La cattedrale e il bazaar

di Eric S. Raymond
(22/11/1998 ore 04:01:20)

Quella che segue è la mia analisi di un progetto open source di successo, fetchmail, deliberatamente utilizzato come test specifico per la verifica di alcune sorprendenti teorie sullo sviluppo del software suggerite dalla storia di Linux. Le mie argomentazioni su tali teorie mettono a confronto due diversi stili di sviluppo, il modello “cattedrale” in voga in gran parte del mondo commerciale, opposto al modello “bazaar” del mondo Linux. Da qui passo poi a dimostrare come tali modelli derivino da premesse divergenti sulla natura dell'attività di debugging del software. Arrivo quindi a stabilire la validità dell'esperienza di Linux riguardo l'affermazione “Con molti occhi puntati addosso, ogni bug diventa una bazzecola”, per suggerire analogie produttive con altri sistemi di agenti indipendenti in grado di auto-correggersi, concludendo infine con una serie di riflessioni sulle implicazioni di queste analisi per il futuro del software.

Sommario:

1. La cattedrale e il bazaar

Linux è sovversivo. Chi avrebbe potuto pensare appena cinque anni fa che un sistema operativo di livello mondiale sarebbe emerso come per magia dal lavoro part-time di diverse migliaia di hacker e sviluppatori sparsi sull'intero pianeta, collegati tra loro solo grazie ai tenui cavi di Internet?
Certamente non il sottoscritto. Quando Linux fece la sua comparsa nel mio raggio d'azione all'inizio del 1993, mi ero occupato dello sviluppo di Unix e di software open source per dieci anni. Ero stato uno dei primi collaboratori al progetto GNU a metà anni '80. Avevo distribuito su Internet un buon numero di software open source, realizzando da solo o in collaborazione con altri parecchi programmi (nethack, Emacs VC e GUD, xlife, etc.) ancor'oggi ampiamente utilizzati. Pensavo di sapere come bisognasse fare.
Linux stravolse gran parte di quel che credevo di sapere. Per anni avevo predicato il vangelo Unix degli strumenti agili, dei prototipi immediati e della programmazione evolutiva. Ma ero anche convinto che esistesse un punto critico di complessità al di sopra del quale si rendesse necessario un approccio centralizzato e a priori. Credevo che il software più importante (sistemi operativi e strumenti davvero ingombranti come Emacs) andasse realizzato come le cattedrali, attentamente lavorato a mano da singoli geni o piccole bande di maghi che lavoravano in splendido isolamento, senza che alcuna versione beta vedesse la luce prima del momento giusto.
Rimasi non poco sorpreso dallo stile di sviluppo proprio di Linus Torvalds – diffondere le release presto e spesso, delegare ad altri tutto il possibile, essere aperti fino alla promiscuità. Nessuna cattedrale da costruire in silenzio e reverenza. Piuttosto, la comunità Linux assomigliava a un grande e confusionario bazaar, pullulante di progetti e approcci tra loro diversi (efficacemente simbolizzati dai siti contenenti l'archivio di Linux dove apparivano materiali prodotti da chiunque). Un bazaar dal quale soltanto una serie di miracoli avrebbe potuto far emergere un sistema stabile e coerente.
Il fatto che questo stile bazaar sembrasse funzionare, e anche piuttosto bene, mi colpì come uno shock. Mentre imparavo a prenderne le misure, lavoravo sodo non soltanto sui singoli progetti, ma anche cercando di comprendere come mai il mondo Linux non soltanto non cadesse preda della confusione più totale, ma al contrario andasse rafforzandosi sempre più a una velocità a malapena immaginabile per quanti costruivano cattedrali.
Fu verso la metà del 1996 che mi parve d'essere sul punto di capirne il perché. Il destino mi offrì l'occasione propizia per mettere alla prova la mia teoria, sotto forma di un progetto open source del quale decisi di occuparmi usando coscientemente lo stile bazaar. Ci provai, e il successo ottenuto fu piuttosto significativo.
Nella parte restante di questo saggio, racconto la storia di quel progetto, usandola per proporre alcuni aforismi sull'efficacia dello sviluppo open source. Non che li abbia imparati tutti dal mondo Linux, ma vedremo come le modalità offerte da quest'ultimo siano del tutto peculiari. Se non ho interpretato male, questi aforismi ci aiuteranno a comprendere con esattezza cos'è che rende la comunità Linux una sorgente così copiosa di buon software – e aiuteranno tutti noi a divenire più produttivi.

2. La posta deve passare

Dal 1993 mi occupo del lato tecnico di un piccolo provider Internet gratuito chiamato Chester County InterLink (CCIL) in West Chester, Pennsylvania (sono tra i fondatori di CCIL e autore del software specifico per il nostro bulletin-board multiutenti – si può dare un'occhiata facendo telnet su locke.ccil.org . Ora dà accesso a quasi tremila utenti su trenta linee). Grazie a questo lavoro posso collegarmi a Internet per 24 ore al giorno con una linea a 56K di CCIL – in realtà, è proprio quel che mi viene richiesto!
Di conseguenza sono ormai abituato alle email istantanee. Per vari motivi, era difficile far funzionare la connessione SLIP tra la mia macchina a casa (snark.thyrsus.com) e CCIL. Quando finalmente ci sono riuscito, mi dava fastidio dover fare ogni tanto telnet su locke per controllare la posta. Volevo fare in modo che i messaggi arrivassero direttamente su snark così da esserne tempestivamente avvisato e poterli gestire a livello locale.
Il semplice “sendmail forwarding” non avrebbe funzionato, perché la mia macchina personale non è sempre online e non ha un indirizzo IP statico. Mi serviva un programma in grado di raggiungere la connessione SLIP e tirar via la posta per farla arrivare localmente. Sapevo dell'esistenza di simili cose, e

del fatto che in genere facevano uso di un semplice protocollo noto come POP (Post Office Protocol). E sicuramente doveva già esserci un server POP3 incluso nel sistema operativo BSD/OS di locke.
Mi serviva un client POP3. Ne ho localizzato subito uno online. Anzi, ne ho trovati tre o quattro. Per un po' ho usato un pop-perl, ma era privo di quella che pareva una funzione ovvia, la capacità di effettuare un "hacking degli indirizzi della posta prelevata in modo che il reply funzionasse correttamente.
Questo il problema: supponiamo di ricevere un messaggio da qualcuno di nome 'joe' su locke. Se lo inoltro su snark e poi cerco di fare reply, il mio programma di posta proverebbe simpaticamente a inviarlo a un inesistente 'joe' su snark. Modificare a mano ogni indirizzo per aggiungere "@ccil.org" diventerebbe in un attimo un problema serio.
Chiaramente questa era un'operazione che toccava fare al computer per conto mio. Ma nessuno dei client POP esistenti sapeva come! E questo ci porta alla prima lezione:
1. Ogni buon lavoro software inizia dalla frenesia personale di uno sviluppatore.
Forse ciò avrebbe dovuto risultare ovvio (è risaputo da tempo che "la necessità è la madre di tutte le invenzioni"), ma troppo spesso gli sviluppatori trascorrono le giornate impegnati a guadagnarsi da vivere con programmi di cui non hanno alcun bisogno e che non apprezzano. Ma non nel mondo Linux – il che spiega l'alta qualità media del software originato dalla comunità Linux.
Mi sono forse lanciato in un'attività frenetica per scrivere il codice di un client POP3 nuovo di zecca in grado di competere con quelli esistenti? Nemmeno per sogno! Ho esaminato attentamente le utility POP che avevo in mano, chiedendomi: "qual'è la più vicina a quel che sto cercando?" Perché:
2. I bravi programmatori sanno cosa scrivere. I migliori sanno cosa riscrivere (e riusare).
Pur non ritenendomi un programmatore tra i più bravi, cerco di imitarli. Importante caratteristica di costoro è una sorta di ozio costruttivo. Sanno che si ottiene il meglio non per le energie impiegate ma per il risultato raggiunto, e che quasi sempre è più facile iniziare da una buona soluzione parziale piuttosto che dal nulla assoluto.
Linus Torvalds, per esempio, non ha mai cercato di riscrivere Linux da zero. È invece partito riutilizzando codici e idee riprese da Minix, piccolo sistema operativo per macchine 386 assai simile a Unix. Alla fine il codice Minix è scomparso oppure è stato completamente riscritto – ma per il tempo che è rimasto lì presente è servito come impalcatura per l'infante che sarebbe infine divenuto Linux.
Con lo stesso spirito, mi sono messo a cercare una utility POP basata su codici ragionevolmente ben fatti, da utilizzare come base di sviluppo.
La tradizione di condivisione dei codici tipica del mondo Unix ha sempre favorito il riutilizzo dei sorgenti (questo il motivo per cui il progetto GNU ha scelto come sistema operativo di base proprio Unix, nonostante alcune serie riserve sullo stesso). Il mondo Linux ha spinto questa tradizione vicina al suo al limite tecnologico; sono generalmente disponibili terabyte di codice open source. È quindi probabile che, impiegando del tempo a cercare il lavoro di qualcuno quasi ben fatto, si ottengano i risultati voluti. E ciò vale assai più nel mondo Linux che altrove.
Proprio quel che è successo a me. Conteggiando i programmi trovati prima, con la seconda ricerca ottenni un totale di nove candidati – fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail e upop. Il primo su cui mi sono concentrato è stato 'fetchpop' di Seung-Hong Oh. Ho inserito l'opzione "header-rewrite" e ho apportato altri miglioramenti, poi accettati dall'autore nella release 1.9.
Alcune settimane dopo, però, mi sono imbattuto nel codice di 'popclient' scritto da Carl Harris, ed mi sono trovato di fronte a un problema. Pur offrendo alcune buone idee originali (come la modalità "daemon"), fetchpop poteva gestire solo POP3 e il codice rifletteva un certo approccio da dilettante (Seung-Hong era un programmatore brillante ma inesperto, ed entrambe le qualità risultavano evidenti). Il codice di Carl era migliore, alquanto professionale e solido, ma il suo programma difettava di varie opzioni presenti in fetchpop, opzioni importanti e piuttosto complesse da implementare (incluse quelle aggiunte dal sottoscritto).
Restare o cambiare? Nel secondo caso avrei buttato via il codice che avevo già scritto in cambio di una migliore base di sviluppo.
Un motivo pratico per passare all'altro programma era il supporto per protocolli multipli. POP3 è il più usato tra i server per l'ufficio postale, ma non è il solo. Fetchpop e l'altro rivale non avevano POP2, RPOP, o APOP, e io stesso stavo meditando, giusto per divertimento, l'aggiunta di IMAP (Internet Message Access Protocol, il protocollo per l'ufficio postale più recente e più potente).
Avevo però altri motivi teorici per ritenere una buona idea il fatto di cambiare, qualcosa che avevo imparato molto tempo prima di Linux.
3. "Preparati a buttarne via uno; dovrai farlo comunque." (Fred Brooks, "The Mythical Man-Month", Capitolo 11)

In altri termini, spesso non si riesce a comprendere davvero un problema fino alla prima volta in cui si prova a implementarne la soluzione. La seconda volta forse se ne sa abbastanza per riuscirci. Per arrivare alla soluzione, preparati a ricominciare almeno una volta.
Be', mi son detto, la mia prima volta erano state le modifiche a fetchpop. Adesso era ora di cambiare, e così feci.
Dopo aver mandato a Carl Harris il 25 Giugno 1996 una prima serie di aggiustamenti per popclient, mi resi conto che da qualche tempo egli aveva perso interesse nel programma. Il codice era un po' polveroso, con vari bug in giro. Avrei dovuto fare molte modifiche, e ci mettemmo rapidamente d'accordo sul fatto che la cosa più logica fosse che il programma passasse in mano mia.
Senza che me ne accorgessi più di tanto, il progetto era cresciuto parecchio. Non mi stavo più occupando soltanto di sistemare i piccoli difetti di un client POP già esistente. Mi ero addossato l'intera gestione di un programma, e mi venivano in mente delle idee che avrebbero probabilmente portato a modifiche radicali.
In una cultura del software che incoraggia la condivisione del codice, non si trattava altro che della naturale evoluzione di un progetto. Questi i punti-chiave:
4. Se hai l'atteggiamento giusto, saranno i problemi interessanti a trovare te.
Ma l'atteggiamento di Carl Harris risultò perfino più importante. Fu lui a comprendere che:
5. Quando hai perso interesse in un programma, l'ultimo tuo dovere è passarlo a un successore competente.
Senza neppure parlarne, io e Carl sapevamo di perseguire il comune obiettivo di voler raggiungere la soluzione migliore. L'unica questione per entrambi era stabilire se le mie fossero mani fidate. Una volta concordato su questo, egli agì con gentilezza e prontezza. Spero di comportarmi altrettanto bene quando verrà il mio turno.

3. L'importanza di avere utenti

E così ho ereditato popclient. Fatto parimenti importante, ho ereditato gli utenti di popclient. Importante non soltanto perché la loro esistenza testimonia che stai rispondendo a un loro bisogno, che hai fatto qualcosa di buono. Coltivati in maniera appropriata, gli utenti possono trasformarsi in co-sviluppatori. Altro punto di forza della tradizione Unix, portato felicemente agli estremi da Linux, è che molti utenti sono essi stessi degli hacker. Ed essendo i sorgenti disponibili a tutti, posso diventare degli hacker molto efficaci. Qualcosa di tremendamente utile per ridurre il tempo necessario al debugging. Con un po' d'incoraggiamento, ogni utente è in grado di diagnosticare problemi, suggerire soluzioni, aiutare a migliorare il codice in maniera impensabile per una persona sola.
6. Trattare gli utenti come co-sviluppatori è la strada migliore per ottenere rapidi miglioramenti del codice e debugging efficace.
È facile sottovalutare la potenza di un simile effetto. In realtà un po' tutti noi del mondo open source eravamo soliti sottovalutare drasticamente il fatto che tale potenza crescesse di pari passo con il numero degli utenti e con la complessità del sistema. Finché Linus Torvalds ci ha mostrato le cose in maniera diversa.
In realtà ritengo che la mossa più scaltra e consequenziale di Linus non sia stata la costruzione del kernel di Linux in sé, bensì la sua invenzione del modello di sviluppo di Linux. Quando ho espresso questo mio pensiero in sua presenza, sorridendo ha ripetuto con calma quel che va spesso affermando: "Praticamente sono una persona molto pigra cui piace prendersi il merito di quel che sono gli altri a fare." Pigro come una volpe. Oppure, come avrebbe detto Robert Heinlein, troppo pigro per fallire.
Guardando all'indietro, un precedente per i metodi e il successo di Linux può esser trovato nello sviluppo della libreria Lisp GNU e per gli archivi del codice Lisp di Emacs. In opposizione allo stile di costruzione a cattedrale del nucleo centrale in C di Emacs e di gran parte di altri strumenti della Free Software Foundation (FSF), l'evoluzione del codice Lisp risultò assai fluida e guidata dagli utenti. Idee e prototipi vennero spesso riscritti tre o quattro volte prima di raggiungere una forma stabile e definitiva. E le collaborazioni estemporanee tra due o più persone consentite da Internet, alla Linux, erano evento frequente.
Non a caso il mio caso di "hack" precedente a fetchmail fu probabilmente la modalità Emacs VC, una collaborazione via email secondo lo stile Linux con altre tre persone, soltanto una delle quali (Richard Stallman, autore di Emacs e fondatore della FSF http://www.fsf.org/) ho poi avuto occasione di incontrare dal vivo. Si trattava di realizzare il frontale per SCCS, RCS e più tardi CVS dall'interno di Emacs, dotato di opzioni a "one-touch" per le operazioni di controllo. Ciò ha preso avvio da un minuto, crudo sccs.el

scritto da qualcun altro. E lo sviluppo di VC ha avuto successo perché, al contrario dello stesso Emacs, il codice di Lisp riuscì a passare molto rapidamente tra diverse generazioni di distribuzione/test/miglioramenti.

4. Distribuire presto e spesso

Elemento centrale del processo di sviluppo di Linux è la rapida e frequente distribuzione delle varie release. La maggior parte degli sviluppatori (incluso il sottoscritto) aveva sempre considerato negativa questa usanza per progetti appena più che minimi, poiché le versioni iniziali sono piene di bug quasi per definizione e non pareva il caso di far spazientire inutilmente gli utenti.
Tale opinione era rinforzata dalla generale aderenza allo stile di sviluppo della costruzione a cattedrale. Se l'obiettivo finale era quello di far vedere meno bug possibili agli utenti, allora conveniva distribuire una nuova release ogni sei mesi (o ancora meno frequentemente) e lavorare duramente sul debugging tra una release e l'altra. Fu seguendo questo processo che venne sviluppato il nucleo centrale in C di Emacs. Ma non fu così per la libreria Lisp –perché erano attivi archivi Lisp al di fuori del controllo della FSF, dove era invece possibile trovare versioni di codice nuovo e in fase di sviluppo indipendentemente dal ciclo di release di Emacs.
Il più importante di tali archivi, l'archivio elisp dell'Ohio State, anticipava lo spirito e molte delle caratteristiche tipiche dei grandi archivi su Linux di oggi. Ma pochi di noi si resero davvero conto di quel che stavamo facendo, o di come l'esistenza stessa di quell'archivio paresse rispondere ai problemi insiti nel modello di sviluppo della costruzione a cattedrale della FSF. Verso il 1992 ho provato a far confluire formalmente gran parte del codice dell'Ohio nella libreria ufficiale Lisp di Emacs. Mi sono trovato politicamente nei guai e di fronte a un chiaro insuccesso.
Ma l'anno seguente, mentre Linux diventava ampiamente visibile, fu chiaro come stesse accadendo qualcosa di assai diverso e più salutare. La policy di sviluppo aperto di Linus rappresentava l'esatto opposto della costruzione a cattedrale. Gli archivi di sunsite e tsx-11 stavano moltiplicandosi, le distribuzioni multiple andavano proliferando. E tutto ciò sotto la spinta della diffusione, con una frequenza mai vista prima, delle varie release.
Linus trattava gli utenti al pari di co-sviluppatori nella maniera più efficace possibile:
7. Distribuisci presto. Distribuisci spesso. E presta ascolto agli utenti.
L'innovazione introdotta da Linus non consisteva tanto nel seguire questa pratica (qualcosa di simile faceva parte da molto tempo della tradizione del mondo Unix), quanto piuttosto nel farla crescere a un tale livello d'intensità da raggiungere la medesima complessità del lavoro di programmazione che stava facendo. A quei tempi (intorno al 1991) non era raro che egli diffondesse versioni del nuovo kernel anche più di una volta al giorno! Qualcosa che poté funzionare grazie all'attenzione dedicata ai co-sviluppatori e all'ampio utilizzo di Internet come strumento di collaborazione.
Ma come funzionava? Era qualcosa che avrei potuto duplicare, o tutto dipendeva esclusivamente dal genio di Linus Torvalds?
No, non lo credevo. Certo, Linus è un gran bell'hacker (quanti di noi saprebbero realizzare per intero un sistema operativo di qualità?). Ma a livello concettuale Linux non rappresentava alcun significativo salto in avanti. Linus non è (almeno, non ancora) quel genio innovativo del design allo stesso modo, ad esempio, di Richard Stallman o James Gosling (di NeWS e Java). Piuttosto, Linus mi sembrava un genio dell'engineering, dotato di un sesto senso per evitare bug e strade senza uscita, oltre che di un ottimo fiuto per arrivare dal punto A al punto B con il minimo sforzo possibile. Non a caso l'intero design di Linux trasuda queste qualità e rispecchia l'approccio essenzialmente conservativo e semplificativo tipico di Linus.
Se, quindi, la rapida diffusione delle release e il pieno sfruttamento del medium Internet non erano casuali, bensì parti integranti delle visioni da genio dell'engineering di Linus lungo il cammino del minimo sforzo possibile, cos'era che stava amplificando? Cos'è che riusciva a tirar fuori da tutto questo gran daffare?
Messa così, la domanda si risponde da sola. Linus tendeva a stimolare e ricompensare costantemente i suoi hacker/utenti – stimolati dalla soddisfazione di sé per aver preso parte all'azione, ricompensati dalla vista dei miglioramenti costanti (perfino giornalieri) ottenuti nel loro lavoro.
Linus puntava direttamente a massimizzare il numero di ore/uomo coinvolte nello sviluppo e nel debugging, rischiando perfino la possibile instabilità del codice e l'estinguersi del contributo degli utenti

qualora fosse risultato impossibile tener traccia di qualche serio bug. Linus si comportava seguendo una concezione più o meno riassumibile come segue:
8. Stabilita una base di beta-tester e co-sviluppatori sufficientemente ampia, ogni problema verrà rapidamente definito e qualcuno troverà la soluzione adeguata.
O, in modo meno formale, "Dato un numero sufficiente di occhi, tutti i bug vengono a galla". Io la chiamo la "Legge di Linus".
La mia formulazione originale era che ogni problema "diventerà trasparente per qualcuno". Linus fece notare come la persona che si rende conto e risolve il problema non necessariamente né di norma è la stessa persona che per prima lo mette a fuoco. "Qualcuno scopre il problema," dice Linus, "e qualcun altro lo comprende. E secondo me il compito più difficile è proprio trovarlo". Ma il punto è che entrambe le cose tendono ad accadere piuttosto rapidamente.
Questa ritengo che sia la differenza fondamentale tra lo stile a cattedrale e quello a bazaar. Nel primo caso la visualizzazione dei problemi relativi a programmazione, bug e sviluppo costituiscono fenomeni dubbi, insidiosi, complessi. Servono mesi di scrutinio ravvicinato da parte di più d'uno per poi sentirsi sicuri di aver risolto tutti i problemi. Da qui i lunghi intervalli tra le release, e l'inevitabile delusione quando le versioni così a lungo attese si rivelano imperfette.
Nella concezione a bazaar, d'altra parte, si dà per scontato che generalmente i bug siano fenomeni marginali – o che almeno divengano rapidamente tali se esposti all'attenzione di migliaia di volenterosi co-sviluppatori che soppesano ogni nuova release. Ne consegue la rapidità di diffusione per ottenere maggiori correzioni, e come positivo effetto collaterale, c'è meno da perdere se viene fuori qualche toppa raffazzonata.
Tutto qui. E non è certo poco. Se la "Legge di Linus" è falsa, allora ogni sistema complesso tanto quanto il kernel Linux, ricavato grazie al lavoro collettivo delle molte mani che lo hanno messo insieme, a un certo punto avrebbe dovuto crollare sotto il peso di interazioni negative impreviste e di "profondi" bug non scoperti. Se invece è vera, allora è sufficiente a spiegare la relativa assenza di bug di Linux.
E forse ciò non dovrebbe rappresentare affatto una sorpresa. Qualche anno addietro sono stati i sociologi a scoprire che l'opinione media di un gruppo di osservatori equamente esperti (o equamente ignoranti) si rivela parametro assai più affidabile di quella di un solo osservatore scelto casualmente in quel gruppo. Si tratta del cosiddetto "effetto Delfi". Ora sembra che Linus abbia dimostrato come ciò vada applicato anche all'operazione di debugging di un sistema operativo – ovvero che l'effetto Delfi è in grado di addomesticare la complessità della programmazione, persino la complessità del kernel di un sistema operativo.
Sono in debito con Jeff Dutky dutky@wam.umd.edu per aver sottolineato come la Legge di Linus possa essere definita anche: "Il debugging è parallelizzabile". Jeff fa notare come nel corso dell'intero processo, pur richiedendo il coordinamento di uno sviluppatore che curi le comunicazioni tra quanti si occupano del debugging, questi ultimi invece non richiedono particolare coordinamento. In tal modo non si cade preda della notevole complessità e dei costi gestionali imposti dal coinvolgimento di nuovi sviluppatori.
In pratica, nel mondo Linux la perdita di efficienza a livello teorico, dovuta alla duplicazione di lavoro da parte di quanti seguono il debugging, non arriva quasi mai a rappresentare un problema. Uno degli effetti della policy "distribuire presto e spesso" è proprio quello di minimizzare tale duplicazione di lavoro propagando rapidamente le soluzioni giunte col feedback degli utenti.
Anche Brooks ha fatto un'osservazione su quanto sostenuto da Jeff: "Il costo totale per il mantenimento di un programma ampiamente utilizzato in genere viene valutato intorno al 40 per cento, o più, del costo dello sviluppo. Non senza sorpresa, tale costo viene notevolmente influenzato dal numero di utenti coinvolti. Maggiori sono questi ultimi, più bug si trovano."
Ciò per via del fatto che con un maggior numero di utenti ci sono più modi differenti di verificare il programma. Un effetto amplificato quando costoro sono anche co-sviluppatori. Ciascuno affronta il compito della definizione dei bug con un approccio percettivo e analitico leggermente differente, una diversa angolazione per affrontare il problema. L'effetto Delfi pare funzionare esattamente sulla base di tali differenze. Nel contesto specifico del debugging, le variazioni tendono anche a ridurre la duplicazione degli sforzi impiegati.
Quindi, dal punto di vista dello sviluppatore, l'aggiunta di altri beta-tester può non ridurre la complessità del bug "più profondo" attualmente sotto studio, ma aumenta la probabilità che l'approccio di qualcuno consentirà il corretto inquadramento del problema, così che per questa persona il bug non apparirà altro che una bazzecola.
Inoltre, in caso di seri bug, le varie versioni del kernel di Linux sono numerate in modo tale che i potenziali utenti possano scegliere o di far girare l'ultima versione definita "stabile" oppure rischiare d'incappare in possibili bug pur di provare le nuove funzioni. Una tattica ancora non formalmente imitata

dalla maggior parte di hacker Linux, ma che forse dovrebbe esserlo. Il fatto che entrambe le scelte siano disponibili le rende entrambe più attraenti.

5. Quando una rosa non è una rosa?

Dopo aver osservato il comportamento di Linus e aver elaborato una mia teoria sul perché del suo successo, ho deciso coscientemente di mettere alla prova tale teoria sul mio nuovo progetto (palesemente assai meno complesso e ambizioso).
Per prima cosa però ho semplificato parecchio popclient. Le implementazioni di Carl Harris erano precise, ma mostravano quella complessità inopportuna comune a molti programmatori in C. Trattava il codice come elemento centrale, considerando solo come supporto a latere la struttura dati. Come conseguenza, il codice era eccezionale ma il design strutturale improvvisato e bruttino (almeno secondo gli standard elevati di questo vecchio hacker di LISP).
Oltre al miglioramento del codice e del design strutturale, perseguivo comunque un altro obiettivo nell'operazione di riscrittura. Volevo si evolvesse in qualcosa che fossi in grado di comprendere pienamente. Non c'è alcun divertimento nel sistemare i problemi di un programma che non si comprende appieno.
Fu così che mi ci volle tutto il primo mese soltanto per seguire le implicazioni del progetto di base di Carl. La prima vera modifica fu l'aggiunta del supporto per IMAP. In pratica riorganizzai le macchine del protocollo in un driver generico con tre opzioni (per POP2, POP3 e IMAP). Insieme ai cambiamenti precedenti, ciò illustra il principio generale che ogni programmatore dovrebbe tenere bene a mente, soprattutto lavorando con linguaggi come il C che non accettano facilmente gli inserimenti dinamici:
9. Meglio combinare una struttura dati intelligente e un codice non eccezionale che non il contrario.
Brooks, capitolo 9: “Mostrami [il codice] e nascondimi [la struttura dati], e io continuerò a essere disorientato. Mostrami [la struttura dati], e non avrò bisogno del [codice]; sarà del tutto ovvio.”
Per esser precisi, lui parlava di “diagrammi” e “tabelle”. Ma considerando il mutamento lessicale/culturale di questi trent'anni, il senso rimane invariato.
A questo punto (inizio Settembre 1996, sei settimane dopo esser partito da zero), ho cominciato a pensare all'opportunità di cambiare il nome – in fondo non si trattava più soltanto di un client POP. Ma esitavo perché mancava ancora qualcosa di genuinamente nuovo nel design. La mia versione di popclient doveva ancora acquisire una propria identità.
Il cambio radicale avvenne quando fetchmail imparò come fare il forward della posta prelevata verso la porta SMTP. Lo spiego meglio tra poco. Prima però: più sopra ho parlato della decisione di usare questo progetto come test per verificare la mia teoria sui brillanti risultati raggiunti da Linus Torvalds. Vi potreste chiedere, in che modo l'ho messa alla prova? Ecco come:

1. Ho diffuso le varie release presto e spesso (quasi mai a meno di dieci giorni di distanza; una volta al giorno nei periodi d'intenso lavoro).
2. Ho inserito nella lista dei beta chiunque mi avesse contattato riguardo fetchmail.
3. Ho mandato simpatici messaggi all'intera lista dei beta per annunciare ogni nuova release, incoraggiando la gente a partecipare.
4. E ho dato ascolto ai beta tester, ponendo loro domande sul design adottato e plaudendoli ogni volta che mi mandavano aggiustamenti e feedback.

Questi semplici accorgimenti produssero una ricompensa immediata. Fin dall'inizio del progetto, mi arrivavano report sui bug presenti di una qualità che qualunque sviluppatore avrebbe invidiato, spesso con buone soluzioni in attach. Ho ricevuto mail piene di critiche costruttive, lodi sperticate, suggerimenti intelligenti. Il che ci porta a:
10. Se tratti i beta tester come se fossero la risorsa più preziosa, replicheranno trasformandosi davvero nella risorsa più preziosa a disposizione.
Un'interessante caratteristica del successo di fetchmail risiede nell'ampiezza dell'elenco dei beta, i “fetchmail-friend”. Si è rapidamente raggiunta quota 249, con nuovi arrivi due o tre volte la settimana. L'ultima revisione, fine Maggio 1997, ha rivelato che la lista andava perdendo membri, dopo aver raggiunto un massimo di 300 nominativi, e ciò per un motivo degno di nota. In parecchi mi hanno chiesto di essere rimossi perché fetchmail funzionava così bene che non c'era più alcun motivo di seguire il traffico della lista! Forse anche ciò fa parte del normale ciclo di vita di un progetto maturo in stile bazaar.

6. Popclient diventa Fetchmail

Il vero punto di svolta del progetto ebbe luogo quando Harry Hochheiser mi spedì il codice iniziale per fare il forward alla porta SMTP della macchina client. Mi sono reso immediatamente conto che l'implementazione affidabile di tale funzione avrebbe reso pressoché obsoleta ogni altra modalità di consegna della posta.
Per molte settimane mi ero messo a giocare con l'interfaccia di fetchmail, passabile ma disordinata – poco elegante e con troppe opzioni sparse tutt'intorno. Tra queste mi davano particolarmente fastidio, anche senza capire perché, quelle utilizzate per trasferire la posta prelevata in una certa mailbox o altrove.
Quel che mi veniva in mente pensando alla funzione del "SMTP forwarding" era che popclient voleva cercare di far troppe cose. Era stato ideato per essere sia un "mail transport agent" (MTA) sia un "mail delivery agent" (MDA) a livello locale. Con il forward SMTP avrebbe potuto smettere di essere un MDA per divenire un puro MTA, trasferendo ad altri programmi la posta per la consegna locale, proprio come fa sendmail.
Perché darsi da fare a sistemare le complesse configurazioni per un MDA o le impostazioni per le mailbox, quando innanzitutto è quasi sempre garantito che la porta 25 rimane disponibile per questo su ogni piattaforma con supporto TCP/IP? Soprattutto quando ciò significa che i messaggi prelevati appariranno come posta SMTP normalmente inviata dal mittente, che è poi quel che stiamo cercando di ottenere.
Ci sono diverse lezioni da trarre a questo punto. Primo, l'idea del "SMTP forwarding" era la prima grossa ricompensa per aver tentato coscientemente di emulare i metodi di Linus. Era stato un utente a suggerirmi questa fantastica idea – non mi restava che comprenderne le implicazioni.
11. La cosa migliore, dopo l'avere buone idee, è riconoscere quelle che arrivano dagli utenti. Qualche volta sono le migliori.
Fatto interessante, è facile scoprire che se sei completamente onesto e autocritico su quanto è dovuto agli altri, il mondo intero ti tratterà come se ogni bit di quell'invenzione fosse opera tua, mentre impari a considerare con sempre maggior modestia il tuo genio innato. Abbiamo visto come tutto ciò abbia funzionato a meraviglia con Linus!
(Quando ho presentato questo scritto alla conferenza su Perl dell'Agosto 1997, in prima fila c'era Larry Wall. Non appena sono arrivato al punto di cui sopra, ha intonato in stile revival-religioso, "Diglielo, diglielo, fratello!" Tutti i presenti si son messi a ridere, perché sapevano come ciò avesse funzionato bene anche per l'inventore di Perl.)
Dopo aver lavorato sul progetto nello stesso spirito per alcune settimane, mi son visto arrivare lodi simili non soltanto dagli iscritti alla lista ma anche da altre persone che venivano a sapere della cosa. Ho conservato qualche email; forse me le andrò a rileggere nel caso iniziassi a chiedermi se la mia vita abbia mai avuto un qualche valore :-).
Ma ci sono altre due lezioni da trarre, più fondamentali e non politiche, buone per ogni tipo di design.
12. Spesso le soluzioni più interessanti e innovative arrivano dal fatto di esserti reso conto come la tua concezione del problema fosse errata.
Avevo cercato di risolvere il problema sbagliato continuando a lavorare su popclient in quanto combinazione MTA/MDA con tutte le possibili modalità di consegna della posta. Il design di fetchmail aveva bisogno di essere reimpostato dall'inizio come un puro MTA, a parte il normale percorso della posta relativo a SMTP.
Quando sbatti la testa contro il muro nel lavoro di programmazione – quando cioè non riesci più a pensare alla prossima "patch" – spesso è ora di chiedersi non se hai la risposta giusta, ma se ti stai ponendo la giusta domanda. Forse bisogna inquadrare nuovamente il problema.
Be', mi toccò inquadrare meglio il problema. Chiaramente la cosa giusta da fare era (1) posizionare il supporto per il "SMTP forwarding" nel driver generico, (2) farla diventare la funzione default, (3) sbarazzarsi di tutte le altre modalità di consegna della posta, specialmente le opzioni "deliver-to-file" e "deliver-to-standard-output".
Per qualche tempo ho esitato a compiere il passo (3), temendo di avvilire quanti usavano da molto tempo popclient proprio per i diversi meccanismi di consegna. In teoria, avrebbero potuto immediatamente passare ai file ".forward" oppure agli equivalenti "non-sendmail" per ottenere il medesimo effetto. In pratica, però, tale transizione sarebbe risultata impraticabile.
Quando mi decisi comunque a farlo, ne risultarono enormi benefici. Le parti più confuse del codice del driver scomparvero. La configurazione divenne radicalmente più semplice – niente più girovagare nel sistema MDA e nella mailbox, niente più preoccupazioni per vedere se il sistema operativo supportasse o meno il blocco dei file.

Venne anche eliminata l'unica possibilità di perdere dei messaggi. Se, specificando la consegna in un file, il disco è pieno, la posta va perduta. Impossibile che ciò accada con il “SMTP forwarding” perché l'altro SMTP in ascolto non accetterà l'OK a meno che il messaggio non possa essere correttamente consegnato o almeno filtrato per il prelievo successivo.
Inoltre, le prestazioni complessive risultarono migliorate (anche se così non sembra quando lo si fa girare una sola volta). Altro beneficio non insignificante del cambiamento fu che la chiamata manuale risultò assai più semplificata.
In seguito fui costretto a reintrodurre la funzione di consegna tramite un MDA locale specificato dall'utente, per consentire la gestione di strane situazioni relative allo SLIP dinamico. Ma riuscii a farlo in maniera più semplice.
Morale? Non esitare a buttar via opzioni inanellate una sull'altra quando puoi rimpiazzarle senza perdere in efficienza. Diceva Antoine de Saint-Exupéry (aviatore e designer di aerei, quando non scriveva libri per bambini):
13. “La perfezione (nel design) si ottiene non quando non c'è nient'altro da aggiungere, bensì quando non c'è più niente da togliere.”
Quando il codice diventa migliore e più semplice, allora vuol dire che va bene. E nel processo, il design di fetchmail acquistò una sua propria identità, diversa dal popclient originario.
Era giunta l'ora di cambiar nome. Il nuovo design assomigliava più a sendmail di quanto lo fosse il vecchio popclient; entrambi sono MTA, ma mentre sendmail spinge e poi consegna, il nuovo popclient tira e poi consegna. Così, a due mesi dai blocchi di partenza, decisi di dargli il nuovo nome, fetchmail.

7. Fetchmail diventa adulto

Eccomi qui con un design ben fatto e innovativo, un codice che ero certo funzionasse bene perchè lo usavo ogni giorno, e una spumeggiante lista di beta tester. Gradualmente mi resi conto che non ero più indaffarato con uno stupido programmino personale che forse avrebbe potuto interessare pochi altri. Stavo lavorando su un programma di cui ogni hacker dotato di mailbox Unix e connessione SLIP/PPP non avrebbe potuto fare a meno.
Grazie all'opzione di “SMTP forwarding”, superava sicuramente i programmi rivali fino a diventare potenzialmente una “categoria killer”, uno di quei programmi classici che occupano la propria nicchia in maniera perfetta, facendo scartare e quasi dimenticare ogni possibile alternativa.
Credo che simili risultati siano impossibili da perseguire o da pianificare. Devi esser trascinato dentro la storia da idee così potenti che, col senno di poi, quei risultati appaiono del tutto inevitabili, naturali, perfino prestabiliti. L'unico modo per provarci è farsi venire un sacco di idee, oppure avere la capacità di portare le idee degli altri al di là del punto in cui essi stessi credevano potessero arrivare.
Andrew Tanenbaum ebbe l'idea originale di realizzare un linguaggio di base Unix per il 386, da usare come strumento didattico. Linus Torvalds ha spinto il concetto del Minix ben oltre quanto lo stesso Andrew ritenesse possibile – ed è diventato qualcosa di meraviglioso. Allo stesso modo (pur se su scala minore) io ho preso alcune idee da Carl Harris e Harry Hochheiser, e le ho spinte oltre. Nessuno di noi è stato “originale” nel senso romantico in cui si immagina un genio. Ma a ben vedere la maggior parte della scienza, dell'ingegneria e dello sviluppo del software non viene realizzata da alcun genio originale, il contrario della mitologia dell'hacker.
I risultati ottenuti erano piuttosto notevoli – meglio, esattamente quel tipo di successo che ogni hacker sogna! E ciò significa che avrei potuto mirare anche a standard più elevati. Per rendere fetchmail così ben fatto come lo vedevo ora, avrei dovuto scrivere non soltanto per le mie necessità ma anche per includere il supporto di opzioni necessarie ad altri e tuttavia fuori dalla mia orbita. E fare ciò mantenendo al contempo semplice e robusto il programma.
La prima funzione di gran lunga più importante che scrissi dopo essermi reso conto di ciò, fu il supporto multiutente – la possibilità di prelevare la posta da più mailbox in cui erano stati accumulati tutti i messaggi per un gruppo di utenti, e quindi smistare ogni singolo messaggio ai rispettivi destinatari.
Decisi di aggiungere il supporto multiutente in parte perché alcuni lo richiedevano, ma soprattutto perché pensavo avrebbe buttato via ogni bug dal codice per un solo utente, costringendomi a fare attenzione alla gestione dei vari indirizzi. E così fu. Mi ci volle parecchio tempo per sistemare tutto nella RFC 822, non perché sia difficile mettere a posto ogni singola parte, ma perché coinvolgeva una montagna di dettagli interdipendenti e instabili.
In ogni caso, anche la funzione per gli indirizzi multiutenti si rivelò un'ottima decisione. Ecco come me ne sono accorto:

14. Ogni strumento dovrebbe rivelarsi utile nella maniera che ci si attende, ma uno strumento davvero ben fatto si presta ad utilizzi che non ci si aspetterebbe mai.
L'uso inatteso della funzione multiutenti è per una mailing list quando questa viene mantenuta sul lato client della connessione SLIP/PPP, attivando l'espansione dell'alias. Ne consegue che chi fa girare un PC tramite un account con un provider Internet è in grado di gestire una mailing list senza dover accedere continuamente ai file alias del provider.
Altra importante modifica richiesta dai beta tester fu il supporto per operazioni MIME in 8-bit. Qualcosa piuttosto semplice a farsi, poiché avevo fatto bene attenzione a mantenere il codice pulito per inserire l'8-bit. Non perché avessi anticipato tale richiesta, quanto piuttosto per rispettare un'altra regola:
*15. Quando si scrive del software per qualunque tipo di gateway, ci si assicuri di disturbare il meno possibile il flusso dei dati – e *mai* buttar via alcun dato a meno che il destinatario non ti ci costringa!*
Se non avessi rispettato questa regola, il supporto per MIME in 8-bit sarebbe risultato difficile e problematico. Così invece tutto quel che dovetti fare fu leggere la RFC 1652 e aggiungere poche stringhe per far generare l'header.
Alcuni utenti europei mi hanno costretto a inserire un'opzione per limitare il numero dei messaggi prelevati a ogni sessione (in modo da controllare i costi di collegamento, per via delle care tariffe imposte dalle aziende telefoniche nazionali). Ho resistito per parecchio tempo, e ancor'oggi non ne sono pienamente soddisfatto. Ma se scrivi programmi per tutto il mondo, devi dare ascolto ai tuoi clienti – e ciò rimane valido anche se non ti ricompensano in denaro.

8. Qualche altra lezione da Fetchmail

Prima di tornare alle questioni generali sul “software-engineering”, è il caso di considerare qualche altro insegnamento specifico a seguito di quest'esperienza con fetchmail.
La sintassi del file rc include una serie di parole-chiave facoltative completamente ignorate dall'analizzatore. La sintassi in “quasi-inglese” che si ottiene è notevolmente più leggibile delle semplici coppie nome/valore che si rimangono dopo aver eliminato le prime.
Tutto ebbe inizio come un esperimento a notte fonda dopo essermi accorto di quante fossero le dichiarazioni del file rc che cominciavano ad assomigliare a un minilinguaggio imperativo. (Per lo stesso motivo ho modificato in “poll” la parola-chiave “server” del popclient originale).
Mi venne da pensare che sarebbe stato più facile usare qualcosa di simile all'inglese comune piuttosto che quel minilinguaggio imperativo. Ora, pur essendo il sottoscritto un convinto fautore della scuola di design del tipo “trasformalo in linguaggio”, come esemplificato da Emacs, dall'HTML e da molti motori di database, generalmente non mi annovero tra i fan delle sintassi in “quasi-inglese”.
Tradizionalmente i programmatori hanno sempre avuto la tendenza a favorire sintassi molto precise e compatte, del tutto prive di ridondanza. Si tratta di un'eredità culturale del tempo in cui le risorse informatiche erano costose, così gli analizzatori dovevano risultare semplici ed economici al massimo grado. Allora l'inglese, con quel 50% di ridondanza, sembrava un modello poco appropriato.
Non è questa la ragione per cui generalmente io evito le sintassi in quasi-inglese; l'ho citata qui soltanto per demolirla. Con l'attuale economicità dei cicli e delle strutture, la pulizia non dovrebbe essere un obiettivo in sé. Al giorno d'oggi è più importante che un linguaggio sia conveniente per gli esseri umani anziché economico per il computer.
Esistono comunque buoni motivi per procedere con cautela. Uno è rappresentato dai costi della complessità dell'analizzatore – non è il caso di aumentare tale complessità fino a raggiungere il punto in cui produrrà bug significativi e confusione nell'utente. Un'altra ragione è che cercare di rendere un linguaggio in quasi-inglese spesso richiede un tale aggiustamento linguistico che le somiglianze superficiali con il linguaggio naturale generino confusione tanto quanto l'eventuale sintassi tradizionale. (Come si può notare con evidenza nei linguaggi di interrogazione dei database di “quarta generazione” e commerciali).
La sintassi di controllo di fetchmail riesce a evitare questi problemi perché il dominio riservato al linguaggio è estremamente limitato. Non si avvicina neppure lontanamente a un linguaggio di tipo generale; le cose che dice non sono affatto complicate, lasciando quindi poco spazio a potenziali confusioni, quando ci si sposta mentalmente tra un ristretto ambito d'inglese e il linguaggio di controllo vero e proprio. Credo qui si tratti di una lezione di più ampia portata:
16. Quando il linguaggio usato non è affatto vicino alla completezza di Turing, un po' di zucchero sintattico può esserti d'aiuto.

Un'altra lezione riguarda la sicurezza “al buio”. Alcuni utenti di fetchmail mi avevano chiesto di modificare il software in modo che potesse conservare le password crittografate nel file rc, evitando così il rischio che qualche ficcanaso potesse scoprirle casualmente.
Non l'ho fatto perché in realtà ciò non aggiunge alcuna protezione. Chiunque sia stato autorizzato a leggere il file rc, sarebbe comunque in grado di far girare fetchmail – e se è la password che cerca, saprebbe come decodificarla tirandola fuori dallo stesso codice di fetchmail.
Qualsiasi possibile codifica della password operata da .fetchmailrc, non avrebbe fatto altro che fornire un falso senso di sicurezza a quelli che non vogliono spremersi le meningi.
La regola generale è:
17. Un sistema di sicurezza è sicuro soltanto finché è segreto. Meglio diffidare degli pseudo-segreti.

9. Le pre-condizioni necessarie per lo stile bazaar

I primi lettori e revisori di questo scritto hanno ripetutamente sollevato domande sulle pre-condizioni necessarie per il successo dello sviluppo in stile bazaar, comprendendo con ciò sia le qualifiche del leader del progetto sia lo stato del codice al momento della diffusione pubblica e della nascita della possibile comunità di co-sviluppatori.
È alquanto evidente come lo stile bazaar non consenta la scrittura del codice partendo da zero. Si possono fare test, trovare i bug, migliorare il tutto, ma sarebbe molto difficile dar vita dall'inizio a un progetto in modalità bazaar. Linus non lo ha fatto. Neppure io. La nascente comunità di sviluppatori deve avere qualcosa da far girare e con cui giocare.
Quando s'inizia a costruire la comunità, bisogna essere in grado di presentare una promessa plausibile. Non è detto che il programma debba funzionare particolarmente bene. Può anche essere crudo, pieno di bug, incompleto, scarsamente documentato. Non deve però mancare di convincere i potenziali co-sviluppatori che possa evolversi in qualcosa di veramente ben fatto nel prossimo futuro.
Quando Linux e fetchmail vennero diffusi pubblicamente, erano dotati di un design di base forte e attraente. Molte persone ritengono che il modello bazaar da me presentato riveli correttamente questa fase critica, per poi da qui saltare alla conclusione che sia indispensabile un elevato livello di intuizione e bravura da parte di chi guida il progetto.
Ma Linus prese il suo design da Unix. All'inizio ho fatto lo stesso con il popclient originario (anche se poi risultò molto diverso, assai più di quanto è accaduto con Linux, fatte le debite proporzioni). È dunque vero che il leader/coordinatore di un progetto in stile bazaar debba possedere un eccezionale talento nel design? Oppure può cavarsela facendo leva sui talenti altrui?
Non credo sia essenziale che il coordinatore possa produrre design eccezionali, ma è assolutamente centrale che sia capace di riconoscere le buone idee progettuali degli altri.
Ciò appare evidente da entrambi i progetti di Linux e fetchmail. Pur non essendo un designer particolarmente originale (come discusso in precedenza), Linus ha dimostrato un ottima intuizione nel saper riconoscere il buon design per poi integrarlo nel kernel di Linux. Ed ho già descritto come l'idea più potente di fetchmail (SMTP forwarding) mi sia arrivata da qualcun altro.
I primi lettori di questo testo mi hanno fatto i complimenti mettendo a fuoco la mia propensione a sottovalutare l'originalità del design nei progetti in stile bazaar perchè ne possiedo a volontà io stesso, e quindi la dò per scontata. In effetti, c'è qualcosa di vero in quest'affermazione; il design (in alternativa al codice o al debugging) è la mia capacità migliore.
Ma il problema con il fatto di essere bravi e originali nel design del software è che tende a divenire un'abitudine – prendi a fare cose carine e complicate quando invece dovresti tenerle semplici e robuste. Proprio per questa ragione mi sono crollati addosso vari progetti, ma con fetchmail sono stato attento a non farlo.
Credo insomma che il progetto di fetchmail abbia avuto successo in parte perché ho limitato la mia tendenza a esser bravo; ciò va (almeno) contro l'essenzialità dell'originalità del design per il successo dei progetti a bazaar. Consideriamo Linux. Supponiamo che Linus Torvalds avesse cercato di tirar fuori le innovazioni fondamentali dal design del sistema operativo nel corso dello sviluppo; quali probabilità esistono che il kernel risultante fosse ugualmente stabile ed efficace come quello che abbiamo ora?
È chiaro che occorrano capacità di un certo livello per il design e il codice, ma personalmente mi aspetto, da quasi tutte le persone che pensano seriamente di lanciare un progetto bazaar, un livello superiore. Il mercato interno della reputazione della comunità open source esercita una sottile pressione sulle

persone in modo che non si lancino dei progetti se non si è abbastanza competenti per seguirli. Finora quest'approccio ha funzionato piuttosto bene.
Esiste un altro tipo di capacità normalmente non associata allo sviluppo del software che ritengo importante al pari della bravura nel design per i progetti bazaar – anzi, forse ancora più importante. Il coordinatore o leader deve essere in grado di comunicare efficacemente con gli altri.
D'altronde è ovvio che per metter su una comunità di sviluppatori occorra attirare gente, coinvolgerli in quel che stai facendo, tenerli contenti per il lavoro che fanno. Lo sfrigolìo tecnico aiuta molto in questo senso, ma è ben lungi dall'esser tutto. È anche importante il tipo di personalità che proietti.
Non è certo una coincidenza che Linus sia un tipo simpatico, capace di piacere alla gente e di farsi aiutare. Da parte mia, io sono un estroverso energetico cui piace lavorare tra la folla, oltre ad avere qualcosa dei modi e dell'istinto del comico. Per far funzionare il modello a bazaar, aiuta parecchio essere in grado di esercitare almeno un po' di fascino sulla gente.

10. Il contesto sociale del software open source

È proprio vero: le migliori operazioni di hacking nascono come soluzioni personali ai problemi quotidiani dell'autore, e si diffondono perchè si scopre che tali problemi sono comuni a molte altre persone. Questo ci riporta indietro alla questione della regola numero uno, riformulata forse in maniera più consona:
18. Per risolvere un problema interessante, comincia a trovare un problema che risvegli il tuo interesse.
Così è successo con Carl Harris e l'iniziale popclient, lo stesso con me e fetchmail. Ma ciò era chiaro da molto tempo. Il punto interessante che richiede attenzione, sulla base dell'evolversi di Linux e di fetchmail, è il palcoscenico successivo – l'evoluzione del software in presenza di una vasta e attiva comunità di utenti e co-sviluppatori.
In “The Mythical Man-Month”, Fred Brooks osserva come il tempo del programmatore non sia calcolabile; aggiungendo altri sviluppatori ad un progetto in ritardo, lo si fa tardare ancora di più. Secondo lui, i costi della complessità e delle comunicazioni di un progetto crescono esponenzialmente con il numero degli sviluppatori coinvolti, mentre il lavoro cresce soltanto in senso lineare.
Quest'affermazione è nota come la “Legge di Brooks”, ed è considerata una verità pressoché universale. Ma se la Legge di Brooks fosse stata l'unica verità, Linux non sarebbe mai esistito.
Il classico di Gerald Weinberg “The Psychology Of Computer Programming” spiega in che modo, a posteriori, sia possibile individuare una vitale correzione alla tesi di Brooks. Parlando di “programmazione senza ego”, Weinberg fa notare come laddove gli sviluppatori non si dimostrano territoriali rispetto al proprio codice, incoraggiando altre persone a cercare bug e offrire miglioramenti, questi ultimi prendono corpo molto più in fretta che altrove.
Forse le scelte terminologiche operate da Weinberg hanno impedito alla sua analisi di ottenere il riconoscimento che merita – viene da ridere al solo pensiero di riuscire a descrivere un hacker “senza ego”. Ma ritengo la sua posizione stringente più che mai.
La storia di Unix avrebbe dovuto prepararci per quel che stiamo imparando da Linux (e per quello che io stesso ho verificato sperimentalmente su scala ridotta, copiando deliberatamente i metodi di Linus). Ovvero, pur rimandando la scrittura del codice un'attività prettamente solitaria, la questione davvero importante rimane la capacità di sfruttare l'attenzione e la potenza dell'intera comunità. Lo sviluppatore che impiega soltanto il proprio cervello su un progetto chiuso risulterà sempre dietro allo sviluppatore che sa come creare un contesto aperto, evolutivo dove sono centinaia le persone che si occupano dei miglioramenti e del debugging.
Ma una serie di elementi hanno impedito al mondo tradizionale Unix di applicare tale approccio. Tra questi, gli impedimenti legali connessi alle varie licenze, ai segreti e agli interessi commerciali. Altro impedimento (in retrospettiva), il fatto che allora Internet non fosse ancora abbastanza sviluppata. Prima dell'attuale Internet super-diffusa, esistevano poche comunità geograficamente compatte in cui veniva incoraggiata la cultura della programmazione “senza ego” di Weinberg, dove uno sviluppatore poteva facilmente attirare molti altri co-sviluppatori in gamba. Il Bell Lab, il MIT, UC Berkeley – queste le dimore dell'innovazione che rimangono leggendarie e potenti ancor'oggi.
Linux è stato il primo progetto a proporre lo sforzo cosciente e coronato da successo verso l'utilizzo del mondo intero come fucina di talenti. Non ritengo una combinazione il fatto che la gestazione di Linux sia coincisa con la nascita del World Wide Web, e che Linux si sia lasciato alle spalle l'infanzia negli stessi anni 1993-1994 che hanno visto il decollo dei provider locali e l'esplosione dell'interesse di massa per

Internet. Linus è stata la prima persona ad aver imparato come giocare secondo le nuove regole rese possibili dalla diffusione di Internet.
Se è vero che tale diffusione si è rivelata necessaria per l'evoluzione del modello Linux, non credo però possa ritenersi da sola una condizione sufficiente. Altro fattore vitale è stata l'implementazione di un certo stile di leadership e di un insieme di usanze cooperative che hanno consentito agli sviluppatori di coinvolgere altri co-sviluppatori e ottenere il massimo possibile dal medium stesso.
Ma cosa s'intende esattamente con un certo stile di leadership e quali sarebbero queste usanze cooperative? Intanto, non ci si basa su relazioni di potere – e anche se tali dovessero essere, una leadership fondata sulla costrizione non produrrebbe i risultati che abbiamo visto. Weinberg cita al riguardo l'autobiografia dell'anarchico russo del XIX secolo Pyotr Alexeyvich Kropotkin, "Memorie di un rivoluzionario":
"Essendo cresciuto in una famiglia che aveva dei servitori, sono entrato nella vita attiva, al pari di tutti i giovani della mia epoca, con un notevole carico di confidenza nella necessità di comandare, impartire ordini, rimproverare, punire. Ma quando, ancora giovane, dovetti gestire degli affari seri e avere a che fare con uomini [liberi], quando ogni errore avrebbe portato da solo a pesanti conseguenze, iniziai ad apprezzare la differenza tra l'agire basato sul principio del comando e della disciplina e l'agire basato sul principio della comprensione condivisa. Il primo funziona mirabilmente in una parata militare, ma non ha valore alcuno allorché si tratta della vita reale, dove ogni obiettivo può essere raggiunto soltanto tramite i duri sforzi di molte volontà convergenti."
È precisamente i "duri sforzi di molte volontà convergenti" sono quel che un progetto come Linux richiede – e il "principio del comando" è veramente impossibile da praticare tra i volontari di quel paradiso anarchico chiamato Internet. Per operare e competere con efficacia, ogni hacker che voglia guidare progetti collettivi deve imparare come creare e dare energia a reali comunità d'interesse secondo le modalità vagamente suggerite dal "principio della comprensione" citato da Kropotkin. Deve imparare ad usare la Legge di Linus.
Più sopra mi sono riferito all'effetto Delfi come possibile spiegazione della Legge di Linus. Ma si potrebbero anche fare analogie forse più calzanti con i sistemi d'adattamento delle scienze biologiche ed economiche. Sotto molti aspetti il mondo Linux si comporta come un "free market" oppure come un sistema ecologico, una serie di agenti indipendenti che cercando di massimizzare quell'utilitarismo che nel processo va producendo un ordine spontaneo e in grado di auto-correggersi, più elaborato ed efficiente di quanto avrebbe potuto raggiungere qualsiasi pianificazione centralizzata. È dunque questo il luogo dove cercare il "principio della comprensione".
La "funzione utilitaristica" che gli hacker di Linux vanno massimizzando non è economica in senso classico, quanto piuttosto espressione dell'intangibile, egoistica reputazione e soddisfazione che si guadagna tra gli altri hackers. (La loro motivazione potrebbe essere definita "altruista", ma ciò significherebbe ignorare il fatto che a ben vedere l'altruismo stesso altro non è che una forma di soddisfazione egoistica). In realtà le culture del lavoro volontario che funzionano in tal modo non sono così rare; un'altra cui ho preso parte a lungo è quella dei fan della fantascienza, che al contrario del giro hacker riconosce esplicitamente come motore propulsore dietro tale attività volontaria proprio il cosiddetto "egoboo" (l'esaltazione della reputazione individuale tra gli altri fan).
Linus, posizionandosi con successo come filtro di un progetto nel quale il lavoro è in gran parte svolto da altri, e alimentando interesse nel progetto stesso finché non arriva ad auto-alimentarsi, ha dimostrato di aver acutamente fatto proprio il "principio della comprensione condivisa" di Kropotkin. Questa visione quasi-economica del mondo Linux ci consente di vedere come applicare tale comprensione.
È possibile ritenere il metodo di Linus come un modo per creare un mercato efficiente all'interno dell'"egoboo" – per collegare nel modo più sicuro possibile l'egoismo dei singoli hacker con quegli obiettivi difficili che possono essere raggiunti soltanto grazie alla concreta cooperazione collettiva. Con il progetto fetchmail ho dimostrato (pur se su scala più ridotta) che è possibile duplicarne il metodo ottenendo buoni risultati. Forse l'ho perfino messo in atto un po' più coscientemente e sistematicamente di quanto non abbia fatto Linus.
Molte persone (soprattutto quanti politicamente diffidano del "free market") immaginavano che una cultura di egoisti auto-referenziale si rivelasse frammentaria, territoriale, sprecona, segreta, ostile. Ma tale aspettativa viene chiaramente confutata (per fornire un solo esempio) dalla notevole varietà, qualità e profondità della documentazione relativa a Linux. Se è un dato di fatto che i programmatori odiano lavorare sulla documentazione, com'è allora che gli hacker di Linux ne producono di così copiosa? Evidentemente il "free market dell'egoboo" di Linux funziona meglio nella produzione di comportamenti virtuosi, diretti verso gli altri, rispetto a quei negozi dei produttori di software commerciale che si occupano della documentazione, avendo alle spalle massicci finanziamenti.

Sia il progetto di fetchmail che del kernel di Linux dimostrano come, dando la giusta ricompensa all'ego di molti altri hacker, un bravo sviluppatore/coordinatore possa utilizzare Internet per catturare i vantaggi dell'avere a disposizione molti co-sviluppatori senza che il progetto si frantumi in una confusione caotica. Ecco quindi la mia controproposta alla Legge di Brooks:
19: Stabilito che il coordinatore dello sviluppo abbia a disposizione un medium almeno altrettanto affidabile di Internet, e che sappia come svolgere il ruolo di leader senza costrizione, molte teste funzionano inevitabilmente meglio di una sola.
Penso che il futuro del software open source apparterrà sempre più alle persone che sanno come giocare al gioco di Linus, persone che si lasceranno alle spalle la cattedrale per entrare nel bazaar. Ciò non significa che non saranno più importanti le visioni e l'intelligenza individuali; credo piuttosto che la punta avanzata del software open source apparterrà a quanti sapranno muovere da visioni e intelligenza individuali per poi amplificarle tramite l'effettiva costruzione di comunità volontarie d'interesse.
E forse ciò vale non soltanto per il futuro del software open source. Nessuno sviluppatore in “closed-source” potrà mai pareggiare la fucina di talenti che la comunità Linux è in grado di riunire per affrontare un problema. Sono davvero in pochi a potersi permettere di assumere le oltre duecento persone che hanno contribuito a fetchmail!
Forse alla fine la cultura dell'open source trionferà non perchè la cooperazione sia moralmente giusta o perché il software “costretto” sia moralmente sbagliato (dando per scontato che si creda a quest'ultima affermazione, cosa che né io né Linus facciamo), ma semplicemente perché il mondo “closed-source” non è in grado di vincere la corsa agli armamenti dell'evoluzione contro quelle comunità open source capaci di affrontare un problema con tempi e capacità superiori di diversi ordini di grandezza.

11. Ringraziamenti

Questo scritto è stato migliorato grazie alle conversazioni avute con molte persone che hanno contribuito a trovarne i bug. Ringrazio in modo particolare Jeff Dutky , che mi ha suggerito la formulazione “il debugging è parallelizzabile” e che mi ha aiutato a svilupparne le analisi conseguenti. Grazie anche a Nancy Lebovitz per avermi suggerito di emulare Weinberg citando Kropotkin. Critiche costruttive sono venute anche da Joan Eslinger e Marty Franz della lista General Technics. Sono grato ai membri del PLUG, Philadelphia Linux User's Group, per essersi prestati come primo test pubblico per la versione iniziale di questo testo. Infine, i commenti iniziali di Linus Torvalds sono stati di grande aiuto e il suo sostegno molto incoraggiante.

12. Letture consigliate

Ho citato diversi passaggi dal classico di Frederick P. Brooks “The Mythical Man-Month” perché, in molti sensi, le sue introspezioni meritano ulteriori riflessioni. Raccomando di cuore l'edizione pubblicata in occasione del 25° Anniversario da Addison-Wesley (ISBN 0-201-83595-9), che contiene anche il suo saggio del 1986 “No Silver Bullet”.

Questa nuova edizione contiene un'importante retrospettiva dei 20 anni successivi in cui Brooks ammette schiettamente come alcuni giudizi espressi nel testo originale non abbiano sostenuto la prov