A Catedral e o Bazar

Eric S. Raymond

A Catedral e o Bazar

Eric Steven Raymond (*The Cathedral & the Bazaar*) – Autor

Disponível em:
http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/

Página na Web: http://www.catb.org/esr/

Eric Kohler – Tradutor

Disponível em:
http://www.geocities.com/CollegePark/Union/3590/pt-cathedral-bazaar.html

Página na Web:
http://www.geocities.com/CollegePark/Union/3590/

Traduzido por Erik Kohler. <ekohler at programmer.net>

José Lopes O. Júnior – Editor

Disponível em:
http://joselopes.org/documents/a_catedral_eo_bazar.pdf

Página na Web: http://joselopes.org/

Notas da edição: Nos capítulos *Apresentação* e *Libere Cedo, Libere Frequentemente*, a tradução original da frase de Eric Raymond *"Given enough eyeballs, all bugs are shallow."* foi trocada de *"Dados bastante olhos, todos os erros são triviais."* para *"Havendo olhos suficientes, todos os erros são óbvios."*.

03/05/2010

Índice

Apresentação

Eu analiso um projeto bem sucedido de código livre, o *fetchmail*, que foi executado como um teste deliberado de algumas teorias surpreendentes sobre a tecnologia de programação sugerida pela história do *Linux*. Eu discuto estas teorias nos termos de dois estilos fundamentais diferentes de desenvolvimento, o modelo CATEDRAL da maior parte do mundo comercial contra o modelo BAZAR do mundo do *Linux*. Eu mostro que estes modelos derivam de suposições opostas sobre a natureza da tarefa de depurar o *software*. Eu faço então um argumento sustentado na experiência do *Linux* para a proposição que *"Havendo olhos suficientes, todos os erros são óbvios."*, sugiro analogias produtivas com outros sistemas auto-corrigíveis de agentes egoístas e concluo com alguma exploração das implicações desta introspecção para o futuro do *software*.

1. A Catedral e o Bazar

Linux é subversivo. Quem pensaria, mesmo há cinco anos atrás, que um sistema operacional de classe mundial poderia surgir como que por mágica pelo tempo livre de milhares de colaboradores espalhados por todo o planeta, conectados somente pelos tênues cordões da Internet?

Certamente não eu. No tempo em que o *Linux* apareceu em minha tela-radar no início de 1993, eu já tinha me envolvido no desenvolvimento de *Unix* e de código aberto por dez anos. Eu fui um dos primeiros contribuintes para o projeto GNU em meados de 1980. Eu tinha liberado bastante do *software* livre na rede, desenvolvendo ou co-desenvolvendo diversos programas (*nethack*, *Emacs* em modo VC e GUD, *xlife* e outros) que estão ainda em largo uso hoje. Eu pensei que eu sabia como isso era feito.

O *Linux* ultrapassou muito o que eu pensei que sabia. Eu estava pregando o modo *Unix* de uso de pequenas ferramentas, de prototipagem rápida e de programação evolucionária por anos. Mas eu acreditei também que havia alguma complexidade crítica, acima da qual uma aproximação mais centralizada, mais *a priori* era requerida. Eu acreditava que os *software*s mais importantes (sistemas operacionais e ferramentas realmente grandes como *Emacs*) necessitavam ser construídos como as catedrais, habilmente criados com cuidado por mágicos ou pequenos grupos de magos trabalhando em esplêndido isolamento, com nenhum beta para ser liberado antes de seu tempo.

O estilo de Linus Torvalds de desenvolvimento – libere cedo e frequentemente, delegue tudo que você possa, esteja aberto ao ponto da promiscuidade – veio como uma surpresa. Nenhuma catedral calma e respeitosa aqui – ao invés, a comunidade *Linux* pareceu assemelhar-se a um grande e barulhento bazar de diferentes agendas e aproximações (adequadamente

simbolizada pelos repositórios do *Linux*, que aceitaria submissões de qualquer pessoa) de onde um sistema coerente e estável poderia aparentemente emergir somente por uma sucessão de milagres.

O fato de que este estilo bazar pareceu funcionar e funcionar bem, veio como um distinto choque. Conforme eu aprendia ao meu redor, trabalhei duramente não apenas em projetos individuais, mas também tentando compreender porque o mundo do *Linux* não somente não se dividiu em confusão mas parecia aumentar a sua força a uma velocidade quase inacreditável para os construtores de catedrais.

Em meados de 1996 eu pensei que eu estava começando a compreender. O acaso deu-me uma maneira perfeita para testar minha teoria, na forma de um projeto de código aberto que eu poderia conscientemente tentar executar no estilo bazar. Assim eu fiz – e foi um sucesso significativo.

No resto deste artigo eu contarei a história desse projeto e eu irei usá-la para propor alguns aforismos sobre o desenvolvimento eficaz de código aberto. Nem tudo eu aprendi primeiramente no mundo do *Linux*, mas nós veremos como o mundo do *Linux* lhe dá um ponto particular. Se eu estiver correto, eles o ajudarão a compreender exatamente o que é que faz a comunidade do *Linux* ser uma fonte de *software* bom – e ajuda a você a se tornar mais produtivo.

2. O Correio Deve Ser Entregue

Desde 1993 eu venho cuidando da parte técnica de um pequeno Provedor de Acesso Internet de acesso gratuito chamado *Chester County InterLink* (CCIL) em *West Chester, Pensilvânia* (eu fui co-fundador do CCIL e escrevi nosso *software* multiusuário de BBS – você pode observá-lo executando um `telnet` para `locke.ccil.org`. Hoje suporta quase três mil usuários em trinta linhas.). O trabalho permitiu-me o acesso 24 horas por dia à rede através da linha de 56K do CCIL – de fato, praticamente exigiu!

Consequentemente, eu fiquei acostumado a acesso instantâneo ao correio Internet. Por razões complicadas, era difícil fazer o SLIP funcionar entre minha máquina de casa (`snark.thyrsus.com`) e o CCIL. Quando eu finalmente consegui, eu achei incômodo ter que executar *telnet* periodicamente para o *locke* para verificar meu correio. O que eu queria era que ele fosse enviado para o *snark* de modo que eu fosse notificado quando uma mensagem chegasse e pudesse manuseá-lo usando todas as minhas ferramentas locais.

O simples reenvio do *sendmail* não funcionaria, porque minha máquina local não está sempre na rede e não tem um IP estático. O que eu precisava era um programa que pegasse meu correio através da conexão SLIP e o entregasse localmente. Eu sabia que tais programas existiam e que a maioria deles usava um protocolo de aplicação simples chamado POP (*Post Office Protocol*). E, realmente, já havia um servidor POP3 incluído com sistema operacional *BSD/OS* do *locke*.

Eu precisava de um cliente POP3. Então eu procurei na Internet e encontrei um. Na verdade, eu encontrei três ou quatro. Eu usei o *pop-perl* por algum tempo, mas faltava o que parecia uma característica óbvia, a habilidade de alterar os endereços no correio recebido para que as respostas fossem

enviadas corretamente.

O problema era este: suponha que alguém chamado *joe* no *locke* tenha me enviado uma mensagem. Se eu capturasse o correio para o *snark* e tentasse então lhe responder, meu programa de correio tentaria alegremente enviá-lo a um *joe* inexistente no *snark*. Editar manualmente os endereços de resposta para adicionar `@ccil.org` rapidamente tornou-se um tormento.

Isto era claramente algo que o computador teria que fazer para mim. Mas nenhum dos clientes POP existentes sabiam como! E isto nos traz à primeira lição:

1. Todo bom trabalho de *software* começa colocando o dedo na ferida de um programador.

Talvez isto deveria ter sido óbvio (um antigo provérbio diz que *"A necessidade é a mãe da invenção"*) mas muitas vezes os programadores gastam seus dias buscando retorno em programas que eles não necessitam nem gostam. Mas não no mundo do *Linux* – o que pode explicar porque a qualidade média do *software* originada na comunidade de *Linux* é tão alta.

Assim, eu me lancei imediatamente com o ímpeto de codificar um novo cliente POP3 para competir com os existentes? De maneira alguma! Eu olhei com cuidado os utilitários POP que eu tinha à disposição, perguntando-me *"qual deles é o mais próximo do que eu quero?"*. Porque...

2. Os programadores bons sabem o que escrever. O grandes sabem o que rescrever (e reusar).

Embora eu não me considere um grande programador, eu tento me passar por um. Uma importante característica dos grandes é a preguiça construtiva. Eles sabem que você ganha um **A**, não por esforço, mas por resultados e é quase sempre mais fácil partir de uma boa solução parcial do que

do nada.

Linus Torvalds, por exemplo, não tentou realmente escrever *Linux* do nada. Ao contrário, ele começou reusando código e idéias do *Minix*, um pequeno sistema operacional *Unix-like* para máquinas 386. Eventualmente todo o código *Minix* se foi, ou foi completamente rescrito – mas quando estava lá, forneceu as bases para o infante que se transformaria no *Linux*.

Com o mesmo pensamento, eu fui procurar um utilitário POP que fosse razoavelmente bem codificado, para usar como uma base de desenvolvimento.

A tradição do mundo *Unix* de compartilhar o código fonte foi sempre amigável para a reutilização de código (esta é a razão porque o projeto de GNU escolheu o *Unix* como sistema operacional base, apesar das sérias reservas sobre o mesmo). O mundo *Linux* tem levado esta tradição quase a seu limite tecnológico; tem terabytes de códigos abertos amplamente disponíveis. Assim gastando tempo procurando algum *software* quase-bom-o-bastante feito por alguém é mais provável dar a você mais bons resultados no mundo *Linux* do que em qualquer outro lugar.

E fez para mim. Com aqueles que eu achei antes, minha segunda busca compôs um total de nove candidatos – *fetchpop*, *PopTart*, *get-mail*, *gwpop*, *pimp*, *pop-perl*, *popc*, *popmail* e *upop*. O primeiro que eu trabalhei primeiramente era o *fetchpop* por Seung-Hong Oh. Eu pus o meu código de reescrita de cabeçalho nele e fiz várias outras melhorias que o autor aceitou em sua versão 1.9.

Algumas semanas mais tarde, entretanto, eu tropecei pelo código do *popclient* desenvolvido por Carl Harris e descobri que eu tinha um problema. Embora o *fetchpop* tenha tido algumas idéias boas e originais (tal como o modo

daemon), podia somente trabalhar com POP3 e foi codificado de uma maneira um tanto amadora (Seung-Hong era um programador brilhante porém inexperiente e ambas as características ficaram evidentes). O código de Carl era melhor, bastante profissional e sólido, mas em seu programa faltou diversas características importantes e complicadas de serem implementadas do *fetchpop* (incluindo aquelas que eu implementei).

Ficar ou trocar? Se eu trocasse, eu estaria jogando fora o código que eu já havia feito em troca de uma base melhor do desenvolvimento.

Um motivo prático para trocar era a presença de suporte para vários protocolos. POP3 é o protocolo mais comumente utilizado de todos os protocolos de correio dos servidores, mas não o único. *fetchpop* e os outros concorrentes não faziam POP2, RPOP ou APOP e eu estava realmente tendo alguns pensamentos de talvez adicionar IMAP (*Internet Message Access Protocol* ou Protocolo de Acesso a Mensagem da Internet – o mais recente e poderoso protocolo de correio desenvolvido) só para me divertir.

Mas eu tinha uma razão mais teórica para pensar que trocar seria uma boa idéia, alguma coisa que eu aprendi muito antes do *Linux*.

3. *"Planeje jogar algo fora; você irá, de qualquer maneira."* (Fred Brooks, *"The Mythical Man-Month"*, Capítulo 11)

Ou, colocando de outra forma, você frequentemente não entende realmente o problema até depois da primeira vez que você implementa uma solução. Na segunda vez, talvez você saiba o suficiente para fazer corretamente. Então se você quer fazer algo certo, esteja preparado para começar tudo novamente pelo menos uma vez.

Bem (eu disse para mim mesmo) as mudanças no *fetchpop* foram a minha primeira tentativa. Então eu troquei.

Depois que eu mandei meu primeiro conjunto de alterações para Carl Harris em 25 de junho de 1996, eu percebi que na verdade ele perdera o interesse pelo *popclient* há algum tempo até então. O código era um pouco empoeirado, com pequenos erros. Eu tinha muitas mudanças para fazer e nós rapidamente concordamos que o lógico para eu fazer era tomar conta do programa.

Sem perceber, o projeto havia se intensificado. Eu não estava mais contemplando somente pequenos consertos para um cliente POP existente. Eu estava mantendo um cliente completo e havia idéias borbulhando na minha cabeça que eu sabia iriam provavelmente levar a importantes mudanças.

Em uma cultura de *software* que encoraja a troca de código, isto é um caminho natural para um projeto evoluir. Eu estava representando isto:

4. Se você tem a atitude certa, problemas interessantes irão encontrá-lo.

Mas a atitude do Carl Harris foi ainda mais importante. Ele entendeu que:

5. Quando você perde interesse em um programa, sua última obrigação a fazer com ele é entregá-lo a um sucessor competente.

Sem ao menos ter que discutir isso, Carl e eu sabíamos que nós tínhamos um objetivo em comum de ter a melhor solução disponível. A única questão para nós foi se eu poderia me estabelecer como um par de mãos seguras para isso. Uma vez que eu tenha feito isso, ele agiu com cortesia e rapidez. Eu espero que eu aja assim quando chegar a minha vez.

3. A Importância de ter Usuários

E então eu herdei o *popclient*. Tão importante quanto, eu herdei os usuários do *popclient*. Usuários são ótimas coisas para se ter e não somente porque eles demonstram que você está satisfazendo uma necessidade, que você fez algo certo. Cultivados de maneira adequada, eles podem se tornar co-desenvolvedores.

Outra força da tradição do *Unix*, uma que o *Linux* tem levado a um alegre extremo, é que muitos usuários são hackers também. E porque o código fonte está disponível, eles podem ser hackers eficazes. Isto pode ser tremendamente útil para reduzir o tempo de depuração. Com um pouco de estímulo, seus usuários irão diagnosticar problemas, sugerir correções e ajudar a melhorar o código muito mais rapidamente do que você poderia fazer sem ajuda.

6. Tratar seus usuários como co-desenvolvedores é seu caminho mais fácil para uma melhora do código e depuração eficaz.

O poder deste efeito é fácil de se subestimar. De fato, todos nós do mundo do código aberto subestimamos drasticamente como isto iria incrementar o número de usuários e diminuir a complexidade do sistema, até que Linus Torvalds nos mostrou de outra forma.

De fato, eu penso que a engenhosidade do Linus e a maior parte do que desenvolveu não foram a construção do *kernel* do *Linux* em si, mas sim a sua invenção do modelo de desenvolvimento do *Linux*. Quando eu expressei esta opinião na sua presença uma vez, ele sorriu e calmamente repetiu algo que frequentemente diz: *"Sou basicamente uma pessoa muito preguiçosa que gosta de ganhar crédito por coisas que outras pessoas realmente fazem."* Prequiçoso como uma raposa. Ou, como Robert Heinlein teria dito, muito preguiçoso para

falhar.

Em retrospecto, um precedente para o sucesso e métodos do *Linux* pode ser observado no desenvolvimento da biblioteca do GNU *Emacs Lisp* e os repositórios de código *Lisp*. Em contraste com o estilo de desenvolvimento catedral do núcleo do *Emacs C* e a maioria das outras ferramentas da FSF, a evolução do grupo de código *Lisp* foi fluída e bastante dirigida ao usuário. Idéias e protótipos foram frequentemente reescritos três ou quatro vezes antes de atingirem uma forma estável final. E colaborações permitidas pela Internet, a la *Linux*, foram frequentes.

Realmente, minha mais bem sucedida codificação anterior ao *fetchmail* foi provavelmente o modo *Emacs* VC, uma colaboração do tipo do *Linux* por email com três outras pessoas, somente uma das quais (Richard Stallman, o autor do *Emacs* e fundador da FSF) eu conheci pessoalmente até hoje. Foi um *front-end* para SCCS, RCS e posteriormente CVS pelo *Emacs* que oferecia operações de controle de versão “um toque”. Evoluiu de um pequeno e grosseiro modo *sccs.el* que alguém havia escrito. E o desenvolvimento do VC foi bem sucedido porque, ao contrário do próprio *Emacs*, o código do *Emacs Lisp* poderia ir por gerações de lançamento/teste/aperfeiçoamento muito rapidamente.

4. Libere Cedo, Libere Frequentemente

Liberações novas e frequentes são uma parte crítica do modelo de desenvolvimento do *Linux*. A maioria dos desenvolvedores (incluindo eu) costumava acreditar que esta era uma má política para projetos maiores que os triviais, porque versões novas são quase por definição cheias de erros e você não quer acabar com a paciência dos seus usuários.

Esta crença reforçou o compromisso de todos com o estilo de desenvolvimento catedral. Se o principal objetivo era o de usuários verem menos erros quanto possível, por que então você iria somente lançar um em cada seis meses (ou frequentemente menos) e trabalhar como um cachorro depurando entre as liberações. O núcleo do *Emacs C* foi desenvolvido desta forma. A biblioteca *Lisp*, de fato, não foi – porque havia repositórios *Lisp* ativos fora do controle da FSF, aonde você poderia ir para achar versões novas e em desenvolvimento, independentemente do ciclo de liberação do *Emacs*.

A mais importante destas, o repositório *elisp* do Estado de Ohio, antecipou o espírito e muitas das características dos atuais grandes repositório de *Linux*. Mas poucos de nós realmente pensaram muito sobre o que estávamos fazendo, ou sobre o que a existência deste repositório sugeriu sobre problemas no modelo de desenvolvimento catedral da FSF. Eu fiz uma séria tentativa por volta de 1992 para ter bastante código de Ohio formalmente fundido na biblioteca oficial do *Emacs Lisp*. Encontrei problemas políticos e fui muito mal sucedido.

Mas um ano depois, visto que o *Linux* se tornou amplamente conhecido, ficou claro que alguma coisa diferente e muito saudável estava acontecendo. A política de desenvolvimento aberta do Linus era exatamente o oposto do modelo de desenvolvimento catedral. Os repositórios *sunsite* e *tsx-11*

estavam germinando, múltiplas distribuições estavam surgindo. E tudo isto foi guiado por uma frequência desconhecida de liberações de núcleo de sistemas.

Linus estava tratando seus usuários como co-desenvolvedores na maneira mais eficaz possível:

7. Libere cedo. Libere frequentemente. E ouça seus fregueses.

Isso não era muito a inovação do Linus (algo como isso estava sendo a tradição do mundo *Unix* por um longo tempo), mas em elevar isto até um grau de intensidade que alcançava a complexidade do que ele estava desenvolvendo. Nestes primórdios tempos (por volta de 1991) não era estranho para ele liberar um novo *kernel* mais de uma vez por dia! E, porque ele cultivava sua base de co-desenvolvedores e incitava fortemente a Internet por colaboração como nenhum outro, isto funcionou.

Mas como isto funcionou? E era isto algo que eu poderia duplicar, ou era algo que dependia da genialidade única de Linus Torvalds?

Eu não pensei assim. Reconhecidamente, Linus é um excelente hacker (quantos de nós poderia planejar um *kernel* completo de um sistema operacional de qualidade de produção?). Mas o *Linux* não representou nenhum salto conceitual impressionante a frente. Linus não é (ou pelo menos, ainda não) um gênio inovativo de projeto do estilo que, por exemplo, Richard Stallman ou James Gosling (do *NeWS* e *Java*) são. Ao contrário, para mim Linus parece ser um gênio da engenharia, com um sexto sentido em evitar erros e desenvolvimentos que levem a um beco sem saída e uma verdadeira habilidade para achar o caminho do menor esforço do ponto A ao ponto B. De fato, todo o projeto do *Linux* exala esta qualidade e espelha a abordagem conservadora e simplificada de planejamento do Linus.

Então, se liberações rápidas e influenciar a Internet a todo custo

não foram acidentes mas partes integrantes da perspicácia do gênio de engenharia do Linus para o caminho do menor esforço, o que ele estava enfatizando? O que ele estava maquinando?

Posto desta forma, a questão responde a si mesma. Linus estava mantendo seus usuários/hackers constantemente estimulados e recompensados – estimulados pela perspectiva de estarem tendo um pouco de ação satisfatória do ego, recompensados pela visão do constante (até mesmo diário) melhoramento do seu trabalho.

Linus estava diretamente direcionado a maximizar o número de pessoas-hora dedicadas à depuração e ao desenvolvimento, mesmo com o possível custo da instabilidade no código e extinção da base de usuários se qualquer erro sério provasse ser intratável. Linus estava se comportando como se acreditasse em algo como isto:

8. Dada uma base grande o suficiente de *beta-testers* e co-desenvolvedores, praticamente todo problema será caracterizado rapidamente e a solução será óbvia para alguém.

Ou, menos formalmente, *"Havendo olhos suficientes, todos os erros são óbvios."* Eu chamo isso de: "Lei de Linus".

Minha formulação original foi que todo problema "será transparente para alguém". Linus objetou que a pessoa que entende e conserta o problema não é necessariamente ou mesmo frequentemente a pessoa que primeiro o caracterizou. "Alguém acha o problema" – ele diz – "e uma outra pessoa o entende. E eu deixo registrado que achar isto é o grande desafio." Mas o ponto é que ambas as coisas tendem a acontecer rapidamente.

Aqui, penso eu, é o centro da diferença fundamental entre os estilos bazar e catedral. Na visão catedral de programação, erros e problemas

de desenvolvimento são difíceis, insidiosos, um fenômeno profundo. Leva meses de exame minucioso por poucas pessoas dedicadas para desenvolver confiança de que você se livrou de todos eles. Por conseguinte os longos intervalos de liberação e o inevitável desapontamento quando as liberações por tanto tempo esperadas não são perfeitas.

Na visão bazar, por outro lado, você assume que erros são geralmente um fenômeno trivial – ou, pelo menos, eles se tornam triviais muito rapidamente quando expostos para centenas de ávidos co-desenvolvedores triturando cada nova liberação. Consequentemente você libera frequentemente para ter mais correções e como um benéfico efeito colateral você tem menos a perder se um erro ocasional aparece.

E é isto. É o suficiente. Se a "Lei de Linus" é falsa, então qualquer sistema tão complexo como o *kernel* do *Linux*, sendo programado por tantas mãos quantas programam o *kernel* do *Linux*, deveria a um certo ponto tido um colapso sob o peso de interações imprevisíveis e erros "profundos" não descobertos. Se isto é verdade, por outro lado, é suficiente para explicar a relativa falta de erros do *Linux*.

E talvez isso não deveria ser uma surpresa, mesmo assim. Anos atrás, sociologistas descobriram que a opinião média de uma massa de observadores especialistas (ou igualmente ignorantes) é um indicador mais seguro que o de um único observador escolhido aleatoriamente. Eles chamaram isso de o "Efeito Delphi". Parece que o que o Linus tem mostrado é que isto se aplica até mesmo para depurar um sistema operacional – que o Efeito Delphi pode suavizar a complexidade do desenvolvimento até mesmo em nível de complexidade do kernel de um sistema operacional.

Eu sou grato a Jeff Dutky <dutky@wam.umd.edu> por apontar que a Lei de Linus pode ser refeita como "Depurar é paralelizável". Jeff

observa que embora depurar requer que depuradores se comuniquem com algum desenvolvedor coordenador, não requer coordenação significante entre depuradores. Assim não cai vítima para a mesma complexidade quadrática e custos de gerência que faz ser problemático adicionar desenvolvedores.

Na prática, a perda teórica de eficiência devido à duplicação de trabalho por depuradores quase nunca parece ser um problema no mundo do *Linux*. Um efeito da "política libere cedo e frequentemente" é minimizar esta duplicação, propagando consertos rapidamente.

Brooks até mesmo fez uma observação improvisada relacionada à observação de Jeff: "O custo total de manter um programa amplamente utilizado é tipicamente 40% ou mais o custo de desenvolvê-lo. Surpreendentemente este custo é muito afetado pelo número de usuários. Mais usuários acham mais erros." (minha ênfase)

Mais usuários acham mais erros porque adicionar mais usuários adiciona mais maneiras diferentes de testar o programa. Este efeito é amplificado quando os usuários são co-desenvolvedores. Cada um aborda a tarefa de caracterização de erro com um conjunto perceptivo ligeiramente diferente e ferramenta analítica, um ângulo diferente do problema. O Efeito Delphi parece funcionar precisamente por causa desta variação. No contexto específico da depuração, a variação também tende a reduzir o feito da duplicação.

Então adicionar mais *beta-testers* pode não reduzir a complexidade de um erro "profundo" corrente do ponto de vista do desenvolvedor, mas aumenta a probabilidade que a ferramenta de alguém irá de encontro ao problema de uma maneira tal que o problema será trivial para esta pessoa.

Linus cobre suas apostas. Caso haja erros sérios, as versões do *kernel* do *Linux* são numeradas de forma que usuários em potencial podem fazer a escolha de executar a última versão designada “estável” ou correr o risco de encontrar erros para obter novas características. Esta técnica não é ainda formalmente imitada pela maioria dos hackers usuários do *Linux*, mas talvez devesse; o fato de que ambas as escolhas estejam disponíveis faz delas mais atraentes.

5. Quando uma Rosa não é uma Rosa?

Tendo estudado o comportamento do Linus e formado uma teoria sobre como ele foi bem sucedido, eu fiz uma decisão consciente para testar esta teoria em meu novo (reconhecidamente muito menos complexo e ambicioso) projeto.

Mas a primeira coisa que eu fiz foi reorganizar e simplificar bastante o *popclient*. A implementação do Carl Harris era muito atrativa, mas exibia um tipo de complexidade desnecessária comum a vários programadores em *C*. Ele tratou o código como centro das atenções e as estruturas de dados como suporte para o código. Como resultado, o código era muito bonito mas o projeto da estrutura de dados era *ad-hoc* e um tanto feio (ao menos pelos altos padrões deste velho hacker de *LISP*).

Eu tinha outro propósito para reescrever além de aperfeiçoar o código e o projeto da estrutura de dados, entretanto. Era evolui-lo para algo que eu entenderia completamente. Não é nada agradável ser responsável por consertar erros em um programa que você não entende.

Mais ou menos durante o primeiro mês, então, eu estava simplesmente seguindo as implicações do projeto básico do Carl. A primeira mudança séria que fiz foi adicionar suporte ao IMAP. Eu fiz isso reorganizando as rotinas de protocolos em um *driver* genérico e três tabelas de métodos (para POP2, POP3 e IMAP). Esta e as mudanças anteriores ilustram o princípio geral que é bom para os programadores manterem em mente, especialmente em linguagens como C que não implementam naturalmente tipagem dinâmica:

9. Estrutura de dados inteligentes e código burro trabalham muito melhor que ao contrário.

Brooks, Capítulo 11: *"Mostre-me seu [código] e esconda suas [estruturas de dados] e eu poderei continuar mistificado. Mostre-me suas [estruturas de dados] e eu provavelmente não necessitarei do seu [código]; ele será óbvio."*

De fato, ele disse "gráficos" e "tabelas". Mas considerando trinta anos de terminologias/mudanças culturais, é praticamente o mesmo ponto.

Neste ponto (quase setembro de 1996, mais ou menos seis semanas desde o início) eu comecei a pensar que uma mudança de nome deveria ser adequada – afinal de contas, não era mais somente um cliente POP. Mas eu hesitei, porque ainda não havia ainda nada genuinamente novo ainda no projeto. Minha versão do *popclient* ainda teria que desenvolver uma identidade própria.

Isto mudou radicalmente quando o *fetchmail* aprendeu como reenviar mensagens recebidas para a porta SMTP. Eu irei a este ponto em um momento. Mas primeiro: Eu disse acima que eu decidi usar este projeto para testar minha teoria sobre o que Linus Torvalds fez corretamente. Como (você pode perguntar) eu fiz isto? Desta forma:

1. Eu liberei cedo e frequentemente (quase nunca menos que uma vez a cada dez dias; durante períodos de desenvolvimento intenso, uma vez por dia).
2. Eu aumentei minha lista de *beta-testers* adicionando à ela todos que me contatavam sobre o *fetchmail*.
3. Eu mandei extensos anúncios à lista de *beta-testers* toda vez que eu liberava, encorajando as pessoas a participar.
4. E eu ouvia meus *beta-testers*, questionando-os sobre decisões de desenvolvimento e incitando-os toda vez que eles mandavam consertos e respostas.

O retorno destas medidas simples foi imediato. Desde o início do projeto, eu obtive relatórios sobre erros de uma qualidade que a maioria dos desenvolvedores morreria para ter, muitas vezes com ótimos consertos incluídos. Eu obtive críticas severas, obtive mensagens de fãs, obtive sugestões inteligentes de características. O que leva a:

10. Se você tratar seus beta-testers como seu recurso mais valioso, eles irão responder tornando-se seu mais valioso recurso.

Uma medida interessante do sucesso do *fetchmail* é o simples tamanho da lista de *beta-testers*, amigos do *fetchmail*. Ao escrever este texto ela possuía 249 membros e são adicionados dois ou três por semana.

De fato, conforme eu reviso, ao final de maio de 1997, a lista está começando a perder membros depois do pico de aproximadamente 300 pessoas por uma razão interessante. Muitas pessoas têm me pedido para exclui-las da lista porque o *fetchmail* está trabalhando tão bem para elas que elas não precisam mais ver o tráfego da lista! Talvez isto seja parte do ciclo de vida normal de um projeto maduro do estilo bazar.

6. *popclient* transforma-se em *fetchmail*

O verdadeiro ponto de mudança no projeto foi quando Harry Hochheiser me mandou o seu rascunho de código para reenviar mensagens para a porta SMTP da máquina cliente. Eu percebi quase imediatamente que uma implementação segura desta característica iria tornar todos os outros modos de envio perto do obsoleto.

Por muitas semanas eu fiquei customizando o *fetchmail* de maneira incremental enquanto percebia como o projeto de interface estava aproveitável mas feio – não estava elegante e com muitas pequenas opções por toda parte. As opções para extrair mensagens coletadas para um arquivo de mensagens ou saída padrão particularmente me aborreciam, mas eu não conseguia descobrir por que.

O que eu vi quando eu pensei sobre reenvio SMTP foi que o *popclient* estava tentando fazer muitas coisas. Ele foi projetado para ser tanto um agente transportador de correspondência (MTA) quanto um agente local de entrega (MDA). Com reenvio SMTP, ele poderia retirar o MDA e ser somente um MTA, transferindo as correspondências para outros programas para entrega local como o *sendmail* faz.

Por que se envolver com toda a complexidade de configurar um agente de entrega de correspondência ou configurar *lock-e-append* em um arquivo de correio, quando a porta 25 é quase garantida para estar lá, em primeiro lugar em qualquer plataforma, com suporte para TCP/IP? Especialmente quando isso significa que o correio coletado é garantido parecer como um correio SMTP iniciado pelo remetente normalmente, o que, de qualquer maneira, é realmente o que queremos.

Há várias lições aqui. Primeira, esta idéia de reenvio por SMTP foi o primeiro grande retorno que eu obtive por tentar conscientemente emular os métodos do Linus. Um usuário me deu esta idéia brilhante – tudo que eu tive que fazer foi entender as implicações.

11. A melhor coisa depois de ter boas idéias é reconhecer boas idéias dos seus usuários. Às vezes, a última é melhor.

E o que é interessante: você irá rapidamente descobrir que, se você está se achando completamente desacreditado sobre o quanto você deve a outras pessoas, o mundo inteiro irá tratá-lo como se você mesmo tivesse criado cada bit da invenção e está somente sendo modesto sobre seu gênio inato. Nós todos podemos ver o quanto isso funcionou para Linus!

> Quando eu mostrei este documento na conferência de *Perl*, em agosto de 1997, Larry Wall estava na primeira fila. Quando eu li a última linha acima ele gritou fervorosamente, em um estilo religioso nostálgico, “Diga, diga, irmão!”. Toda a audiência riu, porque eles sabiam que isso também funcionou para o criador do *Perl*.

Após poucas semanas executando o projeto no mesmo espírito, eu comecei a ganhar um louvor semelhante não apenas dos meus usuários mas de outras pessoas que deixaram as palavras escaparem. Eu guardei algumas destas mensagens; irei olhar para elas novamente algum dia se eu começar a questionar se a minha vida valeu a pena **:-)**.

Mas há duas lições mais fundamentais aqui, não políticas, que são gerais para todos os tipos de projeto.

12. Frequentemente, as soluções mais impressionantes e inovadoras, surgem ao se perceber que o seu conceito do problema estava

errado.

Eu estava tentando resolver o problema errado ao desenvolver continuamente o *popclient* como um MTA/MDA combinado com todos os tipos de modos de distribuição local. O projeto do *fetchmail* precisava ser repensado do início como um puro MTA, uma parte do caminho normal do correio da Internet que fala SMTP.

Quando você atinge uma parede ao desenvolver – quando você dificilmente se encontra pensando em algo depois do próximo *patch* – é, frequentemente, tempo para perguntar, não se você tem a resposta certa, mas se você está perguntando a pergunta correta. Talvez o problema precise ser reformulado.

Bem, eu reformulei meu problema. Claramente, a coisa certa a fazer era: **1.** codificar o suporte para reenvio por SMTP no *driver* genérico. **2.** tornar isto o modo padrão. **3.** eventualmente desfazer todos os outros modos de envio, especialmente as opções de enviar-para-arquivo e enviar-para-saída-padrão.

Eu hesitei sobre o passo 3 por algum tempo, temendo aborrecer os usuários de longa data do *popclient* dependentes dos mecanismos alternativos de envio. Em teoria, eles poderiam imediatamente passar a usar arquivos `.forward` ou seus equivalentes *não-sendmail* para obter os mesmos efeitos. Na prática a transição poderia ser confusa.

Mas quando eu a fiz, os benefícios provaram-se altos. As partes obscuras do código do *driver* sumiram. A configuração ficou radicalmente mais simples – não mais rastejar a volta atrás do MDA do sistema e da caixa de correio do usuário, não mais preocupações sobre se o sistema operacional suportava *locking* de arquivo.

Além disso, a única maneira de perder correio sumira. Se você especificava envio para um arquivo e o disco estava cheio, seu correio estaria perdido. Isto não pode acontecer com o reenvio por SMTP porque o receptor SMTP não retornará OK a não ser que a mensagem possa ser enviada ou pelo menos reprogramada para um envio mais tarde.

E também a performance aumentou (embora você não perceberia isso somente com uma execução). Outro benefício não insignificante foi que a página do manual ficou muito mais simples.

Mais tarde, eu tive que recolocar envio por um MDA local especificado pelo usuário para permitir o tratamento de algumas situações obscuras envolvendo SLIP dinâmico. Mas eu encontrei uma maneira muito mais simples de fazer isto.

A moral? Não hesite em jogar fora características obsoletas quando você puder fazer isso sem perda de efetividade. Antoine de Saint-Exupéry (que foi um aviador e projetista de aviões quando ele não estava sendo o autor de livros clássicos infantis) disse:

13. "A perfeição [em projetar] é alcançada não quando não há mais nada a adicionar, mas quando não há nada para jogar fora."

Quando o seu código está ficando tão bom quanto simples, é quando você sabe que está correto. E no processo, o projeto do *fetchmail* adquiriu uma identidade própria, diferente do ancestral *popclient*.

Era tempo para a mudança de nome. O novo projeto parecia muito mais como um irmão do *sendmail* do que o velho *popclient* parecia; ambos eram MTAs, mas aonde o *sendmail* passa e então envia, o novo *popclient* coleta e então envia. Então, após 2 meses, eu mudei o nome para *fetchmail*.

7. *fetchmail* Cresce

Lá estava eu com um projeto elegante e inovador, um código que eu sabia que funcionava bem porque eu usava todos os dias e uma crescente lista de *beta-testers*. E gradualmente eu percebia que eu não estava mais ocupado com uma trivial codificação pessoal que porventura poderia se tornar útil para algumas pessoas. Eu tinha em minhas mãos um programa que todos os usuários de um sistema *Unix* com uma conexão de correio SLIP/PPP realmente precisavam.

Com a característica de reenvio por SMTP, ele passou muito à frente da competição para potencialmente se tornar um *"category killer"*, um destes programas clássicos que preenchem seu nicho tão completamente que as alternativas não são somente descartadas como também esquecidas.

Eu penso que você não pode realmente almejar ou planejar um resultado como este. Você tem que ser atraído para isso por idéias de projeto tão poderosas que posteriormente os resultados parecem inevitáveis, naturais, mesmo predestinados. A única maneira de tentar idéias como esta é tendo muitas idéias – ou tendo o julgamento de engenharia para levar as boas idéias das outras pessoas além do que estas que as tiveram pensariam que elas poderiam ir.

Andrew Tanenbaum teve a idéia original de construir um *Unix* nativo simples para o 386, para uso como uma ferramenta de ensino. Linus Torvalds levou o conceito do *Minix* para além do que Andrew provavelmente pensou que poderia ir – e cresceu para algo maravilhoso. Da mesma forma (embora em uma menor escala), eu peguei algumas idéias de Carl Harris e Harry Hochheiser e dei um empurrão. Nenhum de nós foi "original" no sentido romântico que as pessoas pensam é um gênio. Mas a maioria do

desenvolvimento de *software* científico e de engenharia não é feita por um gênio original, a mitologia hacker ao contrário.

Os resultados foram todos sempre muito precipitados – de fato, o tipo de sucesso que qualquer hacker está sempre procurando! E eles significaram que eu teria que definir meus padrões ainda mais altos. Para fazer o *fetchmail* tão bom como eu então achava que poderia se tornar, eu teria que escrever não somente para minhas próprias necessidades, mas também incluir e suportar características necessárias para outros fora do meu relacionamento. E fazer isto mantendo o programa simples e robusto.

A primeira e mais importante esmagadora característica que eu escrevi depois de perceber isto foi o suporte a *multidrop* – a habilidade de capturar correio de caixas de correio que acumularam todas as mensagens para um grupo de usuários e então destinar cada pedaço de correio para seus destinatários individuais.

Eu decidi adicionar o suporte ao *multidrop* em parte, porque alguns usuários estavam pedindo por isto, mas principalmente porque eu pensei que isto retiraria os erros do código para o envio simples, forçando-me a manipular o endereçamento de um modo geral. E assim se provou ser. Fazer funcionar a análise da RFC 822 me tomou um tempo consideravelmente longo, não porque qualquer pedaço dele é difícil mas porque envolvia uma pilha de detalhes interdependentes e elaborados.

Mas o endereçamento *multidrop* se tornou uma decisão de projeto excelente. Aqui está como eu soube:

14. Qualquer ferramenta deve ser útil da maneira esperada, mas uma ferramenta verdadeiramente BOA leva ela própria a usos que você nunca esperou.

O uso inesperado para o *multidrop* do *fetchmail* é executar *mailing lists* com a lista mantida e expansão de apelidos feita, no lado do cliente da conexão SLIP/PPP. Isto significa que alguém executando uma máquina pessoal por uma conta ISP pode administrar uma *mailing list* sem acesso contínuo aos arquivos de apelidos do ISP.

Outra importante mudança solicitada pelos meus *beta-testers* foi suporte para operação MIME 8-bit. Isto foi muito fácil de fazer, porque eu estava sendo cuidadoso mantendo o código pronto para 8-bit. Não porque eu antecipei a demanda para esta característica, mas em obediência a outra regra:

15. Quando escrevendo um *software gateway* de qualquer tipo, faça tudo para perturbar o conjunto de dados o menos possível – e NUNCA jogue fora informação, a não ser que o destinatário force você a isto!

Se eu não tivesse obedecido esta regra, o suporte a MIME 8-bit teria sido difícil e cheio de erros. Como estava, tudo o que tive que fazer foi ler a RFC 1652 e adicionar um pouco de lógica trivial para a geração de cabeçalho.

Alguns usuários europeus insistiram para que eu adicionasse uma opção para limitar o número de mensagens recebidas por seção (de maneira que eles poderiam controlar custos das suas caras redes de telefone). Eu resisti por um longo tempo e ainda não estou inteiramente alegre sobre isto. Mas se você está escrevendo para o mundo, você tem que ouvir os seus clientes – isto não muda somente porque eles não estão pagando dinheiro a você.

8. Algumas Lições a mais do *fetchmail*

Antes de voltarmos ao assunto de engenharia de *software* em geral, existem algumas lições a mais da experiência do *fetchmail* para ponderar.

A sintaxe do arquivo rc inclui palavra-chaves "ruidosas" opcionais que são inteiramente ignoradas pelo analisador. A sintaxe parecida com o inglês que elas permitem, é consideravelmente mais inteligível que os concisos pares tradicionais *palavra-chave-valor* que você obtém quando as retira.

Estas começaram como um experimento posterior quando eu percebi como as declarações do arquivo rc estava começando a lembrar uma minilinguagem imperativa (é por isto também que eu mudei a palavra-chave original "*server*" do *popclient* para "*poll*").

Parecia para mim que tentar fazer esta mini-linguagem imperativa parecer mais como o inglês poderia torná-la mais fácil de ser usada. Agora, embora eu seja um convencido adepto da escola de projeto "faça isso uma linguagem" como exemplificado pelo *Emacs* e HTML e muitos sistemas de banco de dados, eu normalmente não sou um grande fã das sintaxes "*English-like*".

Programadores tradicionais tendem a serem a favor de sintaxes de controle que são muito precisas e compactas e não contém redundância alguma. Isto é um legado cultural de quando os recursos computacionais eram caros, então os estágios de análise tinham que ser tão baratos e simples quanto possíveis. O inglês, com redundância de aproximadamente 50%, parecia ser um modelo muito impróprio.

Esta não é minha razão para normalmente evitar sintaxes no estilo do inglês; eu menciono isto aqui somente para arrasá-la. Com ciclos e núcleos baratos, concisão não deve ser ela mesma um fim. Hoje em dia é mais importante para uma linguagem ser conveniente para humanos do que ser barata para o computador.

Há, entretanto, boas razões para ser cauteloso. Uma é o custo da complexidade do estágio de análise – você não quer aumentá-lo ao ponto onde é uma fonte significativa de erros e confusão de usuários. Outra é que tentar fazer a sintaxe de uma linguagem *English-like* frequentemente demanda que o "inglês" que é falado seja seriamente propenso a ser fora de forma, tanto como a semelhança superficial com a linguagem natural é tão confusa como a sintaxe tradicional seria (você vê isso em várias linguagens chamadas de "quarta geração" e em linguagens de consulta de banco de dados comerciais).

A sintaxe de controle do *fetchmail* parece evitar estes problemas porque o domínio da linguagem é extremamente restrito. Não é perto de uma linguagem de uso geral; as coisas que ela diz simplesmente não são muito complicadas, então há pouco potencial para confusão em mover mentalmente entre um pequeno conjunto do inglês e a real linguagem de controle. Eu acho que há uma lição mais abrangente aqui:

16. Quando sua linguagem não está perto de um *Turing* completo, açúcar sintático pode ser seu amigo.

Outra lição é sobre segurança por obscuridade. Alguns usuários do *fetchmail* me pediram para mudar o *software* para guardar as senhas encriptadas no arquivo rc, de maneira que bisbilhoteiros não poderiam casualmente vê-las.

Eu não fiz isso, porque isto não adiciona realmente proteção. Qualquer pessoa que adquira permissões para ler seu arquivo rc poderá executar o *fetchmail* como você de qualquer maneira – e se é a sua senha o que eles

procuram, poderiam retirar o decodificador do próprio *fetchmail* para conseguila.

Tudo o que a encriptação de senha do arquivo *.fetchmailrc* faria seria dar a falsa impressão de segurança para pessoas que não pensaram bem sobre este assunto. Aqui a regra geral é:

17. Um sistema de segurança é tão seguro quanto é secreto. Esteja atento a pseudo-segredos.

9. Pré-condições Necessárias para o Estilo Bazar

Os primeiros leitores e audiências para este documento seguidamente levantaram questões sobre as pré-condições para o desenvolvimento bem-sucedido do estilo bazar, incluindo as qualificações do líder do projeto e o estado do código ao tempo em que se torna público e começa a tentar construir uma comunidade de co-desenvolvedores.

Está claro que ninguém pode codificar desde o início no estilo bazar. Alguém pode testar, achar erros e aperfeiçoar no estilo bazar, mas seria muito difícil originar um projeto no estilo bazar. Linus não tentou isto. Eu também não. A sua comunidade nascente de desenvolvedores precisa ter algo executável e passível de testes para utilizar.

Quando você começa a construção de uma comunidade, o que você precisa ter capacidade de apresentar é uma promessa plausível. Seu programa não precisa funcionar particularmente bem. Ele pode ser grosseiro, cheio de erros, incompleto, e pobremente documentado. O que não pode deixar de fazer é convencer co-desenvolvedores em potencial de que ele pode evoluir para algo realmente elegante em um futuro próximo.

Tanto o *Linux* quanto o *fetchmail* se tornaram públicos com projetos básicos fortes e atraentes. Muitas pessoas pensando sobre o modelo bazar como eu tenho apresentado têm corretamente considerado isto como crítico, então concluíram a partir disso que um alto grau de intuição e inteligência no líder do projeto é indispensável.

Mas Linus obteve seu plano do *Unix*. Eu obtive o meu inicialmente do ancestral do popclient (embora iria posteriormente mudar bastante, muito mais proporcionalmente falando do que mudou o *Linux*). Então

é necessário realmente para o líder/coordenador de um empenho no estilo bazar ter um talento excepcional para planejamento, ou ele pode conseguir o mesmo efeito coordenando o talento de planejamento de outras pessoas?

Eu penso que não é crítico que o coordenador seja capaz de originar projetos de excepcional brilho, mas é absolutamente crítico que o coordenador seja capaz de reconhecer boas idéias de projetos de outras pessoas.

Tanto os projetos do *Linux* quanto o do *fetchmail* mostram evidências disso. Linus, não sendo (como previamente discutido) um planejador espetacularmente original, mostrou uma habilidade poderosa de reconhecer um bom projeto e integrá-lo no kernel do *Linux*. E eu já descrevi como a idéia de projeto mais poderosa do *fetchmail* (reenvio por SMTP) veio de outra pessoa.

Os primeiros leitores deste documento me elogiaram sugerindo que eu sou propenso a subestimar a originalidade do planejamento de projetos no estilo bazar porque eu tenho muito disso em mim mesmo, e portanto suponho isto. Pode haver alguma verdade nisto; projetar (como oposto a codificar ou depurar) é certamente minha mais forte habilidade.

Mas o problema em ser inteligente e original no planejamento de *software* é que isto se torna um hábito – você começa reflexivamente a fazer coisas atraentes e complicadas quando você deveria mantê-las robutas e simples. Eu tive projetos que falharam porque eu cometi este erro, mas eu administrei para que não acontecesse o mesmo com o *fetchmail*.

Então eu acredito que o projeto do *fetchmail* foi um sucesso em parte porque eu impedi a minha tendência a ser engenhoso; isto vai contra (pelo menos) com o fato de a originalidade em projetar ser essencial para projetos bem-sucedidos no estilo bazar. E considere o *Linux*. Suponha que Linus Torvalds estivesse tentando levar a cabo inovações fundamentais no projeto de

sistemas operacionais durante o desenvolvimento; pareceria que o kernel resultante seria tão estável e bem-sucedido como é?

Um certo nível básico de habilidade de projetar e codificar é requerido, claro, mas eu suponho que praticamente qualquer um que pense seriamente em lançar um esforço no estilo bazar estará acima do mínimo necessário. O mercado interno da comunidade de *software* aberto, por reputação, exerce uma sutil pressão nas pessoas para que não lancem esforço de desenvolvimento que não sejam competentes para manter. Até agora isto parece ter funcionado muito bem.

Há outro tipo de habilidade que não é normalmente associada com o desenvolvimento de *software* que eu penso é tão importante quanto a engenhosidade em planejar projetos no estilo bazar – e isto pode ser mais importante. Um coorrdenador ou líder de um projeto no estilo bazar deve ter boa habilidade de comunicação e relacionamento.

Isto deve parecer óbvio. Para construir uma comunidade de desenvolvimento, você precisa atrair pessoas, fazer com que se interessem no que você está fazendo, e mantê-las alegres sobre a quantidade de trabalho que estão fazendo. O entusiasmo técnico constitui uma boa parte para atingir isto, mas está longe de ser toda história. A personalidade que você projeta também importa.

Não é uma coincidência que Linus é um rapaz gentil que faz com que as pessoas gostem dele e que o ajudem. Não é uma coincidência que eu seja um enérgico extrovertido que gosta de trabalhar com pessoas e tenha um pouco de porte e instinto de um cômico. Para fazer o modelo bazar funcionar, isto ajuda enormemente se você tem pelo menos um pouco de habilidade para encantar as pessoas.

10. O Contexto Social do Código Aberto

Está escrito: os melhores hacks começam como soluções pessoais para os problemas diários do autor e se espalham porque o problema se torna típico para uma grande classe de usuários. Isto nos traz novamente para a regra 1, recolocada talvez de uma maneira mais útil:

18. Para resolver um problema interessante, comece achando um problema que é interessante para você.

Isso foi o que aconteceu com Carl Harris e o ancestral *popclient*, e também comigo e o *fetchmail*. Mas isso foi entendido por um longo tempo. O ponto interessante, o ponto que as histórias do *Linux* e do *fetchmail* parecem demandar o nosso foco é o próximo estágio – a evolução do *software* na presença de uma comunidade grande e ativa de co-desenvolvedores.

Em "*The Mythical Man-Month*", Fred Brooks observou que o tempo de um programador não é acumulativo; adicionar desenvolvedores em um projeto atrasado de *software* faz com que ele se torne mais atrasado. Ele expôs que a complexidade e custos de comunicação de um projeto crescem com o quadrado do número de desenvolvedores, enquanto o trabalho feito cresce somente linearmente. Esta afirmação é desde então conhecida como a "Lei de Brooks" e é largamente considerada como correta. Mas se a Lei de Brooks fosse tudo a ser levado em consideração, o *Linux* seria impossível.

O clássico "*The Psychology of Computer Programming*", de Gerald Weinberg, sustentou o que, em retrospectiva, nós podemos ver como uma correção essencial ao Brooks. Em sua discussão sobre "programação sem ego", Weinberg observou que nos lugares onde desenvolvedores não são territoriais sobre seu código e encorajam outras pessoas a procurar por erros e melhorias potenciais nele, as melhorias acontecem dramaticamente mais rápidas que em

qualquer outro lugar.

A escolha da terminologia de Weinberg talvez tenha prevenido sua análise de ganhar a aceitação que merecia – é engraçado pensar em descrever os hackers da Internet como “sem ego”. Mas acho que seu argumento parece mais mais convincente hoje do que nunca.

A história do *Unix* deveria ter sido preparada para nós do que nós estamos aprendendo com o *Linux* (e o que eu tenho verificado experimentalmente em uma menor escala por copiar deliberadamente os métodos do Linus). Isto é, embora codificar permaneça uma atividade essencialmente solitária, os hacks realmente bons advém de captar a atenção e poder de mente de comunidades inteiras. O desenvolvedor que utiliza apenas a capacidade cerebral dele mesmo em um projeto fechado irá ficar atrás de desenvolvedores que saibam como criar um contexto aberto e evolutivo no qual a visualização de erros e melhorias sejam feitas por centenas de pessoas.

Mas o mundo tradicional do *Unix* foi impedido de seguir completamente este método por vários fatores. Alguns deles foram os aspectos legais de várias licenças, segredos de comércio e interesses comerciais. Outro (tardio) foi que a Internet ainda não estava boa o suficiente.

Antes de a Internet baratear, havia algumas comunidades geograficamente pequenas aonde a cultura motivava a programação “sem ego” de Weinberg e aonde um desenvolvedor poderia facilmente atrair muitos co-desenvolvedores habilidosos. Bell Labs, o MIT AI Lab, UC Berkeley – estes se tornaram a origem de inovações que são lendárias e ainda fortes.

Linux foi o primeiro projeto a fazer um esforço consciente e bem-sucedido a utilizar o mundo inteiro como sua reserva de talentos. Eu não acho que seja uma coincidência que o período de gestação do *Linux* tenha coincidido

com o nascimento da *World Wide Web* e que o *Linux* tenha deixado a sua infância durante o mesmo período em 1993-1994 que viu a expansão da indústria de ISP e a explosão do principal interesse da Internet. Linus foi a primeira pessoa que aprendeu como jogar com as novas regras que a onipresente Internet fez possível.

Embora uma Internet barata fosse uma condição necessária para que o modelo do *Linux* evoluísse, eu penso que não foi uma condição por si só suficiente. Outro fator vital foi o desenvolvimento de um estilo de liderança e conjunto de formalidades cooperativas que permitiria aos desenvolvedores atrair co-desenvolvedores e obter o máximo suporte do ambiente.

Mas o que é este estilo de liderança e estas formalidades? Eles não podem estar baseados em relações de poder – e mesmo que pudessem, a liderança por coerção não produziria os resultados que nós vemos. Weinberg cita a autobiografia do anarquista do século 19 chamado Pyotr Alexeyvich Kropotkin, *"Memórias de um Revolucionista"* para demonstrar o efeito neste assunto:

> *"Tendo sido criado em uma família possuidora de vassalos, eu entrei à vida ativa, como todos os jovens homens da minha idade, com uma grande confidência na necessidade de comandar, ordenar, repreender, punir e etc. Mas quando cedo tive que conduzir empreendimentos sérios e lidar com homens [livres] e quando cada erro levaria de uma vez a sérias consequências, eu comecei a apreciar a diferença entre atuar segundo o princípio de comando e disciplina e atuar segundo o princípio da compreensão comum. O primeiro funciona de forma admirável em uma parada militar, mas de nada vale aonde a vida real é considerada e o objetivo pode ser atingido somente pelo esforço severo de muitos propósitos*

convergentes.”

O *“esforço severo de muitos propósitos convergentes”* é precisamente o que um projeto como o *Linux* requer – e o *“princípio de comando”* é efetivamente impossível de ser aplicado entre voluntários no paraíso anarquista que nós chamamos de Internet. Para operar e competir efetivamente, hackers que querem liderar projetos colaborativos têm que aprender como recrutar e energizar comunidades eficazes com interesse no modo vagamente sugerido pelo *“princípio de compreensão”* sugerido por Kropotkin. Eles precisam aprender a Lei de Linus.

Inicialmente eu me referi ao Efeito Delphi como uma possível explicação para a Lei de Linus. Porém, analogias mais poderosas aos sistemas adaptativos em biologia e economia também se implicam fortemente. O mundo do *Linux* se comporta em vários aspectos como um mercado livre ou uma ecologia, uma coleção de agentes autônomos tentando maximizar um empreendimento que no processo produz uma ordem espontânea auto-evolutiva mais elaborada e eficiente que qualquer quantidade de planejamento central poderia ter alcançado. Aqui, então, é o lugar para procurar o *“princípio da compreensão”*.

A *“função empreendedora”* que os hackers do *Linux* estão maximizando não é economia clássica, mas é a intangível satisfação do seu próprio ego e reputação entre outros hackers – Alguém pode chamar a sua motivação de “altruísta”, mas isso ignora o fato que altruísmo é em si mesmo uma forma de satisfação do ego para um altruísta. Culturas voluntárias que trabalham desta maneira não são realmente incomuns; uma outra que eu tenho participado há tempos é os fãs de ficção científica, que ao contrário dos hackers, explicitamente reconhecem o “egoboo” (o aumento da reputação de alguém entre os outros fãs) como o guia básico por trás da atividade voluntária.

Linus, por posicionar ele mesmo como o guia de um projeto em que o desenvolvimento é praticamente feito por outros, e por inspirar interesse no projeto até que ele se tornasse auto-sustentável, tem mostrado um intenso entendimento do "princípio da compreensão comum" de Kropotkin. Esta visão quase-econômica do mundo do *Linux* nos permite ver como esta compreensão é aplicada.

Nós podemos ver o método do Linus como uma maneira de criar um mercado eficiente no "egoboo" – para ligar a autonomia de hackers individuais tão firme quanto possível para dificultar fins que podem ser somente atingidos por uma cooperação sustentada. Com o projeto do *fetchmail* eu tenho mostrado (embora em uma menor escala) que seus métodos podem ser duplicados com bons resultados. Talvez eu tenha até mesmo feito de uma maneira um pouco mais consciente e sistemática do que ele.

Muitas pessoas (especialmente aquelas que politicamente desacreditam os mercados livres) poderiam esperar de uma cultura de egoístas auto-direcionadores ser fragmentada, territorial, desperdiçadora, reservada e hostil. Mas esta expectativa é claramente desfeita pela (para dar só um exemplo) imensa variedade, qualidade e profundidade da documentação do *Linux*. É notoriamente conhecido que os programadores detestam documentar; como, então, os hackers do *Linux* geram tanto disto? Evidentemente o mercado livre do *Linux* em egoboo trabalha melhor para produzir comportamentos virtuosos e direcionados a outros propósitos que os centros de documentação massivamente fundados de produtores de *software* comercial.

Tanto o projeto do *fetchmail* como o do *kernel* do *Linux* mostram que ao se recompensar propriamente o ego de muitos outros hackers, um desenvolvedor/coordenador forte pode usar a Internet para capturar os benefícios de se ter vários co-desenvolvedores sem fazer o projeto colapsar em uma confusão caótica. Então eu contra-proponho para a Lei de Brooks o

seguinte:

19. Contanto que o coodenador do desenvolvimento tenha uma mídia pelo menos tão boa quanto a Internet, e saiba como liderar sem coerção, muitas cabeças são inevitavelmente melhores que uma.

Eu acredito que o futuro do *software* de código aberto irá pertencer gradativamente a pessoas que saibam como jogar o jogo do Linus, pessoas que deixam para trás a catedral e abraçam o bazar. Isto não quer dizer que uma visão individual e brilhante não irá mais ter importância; ao contrário, eu acredito que o estado da arte do *software* de código aberto irá pertencer a pessoas que iniciem de uma visão individual e brilhante, então amplificando-a através da construção efetiva de uma comunidade voluntária de interesse.

E talvez não somente o futuro do *software* de código aberto. Nenhum desenvolvedor de código fechado pode competir com o conjunto de talento que a comunidade do *Linux* pode dar para resolver um problema. Muito poucos podem até mesmo ser capazes de pagar as mais de duzentas pessoas que têm contribuído para o *fetchmail*!

Talvez no final a cultura de código aberto irá triunfar não porque a cooperação é moralmente correta ou a "proteção" do *software* é moralmente errada (assumindo que você acredita na última, o que não faz tanto o Linus como eu), mas simplesmente porque o mundo do *software* de código fechado não pode vencer uma corrida evolucionária com as comunidades de código aberto que podem colocar mais tempo hábil ordens de magnitude acima em um problema.

11. Agradecimentos

Este documento foi enriquecido através de conversas com um grande número de pessoas que ajudaram a depurá-lo. Agradecimentos particulares a Jeff Dutky <dutky@wam.umd.edu>, que sugeriu a formulação "depuração é paralelizável" e ajudou a desenvolver a análise que se procedeu a isso. E também a Nancy Lebovitz <nancyl@universe.digex.net> pela sua sugestão que eu imitei Weinberg ao citar Kropotkin. Críticas incisivas também vieram de Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> e Marty Franz <marty@net-link.net> da lista de Técnicas Gerais. Sou grato aos membros do PLUG, o Grupo de Usuários de *Linux* da Filadélfia, por permitir a primeira audiência de teste para a primeira versão pública deste documento. Finalmente, os comentários de Linus Torvalds foram importantes e seu apoio desde o início foi muito estimulante.

12. Para Leitura Adicional

Eu citei várias frases do clássico *"The Mythical Man-Month"* de Frederick P. Brook porque, em muitos aspectos, suas considerações ainda precisam ser mais analisadas. Eu recomendo fortemente a 25a. edição de aniversário de Addison-Wesley (ISBN 0-201-83595-9), que adiciona seu documento *"No Silver Bullet"* de 1986.

A nova edição é envolvida por uma inestimável retrospectiva de 20 anos atrasada na qual Brooks admite francamente os poucos julgamentos no texto original que não resistiram ao teste do tempo. Achei que a retrospectiva deste documento estava substancialmente completa e fiquei surpreso ao descobrir que Brooks atribui práticas como o estilo bazar à *Microsoft*! (De fato, entretanto, esta atribuição se mostrou errada. Em 1998 nós entendemos dos *Documentos Halloween* que a comunidade interna de desenvolvimento da *Microsoft* é estratificada, com o tipo comum de acesso ao código necessário para suportar o bazar muitas vezes nem mesmo possível.)

The Psychology of Computer Programming, de Gerald M. Weinberg (*New York, Van Nostrand Reinhold* 1971) introduziu o conceito um tanto mal nomeado de "programação sem ego". Embora ele não estivesse nem perto de ser a primeira pessoa a perceber a futilidade do *"princípio do comando"*, ele foi provavelmente o primeiro a reconhecer e argumentar o ponto particular de ligação com o desenvolvimento de *software*.

Richard P. Gabriel, contemplando a cultura *Unix* da era pré-*Linux*, relutantemente argumentou a favor da superioridade do modelo primitivo do estilo bazar em seu documento *Lisp: Good News, Bad News, and How to Win Big*, de 1989. Embora obsoleto em alguns aspectos, este ensaio ainda é muito cotado entre os fãs do *Lisp* (incluindo eu). Um correspondente me lembrou que

a seção intitulada *"O Pior é o Melhor"* parece-se mais como uma antecipação do *Linux*. O documento está disponível na *World Wide Web* em http://www.naggum.no/worse-is-better.html.

Peopleware: Productive Projects and Teams (*New York*; *Dorset House*, 1987; ISBN 0-932633-05-6), de De Marco e Lister, é uma jóia pouco apreciada na qual fiquei deliciado em ver Fred Brooks citá-la em sua retrospectiva. Embora pouco do que os autores têm a dizer é diretamente aplicável às comunidades do código aberto do *Linux*, as idéias do autor sobre as condições necessárias para um trabalho criativo é incisivo e válido para qualquer um que tente importar um pouco das virtudes do modelo bazar para o contexto comercial.

Finalmente, eu devo admitir que eu quase chamei este documento de *"A Catedral e a Ágora"*, o último termo sendo o grego para um mercado aberto ou um lugar de encontro público. Os documentos "sistemas agóricos" de Mark Miller e Eric Drexler, descrevendo as propriedades emergentes de ecologias computacionais de mercado, me ajudaram a me preparar a pensar claramente sobre fenômenos análogos na cultura do código aberto quando o *Linux* esfregou o meu nariz neles cinco anos depois. Estes documentos estão disponíveis na Web em http://www.agorics.com/agorpapers.html.

13. Epílogo: *Netscape* Acata o Bazar!

É um sentimento estranho perceber que você está ajudando a fazer história...

Em 22 de janeiro de 1998, aproximadamente sete meses depois que eu publiquei pela primeira vez este documento, *Netscape Communications, Inc.* anunciou planos para liberar o código do *Netscape Communicator*. Eu não tive nenhuma idéia de que isso poderia acontecer até o dia do anunciamento.

Eric Hahn, Vice Presidente Executivo e Oficial Chefe de Tecnologia da *Netscape*, enviou uma mensagem para mim pouco tempo depois como se segue:

> *"Em nome de todos da Netscape, eu quero agradecer a você por ter nos ajudado chegar a este ponto em primeiro lugar. Seus pensamentos e escritos foram inspirações fundamentais para nossa decisão."*

Na semana seguinte eu voei até o Vale do Silício a pedido da *Netscape* para uma conferência estratégica de um dia (em 4 de fevereiro de 1998) com alguns de seus executivos de alto escalão e técnicos. Nós desenhamos juntos a estratégia de lançamento da *Netscape* para o código fonte e licenciamento, e deixamos mais alguns planos que nós esperamos irá eventualmente ter impactos mais abrangentes e positivos na comunidade do código aberto. Enquanto escrevo, ainda é um pouco cedo demais para ser mais específico; porém detalhes devem se tornar conhecidos em semanas.

A *Netscape* está para nos fornecer um real teste em larga escala do modelo bazar no mundo comercial. A cultura do código aberto agora enfrenta um risco; se os planos da *Netscape* não funcionarem, o conceito de código

aberto pode se tornar tão desacreditado que o mundo comercial não irá tocar nele de novo por uma década.

Por outro lado, isto também é uma oportunidade espetacular. Reações iniciais ao movimento em Wall Street e em outros lugares foram cautelosamente positivas. Nós também estamos tendo a chance para nos provar. Se a *Netscape* retomar uma substancial fatia do mercado através deste movimento, ela poderá iniciar uma revolução tardia na indústria de *software*.

O próximo ano deverá ser muito instrutivo e interessante.

14. Versão e Histórico de Mudanças

```
$Id: cathedral-bazaar.sgml,v 1.42 1998/11/22 04:01:20 esr Exp $
```

- Eu liberei a versão 1.16 no *Linux Kongress*, em 21 de maio de 1997.
- Eu adicionei a bibliografia em 7 de julho de 1997 na versão 1.20.
- Eu adicionei a anedota da *Perl Conference* em 18 de novembro de 1997 na versão 1.27.
- Eu mudei de "*software* livre" para "código aberto" em 9 de fevereiro de 1998 na versão 1.29.
- Eu adicionei *"Epílogo: Netscape Acata o Bazar!"* em 10 de fevereiro de 1998 na versão 1.31.
- Eu removi o gráfico de Paul Eggert sobre GPL vs. bazar em resposta aos argumentos coerentes de RMS em 28 de julho de 1998.
- Eu adicionei uma correção de *Brooks* baseado nos *Documentos Halloween* em 20 de novembro de 1998 na versão 1.40.
- Outros níveis de revisões incorporaram pequenas correções editoriais.