフトウェアの開発マネージャたちが何をしているつもりなのかを理解しなくてはならない。ぼくの知り合いで、この仕事がとても上手らしき女性に言わせると、ソフト

ウェアプロジェクトのマネジメントというのは五つの機能がある。

1. **目標を決めて**、みんなを同じ方向に向かせておく。
2. しっかり**見張って**、大事な細部が見落とされないよう確かめる。
3. みんなの**やる気を出させて**、つまらないけれど必要などた作業をやらせる。
4. メンバーの配置を**組織化して**、最大の生産性をあげるようにする。
5. プロジェクトの維持に必要な**リソースをひっぱってくる**。

一見どれも立派な目標だ。一つ残らず。でもオープンソースモデルと、それをとりまく環境の中では、どれも不思議なくらいどうでもよく思えてくる。ケツのほうから撃退していこうか。

この友人の報告では、**リソース調達**のかなりの部分は、実は手持ちの防衛なんだという。自分のスタッフとマシンとオフィスを確保したら、同じリソースを求めて競合している同僚のマネージャたちや、限られたプールの最有効利用を目指して配置を考えている上司たちから、それを守らなくてはならない。

でもオープンソース開発者たちはボランティアだし、自分のかかわるプロジェクトについて、興味と能力で自薦により貢献することなったわけだ（そしてこれは、オープンソースのハッキングで給料をもらうようになったときにもおおむねあてはまる）。ボランティアの精神は、リソース調達の「攻撃」側の面倒をみてくれる。みんな自主的に自分のリソースを提供してくれるんだ。そして、伝統的な意味でマネージャが「防御」する必要性は、ほとんど、いやまったくない。

どのみち、安い PC や高速インターネット接続の世界では、限られている唯一のリソースというのは、才能ある人々の関心だけだ。オープンソースプロジェクトは、つぶれるときにも、別にマシンやリンクやオフィスが足りないからつぶれるんじゃない。開発者自身が関心を失ったからという、それだけだ。

そういうことなら、オープンソースのハッカーたちが、自薦によって最大の生産性をあげるべく**自ら組織化する**というのは、二重の意味でだいじになる——そしてハッカー界の社会リネンによって、有能な者だけが容赦なく選ばれる。ぼくの友だちは、オープンソース界と巨大閉鎖プロジェクトとの両方を知っていて、オープンソースが成功した理由の一部は、その文化がプログラミング人口のトップ 5% しか受け入れないからだ、と信じている。彼女は、残り 95% の動員を組織するのに時間を費やしている。そして、最高のプログラマと、単に有能なだけのプログラマとの 100 倍もの生産性のちがいというのを、直接見せつけられているのだ。

この生産性の差は、居心地の悪い質問をずっと投げかけ続けてきた。ということはだよ、ソフトの個別プロジェクトや、ソフト産業全体として見ても、使えない下半分を切り捨てたほうがずっとよくなるんじゃないだろうか。もし伝統的なソフト管理の唯一の機能が、一番使えないやつらを、せめて損害は出さずにトントンに持っていくくらいだとしたら、そんな仕事は何の価値もないん

じゃないかということを、有能なマネージャたちは昔から理解していた。

オープンソース・コミュニティの成功は、この質問をもっともっと尖鋭化する。インターネットで自薦のボランティアをつのったほうが、ほかのことをしていたい人たちをビルいっぱい集めて管理するよりも安くて効率がいいことが多いという、動かしがたい証拠をつきつけているからだ。

というところで、話は都合よく**動機づけ**のほうにやってきた。友人の論点と同じことを別の言い方で言っているのをよく聞く。伝統的な開発マネジメントは、そのままではいい仕事をしてくれない、やる気のないプログラマたちを補うためのものなんだ、と。

この答はまた、オープンソースは「セクシー」で技術的に魅力ある仕事でしかあてにならない、という議論とセットになっていることが多い。それ以外のことは、放っておかれるままだ（あるいはいい加減にしか処理されない）。それをやるには、札束でひっぱたかれて、区画に閉じこめられた日雇いプログラマが、頭上でマネージャのふりまわす鞭の響きをききつつ必死で書くしかない、というわけだ。ぼくは、こういう主張がまゆつばだと思う理由について『ノウアスフィアの開墾』で述べた。でもこの文の趣旨からして、これを本当として受け入れたらどういうことになるかを指摘するほうが、もっとおもしろいだろう。

もし従来型のクローズドソースでマネジメント過大のソフト開発スタイルが、退屈からくる問題でつくったマジノ線によってしか弁護できないのならば、その議論が成立するのは、それぞれのアプリケーション領域で、だれもその問題を本気で面白いとは思わないし、さらに他にだれもその問題の抜け道を見つけない場合に限られる。「退屈」なソフトにオープンソースの競合がでてきた瞬間に、顧客たちは、その問題自体が面白いから解決してやろうという人物が、それに取り組んでいるんだな、というのがわかるわけだ——そしておもしろさというのは、ソフトに限らずあらゆるクリエイティブな仕事に言えることだけれど、ただのお金なんかよりもずっとずっと優れたニンジンなんだ。

伝統的なマネジメント構造を、みんなの尻を叩くためだけに持っておくというのは、戦術的には優れていても、戦略的にはダメだ。短期的には成功しても、長期的にはまちがいなく負ける。

いまのところ、伝統的な開発マネジメントは２つの点（リソースの調達と、組織）で、オープンソースに対して勝ち目がなさそうに見える。そして三番目の点（動機づけ）でも、先は短そうだ。さらにあわれな包囲された伝統的マネージャは、**監督面でも**点をあげられない。オープンソースコミュニティを支持する最強の議論が、分散化された同業者レビューシステムが、細部の見落としがないようにするための従来型のあらゆる方式など足下にも及ばないほど優秀だ、ということなんだから。

じゃあ、伝統的なソフトプロジェクト管理のオーバーヘッドを正当化するのに、**目標設定**というのくらいは救ってやれるかな？かもね。でも、そのためには、マネジメント委員会だの企業のロードマップだのが、オープンソース界で似たような役割を果たすプロジェクトごとの「優しき独裁

者」や部族の長老に比べて、価値あるみんなに共有される目標を定義するのが上手だ、と信ずべきまともな理由が必要になる。

こいつを正面きって主張するのは、なかなかにむずかしいことだ。そしてそれがむずかしいのは、オープンソース側がどうのというわけではない（Emacs の長寿ぶりとか、リーヌス・トーヴァルズが「世界征服」を語って開発者の群を蜂起させられるとか）。むしろ、ソフトプロジェクトの目標設定にあたって、伝統的な仕組みがそのタコぶりを遺憾なく証明してきてしまったという点が問題なんだ。

ソフト工学の、口伝理論でいちばん有名なものとして、伝統的なソフトプロジェクトの 60% から 75% は、結局完成せずに終わるか、あるいはその予定顧客に拒絶される、というものがある。もしこの数字が多少なりとも真実に近いなら（そして多少なりとも経験あるマネージャで、これを否定する人にはお目にかかったことがない）、たぶん半分以上のプロジェクトは、現実的に達成不可能か、あるいは単純にひたすらまちがっている目標を目指してすすんでいるということになる。

これは、他のどんな問題にも増して、いまのソフト工学界で「マネジメント委員会」ということばがそれを聞いた者の背中に寒気を走らせる理由なのだ。それを聞いた人が、たとえマネージャであったとしても（いや、マネージャであればこそ、かな）。このパターンについてグチっていたのがプログラマだけだった時代は、とうの昔に過ぎ去っている。「ディルバート」のマンガが、いまでは**重役の**デスクにだって置かれているのだ。

すると、伝統的なソフト開発マネージャへのわれわれの答は簡単だ——もしオープンソース・コミュニティが伝統的なマネジメントの価値を過小評価しているというんなら、**なぜあなたたちのそんなに多くが、自分自身のプロセスに対して恐怖や恐れや侮蔑を表明しているのでしょうか**？

またもや、オープンソース・コミュニティの存在がこの質問をかなり尖鋭にしてしまう——ぼくたちは、楽しんでやっているんだもの。ぼくたちの創造的な遊びは、技術面でも、市場シェア面でも、精神的なシェアでも、すさまじい勢いで成功を重ねてきている。ぼくたちは、もっといいソフトがつくれることを示しただけじゃない。**よろこびが資産**であることを証明してもいるんだ。

この論文の最初の論文から二年半たって、ぼくが最後に提供できるいちばんラジカルなアイデアというのは、オープンソースが圧勝した世界のビジョンではない。だってそんなものは、いまではスーツ姿のしらふ人間たちにだって、ずいぶん納得のいくものになっているようじゃないか。

むしろぼくは、ソフトウェア（そしてあらゆる創造的またはプロフェッショナルな仕事）についての、もっと広い教訓をここで提示してみたい。人間は仕事をするとき、それが最適な挑戦ゾーンになっていると、いちばん嬉しい。簡単すぎて退屈でもいけないし。達成不可能なほどむずかしくてもダメだ。シヤワセなプログラマは、使いこなされていないこともなく、どうしようもない目標や、ストレスだらけのプロセスの摩擦でげんなりしていない。**楽しみが能率をあげる**。

自分の仕事のプロセスにびくびくゲロゲロ状態で関わり合う（それがつきはなした皮肉なやりか

ただったとしても）というのは、それ自体が、そのプロセスの失敗を告げるものととらえるべきだ。楽しさ、ユーモア、遊び心は、まさに財産だ。ぼくがさっき、「シヤワセな集団」という表現を使ったのは、別に「シ」の頭韻のためだけじゃないし、Linux のマスコットがぬくぬくした幼形成熟（ネオテニー）っぽいペンギンなのもただの冗談じゃあない。

オープンソースの成功のいちばんだいじな影響の一つというのは、いちばん頭のいい仕事の仕方は遊ぶことだということを教えてくれることかもしれない。

12 謝辞

この論文は、デバッグを手伝ってくれた多数の人々との会話によって改善されてきた。とくに感謝したいのが Jeff Dutky <dutky@wam.umd.edu>。かれは「デバッグは並列処理可能である」という言い方を提案して、そこから生まれる分析の形成を助けてくれた。Nancy Lebovitz <nancyl@universe.digex.net>は、クロポトキンを引用してワインバーグをまねしたらどうかと示唆を与えてくれた。ほかに有益なコメントをくれた人としては、General Technics リストの Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> と Marty Franz <marty@net-link.net>。Glen Vandenburg <glv@vanderburg.org> は貢献者人口の自己選別性が重要だということを指摘してくれたし、多くの開発というのが「見過ごしというバグ」を修正するものだという有益なアイデアをくれた。Daniel Upper <upper@peak.org> は、これについて自然界のアナロジーを提供してくれた。PLUG（フィラデルフィア Linux ユーザグループ）のメンバーたちにも感謝する。かれらはこの論文の初の公開版について、最初のテスト読者になってくれた。Paula Matuszek <matusp00@mh.us.sbphrd.com> はソフトウェア管理の実務についてぼくを啓蒙してくれた。Phil Hudson <phil.hudson@iname.com> は、ハッカー文化の社会組織構成はそのソフトの構成を反映するものであり、その逆も真だということを思い出させてくれた。最後に、リーヌス・トーヴァルズのコメントは有益だったし、かれがはやいうちから賛同してくれたので、ぼくとしてもやりやすかった。

13 もっと考えたい人のための文献リスト

Frederick P. Brooks の古典 *The Mythical Man-Month* [*9]からはあちこち引用させてもらった。というのも、かれの洞察はいろいろな意味で、まだまだそのまま通用するものだからだ。Addison-Wesley から出ている刊行 25 周年記念版 (ISBN 0-201-83595-9) を是非ともお奨めする。

[*9] フレデリック・P・ブルックス『人月の神話――狼人間を撃つ銀の弾はない』アジソン・ウェスレイパブリッシャーズ・ジャパン、1996 年

これにはかれの 1986 年論文 *No Silver Bullet*[*10] も収められている。

この新版の巻末には、非常に有益なブルックスの 20 年後の回想記がついていて、このなかでブルックスは文章において結果的にまちがっていた部分について、すなおに認めている。ぼくはこの論文をほとんど書き上げたときにこの回想記を読んだのだけれど、ブルックスがバザール式のやり方の例としてマイクロソフトを挙げていたと知ったときにはたまげたね！（ただし実際には、このかれの例示はまちがっていたことがわかった。1998 年に、ぼくらは『ハロウィーン文書[*11]』によって、マイクロソフト内部の開発者コミュニティはひどい戦国状態にあることを知った。バザールを支えるために必要な、広いソースコードへのアクセスは、実はぜんぜん可能ではないんだ）。

Gerald M. Weinberg の *The Psychology of Computer Programming* (New York, Van Nostrand Reinhold 1971)[*12] は、「エゴのないプログラミング」という考え方を導入していて、これは名前のつけかたがまずかったと思う。「命令主義」の不毛さについて認識したのは、かれが最初でもなんでもないけれど、でもそれを特にソフト開発に結びつけて論じたのはたぶんかれが最初だと思う。

Richard P. Gabriel は Linux 以前の時代の Unix 文化を考察し、1989 年の論文 *Lisp: Good News, Bad News, and How To Win Big* のなかで、初期のバザール状モデルの優位性を論じている。古びたところもあるけれど、この文章はいまでも Lisp ファン（ぼくを含め）の間では当然ながら珍重されている。ある人がぼくに、この文のなかの「劣るほうが優秀」という章は、まるで Linux を予見しているかのように読めることを指摘してくれた。この論文は World Wide Web で入手可能[*13]。

De Marco と Lister の *Peopleware: Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6) [*14]は知られざる名著で、フレッド・ブルックスが回想記の中で触れているのを見たときは嬉しかった。著者たちの議論のなかで、直接 Linux やフリーソフト（オープンソース）界に適用できるものはあまりないけれど、創造的な作業に必要な条件に関する著者たちの洞察は正確で、バザールモデルの長所をもっと商業的な場に導入したいと試みる人には一読の価値がある。

最後に、ぼくはこの論文を寸前まで「伽藍とアゴラ」と呼ぶところだったのを白状しておこう。アゴラというのは、ギリシャ語で自由市場や公共集会場所をさすことばだ。Mark Miller と Eric

[*10] 前掲書第 16 章所収。邦題「銀の弾などない」

[*11] 原文：http://www.opensource.org/halloween/
邦訳：http://cruel.org/freeware/halloween.html

[*12] G. M. ワインバーグ『プログラミングの心理学　またはハイテクノロジーの人間学』木村泉他訳、技術評論社、1994 年

[*13] http://www.naggum.no/worse-is-better.html

[*14] デ・マルコ＆リスター『ピープルウェア』日立 SK 訳、日経 BP 社、1989

Drexler の先駆的な論文「アゴラ的システム」は、市場状のコンピュータ生態学にあらわれつつある性質を記述していて、5 年後に Linux がぼくをフリーソフト（オープンソース・ソフト）での類似現象に直面させたときも、これを読んでいたおかげで明確にものを考える準備ができていた。この論文は Web 上で入手可能[*15] 。

14　エピローグ：Netscape もバザール方式を受け入れる

自分が歴史の変化に手を貸したと気がつくのは、なんとも奇妙な感じだ……

1998 年 1 月 22 日、ぼくがこの論文を初めて発表してからおよそ 7 ヶ月後、Netscape Communications, Inc. が Netscape Communicator のソースを無料でばらまく計画を発表した[*16]。

この発表の前日でさえ、こんなことが起こるとはつゆほども知らなかった。

Netscape の専務副社長兼技術担当重役の Eric Hahn が、発表のすぐ後にぼくにメールをくれた。こんな文面だ。「Netscape 全社員を代表して、そもそもこのポイント把握を助けてくれたことに感謝します。あなたの考え方と論文が、われわれの決断にあたって根本的なひらめきを与えてくれました」

翌週、ぼくは Netscape 社の招きでシリコンバレーに飛び、かれらの重役や技術陣との丸一日にわたる戦略会議（1998 年 2 月 4 日）に出席した。ぼくたちは Netscape のソース公開戦略とライセンスをつくり、その他、いずれフリーソフト（オープンソース）コミュニティに重大で前向きな影響をもたらすはずの計画をつくりあげた。この執筆時点では、これ以上細かい話をするのは時期尚早だけれど、でも数週間以内に追って詳細が発表されるはずだ。

この数日後に、ぼくは次のように書いた。

> Netscape は大規模な現実世界におけるバザールモデルのテスト機会を提供してくれようとしている。フリーソフト/オープンソース文化は、いま一つの危険に直面していることになる。もし Netscape のやりくちがうまくいかなければ、フリーソフト/オープンソースの考え方自体がダメなせいだと思われてしまい、商業ソフトの世界はまた 10 年ほどは手を出そうとしなくなるかもしれない。
>
> 一方、これはまたとてつもないチャンスでもある。この動きに対するウォール街などでの初期の反応は、慎重ながらも肯定的だった。ぼくたちは自らの力を証明する機会を与えられているのだ。この動きを通じて Netscape が再び圧倒的な市場シェアを取り戻せば、それをきっかけにもうとっくに起こっていてしかるべきだったコンピュータ産業の革命が動き出す

[*15] http://www.agorics.com/agorpapers.html
[*16] http://www.netscape.com/newsref/pr/newsrelease558.html

かも知れない。

この先一年は、非常に示唆的でおもしろい時期になるだろう。

そして確かに、実におもしろい一年だった。1999 年半ば現在では、後に「Mozilla」と名づけられたものの開発はそこそこの成功だとはいえるだろう。Netscape のもとの目標は達成した。それは、マイクロソフトにブラウザ市場の独占封じ込めを許さないということだった。さらに劇的な成功もあった（特に次世代レンダリング・エンジンの Gecko のリリース）。

しかしながら、Mozilla の創始者たちが願ったような、Netscape 外部からのものすごい開発協力は未だに得られていない。ここでの問題はどうも、Mozilla が長いこと、バザール方式の基本ルールを破っていたことにあるようだ。かれらは、貢献者候補たちがすぐに走らせて動いているのを見られるようなものを出荷しなかった（リリースから 1 年以上たちまで、Mozilla をビルドするには、独占 Motif ライブラリのライセンスが必要だった）。

いちばんのマイナス点（外部世界から見た場合）は、Mozilla グループが未だに商品クラスの高品質ブラウザを出荷していないということだろう——そしてプロジェクトの代表者の一人は、マネジメントのまずさと機会喪失について愚痴りつつ辞職することで、いささかの騒動を巻き起こした。かれは正しくもこうコメントしている。「オープンソースは、魔法の砂なんかじゃない[*17]」と。

そりゃそうだ。Mozilla の長期的な見通しは、いま（1999 年 8 月現在）、Jamie Zawinski の辞職願の頃よりもかなり改善されてはいる——でも、オープンソースにするだけで、見当ちがいの目標やスパゲッティ・コードや、その他ソフト工学の欠陥に苦しむプロジェクトが救われるわけではない、というかれの指摘は正しい。Mozilla は、オープンソースがいかにして成功するかという例と、いかにして失敗するかという例を同時に示してくれたわけだ。

しかし一方では、オープンソースの考え方は、それ以外の場所で成功をおさめ、支持者を見つけてきた。1998 年と 1999 年には、オープンソース開発モデルに対する関心が爆発的に高まった。これは Linux OS の相変わらずの成功に牽引されたものでもあり、それを牽引するものでもある。Mozilla が立ち上げた動きは、ますます加速して前進しつつあるのだ。

15 脚注

[JB] **Programing Pearls** で、有名なコンピュータ科学の警句屋ジョン・ベントレーは、ブルックスの観察についてこうコメントしている。「もし一つ捨てることを予定して置いたら、二つ捨てる結果になるだろう」。かれはほとんど確実にただしい。ブルックスの見解と、ベントレーのコメントというのは、単に最初の試みはまちがえやすいから覚悟しろ、というこ

[*17] 『辞職と回顧：mozilla.org 顛末記』http://www.bekkoame.ne.jp/ kmakoto/opencom/Jamie-san.html

とではない。めちゃくちゃになったのを救うよりは、正しいアイデアで一からやり直すほうが、有効な場合が多い、ということを言いたいのだ。

[QR] インターネットに先立つ、成功したオープンソースのバザール形式開発で、しかも Unix やインターネットの伝統とは関係ないものも存在している。1990-92 年の info-Zip 圧縮ユーティリティ[*18]の開発（主に DOS マシン用）はその一例だ。もう一つあるのが、RBBS BBS ソフトだ（これまた DOS 用）。これは 1983 年に始まって、なかなか強力なコミュニティが形成され、インターネットの電子メールやファイル共有のほうがローカルの BBS よりもずっと技術的なメリットが高くなった今（1999 年半ば）にいたるまで、定期的なリリースを繰り返している。PKZIP コミュニティはある程度までインターネットの電子メールに頼っていたけれど、RBBS 開発者の文化は、RBBS 自身を使って相当なオンラインコミュニティを擁し、完全に TCP/IP インフラとは独立していた。

[JH] John Hasler は、ある作業が重複してやられたところで、オープンソースの開発にとっては差し引きであまり足をひっぱる結果にはならないと示唆してくれた。かれが提案したものを「ハスラーの法則」と呼ぼう。重複作業のコストは、そのチームのサイズの二乗より少ない——つまり、そういう重複を避けるために必要な、計画やマネジメントのオーバーヘッドに比べて増え方が遅いのだ。

この主張は、実はブルックスの法則に反するものではない。複雑なオーバーヘッド総額と、バグへの弱さがチームサイズの二乗に比例するのは事実かもしれないけれど、でも重複作業からくるコストは、もっとゆっくりスケールする特殊な例でしかない。これにもっともらしい理由をつけるのは、そんなにむずかしくない。まずは、ほとんどのバグの原因となっている、計画外のよからぬ相互作用を防ぐのに比べれば、開発者のコード同士で機能の仕分けについて話あうのはずっと簡単だという、まちがいのない事実からも説明できる。

リーヌスの法則とハスラーの法則を組み合わせると、ソフトプロジェクトでサイズの段階が 3 段階くらいあることがわかる。小規模なプロジェクトでは（つまり開発者が一人からせいぜい三人くらいだろう）、リーダーとなるプログラマを選ぶ以外には、ややこしいマネジメント構造は必要ない。そしてそれを超えた中間くらいのところで、伝統的なマネジメントのコストがそこそこ低くて、作業の重複を避けたり、バグを追跡したり、細かい見落としがないかを調べるためのマネジメントがメリットをもたらすサイズがあるだろう。

でもそれを超えるとリーヌスの法則とハスラーの法則が組み合わさって、伝統的なマネジメントのコストと問題が、作業重複からの期待トラブルよりも急速に増えるサイズというのが出てくるだろう。このコストのなかでも無視できないのが、「目玉たくさん効果」を導入

[*18] http://www.cdrom.com/pub/infozip/

できないという構造的な問題だ。目玉が多いほうが（これまで見てきたように）伝統的なマネジメントよりも、バグの見落としや細部の見落としに対してはずっと効果的なのだ。だから大規模プロジェクトのケースでは、この 2 法則の組み合わせのおかげで伝統的なマネジメントのメリットは、ゼロにまで下がってしまう。

[IN] バザール方式でゼロからプロジェクトを立ち上げられるかという問題は、バザール方式が真に革新的なものをサポートできるのか、という問題と関係している。ある人に言わせると、強いリーダーシップのないバザールは、エンジニアリングの最先端にすでに存在しているアイデアをまねしたり、改良したりできるだけで、その最先端を先に進めることはできないそうだ。この議論を提出したいちばん悪名高い例が、ハロウィーン文書[*19]だろう。オープンソース現象について書かれた、マイクロソフトの恥ずかしい社内メモだ。この著者たちは、Linux が Unix に似た OS を開発するのを「テールライトを追いかける」と例えてくれて、「(そのプロジェクトがひとたび技術の最先端の「飽和点」に達してしまえば)、新たなフロンティアに向けてみんなを押し進めるために必要となるマネジメントのレベルはとてつもないものとなる」と論じている。

この議論には、深刻な事実関係のまちがいが含まれている。その一つは、ハロウィーン文書の著者たち自らが次のように洞察しているところではっきり表明されている：

> 具体的にはこれは、新しい研究上のアイデアはまず Linux 上で実装されて入手可能となり、その後でほかのプラットホームで提供されたり組み込まれたりするようになる、ということだ。

ここで Linux を「オープンソース」と読み替えれば、これがまるで目新しい現象でないことはわかるだろう。歴史的に、オープンソース・コミュニティが Emacs や World Wide Web やインターネットを発明したのは、テールライトを追っかけたり、とてつもないレベルのマネジメントがあったりしたためではない——そしていまでも、オープンソースではとても多くの独創的な仕事が続いていて、選ぶのに目移りするほどだ。あえて一つ選ぶとすると、GNOME プロジェクトは GUI とオブジェクト技術の最先端を押し広げていて、Linux とはかけ離れたコンピュータ業界紙からも大いに注目されている。その他の例も目白押しで、これはいつでもいいから Freshmeat [*20]を訪ねてみればすぐに証明される。

でも、伽藍方式が（あるいはバザール方式でも、その他どんなマネジメント方式でもいい)、なにやら技術革新を信頼できるかたちで生じさせられるという、暗黙の仮定にはもっと根本的なまちがいがある。だって、そんなのナンセンスだからだ。群衆は、突破口となる

[*19] 原文　http://www.opensource.org/halloween/
翻訳　http://cruel.org/freeware/halloween.html

[*20] `http://freshmeat.net/`

ような洞察なんか持てない――バザール方式のアナキストたちによるボランティア集団でさえ、まともなオリジナリティは発揮できないし、ましてやなにか現状に生存が係っているような人々からなる、企業委員会なんかからそんなものは絶対に出てきやしない。**洞察は個人からくる**。それをとりまく社会機構としてせいぜい期待できるのは、その突破口となる思いつきに対して敏感に反応することくらいだ――それをつぶすのではなく、ちゃんと育てて報酬を与え、きちんとテストしてやることだ。

これをロマンチックな見方だと決めつける人もいるだろう。孤独な発明家というステロタイプに逆戻りしている、と。ちがうね。ぼくは別に、いったん生まれた突破口となる洞察をグループが**育てられない**なんて言ってるわけではない。それどころか、ピアレビューのプロセスから学ぶのは、こうした開発グループこそが高品質の結果を生み出すために不可欠だということだ。むしろぼくが指摘しているのは、そういうグループ開発もすべて出発点は――つまり、それに火をつけるのは必ず――だれか一人が思いついた、いいアイデア一つなんだ、ということだ。伽藍やバザールなんかの社会的な機構は、その火花をつかまえて洗練させることができるけれど、でもその機構が命令して着想を生み出したりはできないんだ。

だから、技術革新の根本問題（これはソフトウェアに限らずあらゆるところで）というのは、そのアイデアをどうやってつぶさずにおくか、ということだ――でも、もっと根本的には、**そもそも洞察を持てるような人たちをたくさん育てるにはどうしたらいいか**、ということだ。

伽藍方式の開発がこいつを実現できて、参入障壁が低い、プロセスの流動的なバザールではこれができないと仮定するのはバカげている。もしたった一人のたった一つのアイデアでいいなら、一人の人間がそのいいアイデアで、何百、何千という人々の協力をすぐに集められる社会方式のほうが、クビになる心配なしにそのアイデアに基づく作業ができるようになるために、階級機構に対して政治的な売り込みをしなくてはならないようなシステムに比べて、革新は早いに決まっている。

そして実際に、伽藍方式を使った組織によるソフトの技術革新の歴史を見てみると、あまり数がないことがすぐにわかる。巨大企業は新しいアイデアの源として大学の研究に頼っている（だからこそハロウィーン文書の著者たちは、Linux がその研究成果をずっとはやく取り入れられるということに危機意識を見せている）。あるいは、革新者の頭脳を中心に生まれた小企業を買収するだろう。いずれの場合にも、伽藍文化には技術革新は根付いていない。それどころか、そうやって輸入された技術革新の多くは、ハロウィーン文書の著者たちがあんなに持ち上げる「とてつもないレベルのマネジメント」によって、静かに窒息させられてしまう結果となる。

これはでも、否定的なポイントだ。読者のみんなは、もっと肯定的な論点のほうが役にた

つだろう。試しに、以下のような実験をしてみたらどうだろう。

1. 一貫性を持って適用できると思うような、オリジナリティをはかる尺度を選ぶこと。きみの定義が「オリジナリティなんて見りゃわかる」というものであっても、このテストでは問題にはならない。
2. Linux と競合しているクローズドソースの OS をどれでもいいから選んで、その OS 上で進行中の開発作業を記述した最高の情報源を選ぶこと。
3. その情報源と Freshmeat を一ヶ月眺めること。毎日、「オリジナル」な仕事だと思われるリリースの発表の数を数えること。「オリジナル」の定義をその別の OS にも適用して、その数を数える。
4. 30 日たったら、両方の数字をそれぞれ合計。

これを書いた日だと、Freshmeat はリリースのアナウンス 22 件があって、そのうち 3 つは何らかの形で最先端をさらに先へ進めるようなものだった。この日の Freshmeat は低調だったけれど、でもクローズドソースのどんなプロジェクトでも、ものになりそうな技術革新が一月 3 つもあったら驚嘆しちゃうね。

[EGCS] いまや、いくつかの意味で fetchmail よりもバザール方式の実例として好都合なプロジェクトの歴史が手に入った。それが EGCS [*21]、gcc の高速版である Experimental GNU Compiler System だ。

このプロジェクトは 1997 年半ばに、この「伽藍とバザール」初期公開版に登場したアイデアを意識的に適用してみようという試みとしてはじまった。プロジェクトの創始者たちは、GCC（GNU C コンパイラ）の開発が停滞していると感じていた。そしてそれ以降 20 ヶ月にわたり、GCC と EGCS は並行したプロジェクトとして続いていた——どちらも同じインターネット開発人口から人材を集め、どちらも同じ GCC のソースベースから出発してるし、どちらもだいたい同じ Unix ツールセットと開発環境から始めている。両プロジェクトの唯一のちがいは、EGCS が意識的に、ぼくがこれまでに記述してきたバザール戦術を用い、それに対して GCC のほうは、閉じた開発者グループとめったにないリリースとでもっと伽藍的な開発方式を続けたということだった。

これは、対照実験にいちばん近いものといっていい。そして結果は劇的だった。数ヶ月のうちに、EGCS のバージョンは、機能面ではるかに GCC を引き離した。最適化も向上していて、FORTRAN と C++ のサポートも優れていた。多くの人は、EGCS の開発途上スナップショットのほうが、GCC の最新安定版より信頼性が高いと判断しており、主要 Linux ディストリビューションも EGCS に移行しはじめた。

[*21] http://egcs.cygnus.com/

1999 年 4 月には、フリーソフト財団（GCC の公式スポンサー）はもとの GCC 開発チームを解散して、プロジェクトのコントロールを公式に EGCS ステアリング・グループに譲りわたした。

[SP] もちろん、クロポトキンの批判とリーヌスの法則は、社会機構のサイバネティクスについてもっと大きな問題を提起している。ソフト工学についての別の口伝理論が、その一つを示している。これはコンウェイの法則と言われる——ふつうの言われ方では、「もしコンパイラをつくるのに 4 つのグループが作業していたら、できあがるのは 4 パスコンパイラになる」となる。もとの表現はもっと一般的だった。「システムを設計する組織は、その組織のコミュニケーション構造の複製であるような設計を生み出すように縛られる」というものだ。これをもっと平たくして「手段が目的を決定する」と言ってもいいだろう。あるいは「プロセスこそが成果物となる」とでも。

したがって、オープンソース・コミュニティでは、組織形態と機能がいろんなレベルで一致していることは頭にいれておいていいだろう。ネットワークがすべてで、いたるところにある。インターネットだけじゃない。みんな分散した、ゆるい結びつきの、ピア・ツー・ピアのネットワークで作業を進めて、それがいくつもの冗長性を生んで、退行のしかたもとても緩やかだ。いずれのネットワークでも、各ノードはほかのノードがそれと協力したがる度合いに応じてのみ重要となる。

オープンソース・コミュニティのすさまじい生産性には、このピア・ツー・ピアの部分が本当にだいじだ。クロポトキンが力関係について言おうとしていたことは、「SNAFU 原理」によってさらに展開されている。その原理とは、「真のコミュニケーションは対等な者同士の間でしか成立しない。なぜなら、劣る者は上位者に耳障りのいいウソを語ったほうが、真実を語るよりも報酬を得る見込みが高いからだ。」創造的なチームワークは、まさに真のコミュニケーションに依存していて、だからそこに権力関係が入り込むと、かなり深刻に足を引っ張られる。オープンソース・コミュニティは、こういう力関係からは実質的に自由で、しかもそういう力関係がバグや機会損失という面でどんなに高いコストを支払うことになるのか、ということを反面教師的に教えてくれているわけだ。

さらに、SNAFU 原理は権威主義的な組織において、意志決定者たちと現実の間がだんだん乖離していくと預言している。というのも、意志決定者の耳に入る入力は、ますます耳障りのいいウソばかりになってくるからだ。これが従来のソフト開発でどう効いてくるかというのは、すぐにわかる。下位の者たちには、問題を隠し、無視して、過小評価する強いインセンティブがある。このプロセスが製品となったら、ソフトウェアは悲惨なことになる。

16 バージョンと変更履歴

$Id: cathedral-bazaar.sgml,v 1.351998/03/1303:56:18esrExp$

バージョン 1.16: 1997 年 5 月 21 日、Linux Kongress にて発表。

バージョン 1.20: 1997 年 7 月 7 日、文献リストを追加。

バージョン 1.27: 1997 年 11 月 18 日、Perl 会議での逸話を追加。

バージョン 1.29: 1998 年 2 月 9 日、「フリーソフト」を「オープンソース」に変更[*22]。

バージョン 1.31: 1998 年 2 月 10 日、「エピローグ：Netscape もバザール方式を受け入れる」の章を追加。

バージョン 1.4: 1998 年 7 月 28 日に RMS から強力な意見をもらったのを受けて、Paul Eggart による GPL vs バザール方式に関する見解を削除。ハロウィーン文書に基づいてブルックスに訂正を加筆。

バージョン 1.44: 1999 年 7 月末に、「マネジメントとマジノ線について」、デザイン空間探求のためのバザールの有用性を追加、エピローグの大幅加筆。

バージョン 1.45: 1999 年 8 月 8 日に、Snafu 原理について、バザール開発（前）史事例、オリジナリティに関する脚注を追加。

これ以外の変更は、細かい編集上の変更やマークアップの変更を反映したもの。

(翻訳上のミスについて、高瀬 俊朗、山根 信二 <s-yamane@vacia.is.tohoku.ac.jp>、白田秀彰<hideaki@leo.misc.hit-u.ac.jp>、井上友一<marutomo@jade.dti.ne.jp>の各氏からご指摘をいただいた。伏して感謝する——訳者記す)

[*22] 「オープンソース」という用語はまだ日本語として普及していないと判断して、翻訳では併記している。