Produrre Software Open Source

Progettare un Software Libero di successo

Karl Fogel
Gerlando Termini
Giovanni Giorgio
Luca Quaglia

Produrre Software Open Source: Progettare un Software Libero di successo

di Karl Fogel, Gerlando Termini, Giovanni Giorgio, e Luca Quaglia

Dedica

Questo libro è dedicato a due cari amici senza i quali la realizzazione sarebbe stata impossibile: Karen Underhill e Jim Blandy.

Indice

Prefazione

Perché ho scritto questo libro?

A volte, partecipando ad una festa, mi capita di pronunciare la frase "Io scrivo software libero". I primi tempi la gente mi guardava con aria perplessa, adesso rispondono "Ah, si, a codice aperto come Linux?" Annuisco compiaciuto. "Sì, esattamente! Ecco cosa faccio." Provo una piacevole sensazione nel non essere più considerato un alieno. In passato, la domanda successiva era generalmente scontata: "Come fai a guadagnare in questo modo?" Per rispondere, mi toccava riassumere l'economia dello sviluppo a codice aperto (meglio noto come *open source*): spiegavo che ci sono organizzazioni il cui interesse è di risolvere un problema attraverso un dato software, ma alle quali non serve venderne nemmeno una copia, basta che il programma sia disponibile e aggiornato, più come strumento che come merce.

Ultimamente, tuttavia, questa domanda successiva non è sempre stata legata al denaro. Il modello di *business* dei programmi a codice aperto[1] non è più così misterioso, ed un numero sempre maggiore di non addetti ai lavori già comprende — o per lo meno non rimane sconvolto — che alcune persone vi lavorano a tempo pieno. Invece la domanda che iniziano a pormi con maggior frequenza è "*Interessante, ma come funziona?*"

In principio non avevo una risposta soddisfacente a portata di mano, e più mi sforzavo di trovarne una, più mi rendevo conto quanto fosse difficile l'argomento. Dar vita ad un software libero non è esattamente come dirigere un'impresa (immagina di dover costantemente negoziare la natura del tuo prodotto con un gruppo di volontari, la maggior parte dei quali non li hai mai neppure visti in faccia). Tantomeno si può paragonare, per varie ragioni, ad organizzare e gestire un'associazione senza fini di lucro di tipo tradizionale, oppure governativa. Ci sono similitudini con tutte, ma alla fine sono lentamente giunto alla conclusione che il software libero è una cosa *sui generis*. I settori ai quali paragonarlo sono tanti, ma nessuno di questi è perfettamente uguale. A dirla tutta, persino l'assunzione che il software libero può essere "diretto", è una specie di forzatura. Un software libero può certamente essere *avviato*, ed il suo sviluppo può subire l'influenza di collaboratori interessati, anche in maniera forte. Ma le sue fondamenta non possono essere proprietà di un singolo, e finché ci sono persone in giro — ovunque — interessate nel suo sviluppo, non esiste modo unilaterale per cessarne la diffusione. Ognuno coinvolto ha un potere infinito, ed allo stesso tempo non ha nessun potere specifico. Dando vita ad una dinamica interessante.

Ecco il motivo per cui ho deciso di scrivere questo libro. I progetti che sviluppano software libero hanno fatto nascere una cultura differente, un'etica nella quale la libertà di creare programmi che facciano qualsiasi cosa si abbia in mente è al centro di tutto, e dove il risultato di questa libertà non sono individui sparpagliati che decidono separatamente come far evolvere il codice, ma è una collaborazione piena di entusiasmo. Proprio il "saper collaborare" diventa quindi una delle competenze di maggior valore, nel software libero. Gestire questi progetti vuol dire entrare a far parte di una specie di cooperazione ipertrofica, in cui l'abilità del singolo è di scoprire nuove metodologie per lavorare insieme, al fine di fornire tangibili benefici al software stesso. Questo libro vuol provare a descrivere le tecniche attraverso le quali tale processo può essere portato avanti. Non ha la pretesa di trattare l'argomento in maniera esaustiva, ma è pur sempre un inizio.

Fare del *buon* software libero è un obiettivo valido già di per sé. Spero quindi che il lettore desideroso di scoprire nuovi modi per raggiungere quest'obiettivo, si senta soddisfatto di ciò che troverà in questo libro. Ma soprattutto spero di trasmettere parte del puro piacere che si prova a lavorare con una squadra motivata di sviluppatori *open source*, e ad interagire con gli utenti nel modo meraviglioso

[1] I termini "codice aperto" e "libero" sono essenzialmente dei sinonimi in questo contesto, saranno discussi con maggior dettaglio in sezione chiamata «"Free" e "open source" a confronto» nel Capitolo 1, *Introduzione*.

che il software libero incoraggia. Partecipare alla progettazione di un programma aperto di successo è *divertente*, ed in definitiva è ciò che consente all'intero sistema di continuare ad esistere.

Chi dovrebbe leggere questo libro?

Questo libro è pensato per gli sviluppatori di software e per i dirigenti che stanno pensando di far partire un progetto open source, o che ne hanno già avviato uno, chiedendosi come andare avanti. I concetti illustrati potrebbero anche tornare utili per coloro che desiderano partecipare ad un progetto open source, ma non ne hanno mai avuto l'occasione

Il livello di difficoltà non richiede che il lettore sia un programmatore, ma dovrebbero essere noti i concetti di basi dell'ingegneria del software come il codice sorgente, il compilatore e le *patch*.

Una precedente esperienza sull'open source, sia come utente che come sviluppatore, non è necessaria. Coloro che hanno invece lavorato in progetti relativi al software libero, troveranno probabilmente alcune parti del libro un tantino noiose e scontate, e potranno decidere di saltarle senza alcun problema per la comprensione dei concetti seguenti. Proprio a tal proposito, ho posto particolare attenzione nell'attribuzione dei titoli alle sezioni in maniera chiara, indicando quando qualcosa può essere saltata da chi ha già dimestichezza con l'argomento trattato.

Fonti e riferimenti

Buona parte del materiale grezzo che ha dato vita a questo libro, proviene da cinque anni di lavoro al progetto Subversion (http://subversion.tigris.org/). Subversion è un sistema open source per il versionamento di un insieme di cartelle e documenti, scritto da zero, al fine di sostituire CVS come standard *de facto* tra i prodotti della categoria, nella comunità degli sviluppatori open source. Il progetto fu avviato dal mio datore di lavoro, CollabNet (http://www.collab.net/), all'inizio del 2000. Grazie al cielo CollabNet capì sin dall'inizio come portarlo avanti in maniera veramente collaborativa, come l'unione di tanti contributi distribuiti. Si presentarono molti volontari sin da subito; oggi siamo arrivati a circa 50 sviluppatori attivi sul progetto, dei quali solo una minoranza sono impiegati di CollabNet.

Subversion è per molti versi un classico esempio di un progetto open source, e ho finito per impegnarmi su di esso più di quanto originariamente avessi previsto. In parte è stata una questione di praticità: ogni volta che avevo bisogno di un esempio di una situazione particolare, potevo richiamarne di solito uno relativo a Subversion dalla memoria. Ma è stata anche una questione di verifica. Benché io sia impegnato in altri progetti di software libero a vari livelli, e parli con amici e conoscenti coinvolti in molti altri, ho scoperto ben presto che scrivendo articoli era necessario accompagnare la teoria con la pratica. Non volevo fare affermazioni su eventi di altri progetti basandomi solo su ciò che mi era capitato di leggere nelle liste di discussione. Con Subversion, ad esempio, un approccio del genere fornirebbe soltanto la metà delle risposte giuste. Così quando raccolgo ispirazioni o esempi su un progetto del quale non ho una particolare esperienza diretta, ho imparato a parlare prima con qualcuno che possa chiarirmi i dubbi, per acquisire confidenza sull'argomento ed essere in grado di spiegare come vanno le cose.

Subversion è stato il mio lavoro per gli ultimi 5 anni, ma mi occupo di software libero da 12. Tra gli altri progetti che mi hanno ispirato per la stesura di questo libro, mi piace ricordare:

- L'editor di testo GNU Emacs, ed il relativo progetto della Free Software Foundation, all'interno del quale ho gestito alcuni piccoli pacchetti.

- Il Concurrent Versions System (CVS), al quale ho lavorato intensamente nel 1994 – 1995 con Jim Blandy, continuando ad essere sporadicamente coinvolto da allora.

- La collezione di progetti open source nota come Apache Software Foundation, in particolare l'Ambiente Portabile di Apache (APR) ed il server HTTP Apache.
- OpenOffice.org, il sistema di basi di dati della Berkeley, il sistema MySQL; non sono stato coinvolto in questi progetti personalmente, ma li ho osservati e, in alcuni casi, ho avuto l'occasione di parlare direttamente con gli sviluppatori.
- Il debugger GNU (GDB) (in generale).
- Il progetto Debian (in generale).

Quest'elenco non è certo completo. Un po' come la maggior parte dei programmatori open source, mantengo dei legami "deboli" con molti progetti differenti,giusto per avere un senso dello stato generale delle cose. Di sicuro non potrei elencarli qui tutti, ma non trascurerò di citarli ove opportuno, durante la trattazione.

Ringraziamenti

Per scrivere questo libro è stato necessario un tempo quattro volte maggiore di quello che avevo preventivato, e per la maggior parte di quel tempo sentivo come un grande pianoforte sospeso sulla mia testa, ovunque andassi. Senza l'aiuto di molte persone, non sarei stato in grado di completarlo evitando di impazzire.

Andy Oram, il mio editore alla O'Reilly, si è rivelato ciò che ogni scrittore sogna. Oltre a conoscere il tema in maniera approfondita, (ha suggerito lui molti dei temi), ha il dono prezioso di sapere ciò che si vuol dire, e quello di aiutare a trovare il modo giusto per dire una cosa. Per me è stato un onore lavorare con lui. Un ringraziamento va anche a Chuck Toporek per aver girato subito la proposta di questo lavoro ad Andy.

Brian Fitzpatrick ha corretto quasi tutto il materiale mentre lo scrivevo, il che non solo ha reso migliore la qualità del libro, ma mi ha stimolato a continuare la stesura nei momenti in cui avrei desiderato essere da qualsiasi parte tranne che davanti ad un computer. Ben Collins-Sussman e Mike Pilato hanno dato una mano, via via che il documento prendeva forma, e sono sempre stati felici di discutere — a volte a lungo — l'argomento che avrei trattato in un dato momento. Pure loro si accorgevano dei momenti di rallentamento, e con garbo mi stuzzicavano se necessario. Grazie, ragazzi.

Biella Coleman stava scrivendo la sua tesi, nello stesso periodo in cui io scrivevo questo libro. Lei sa cosa vuol dire star seduti a scrivere ogni santo giorno, e mi è stata d'esempio, oltre che offrirsi di ascoltare le mie discussioni. Biella inoltre è dotata di un affascinante punto di vista "antropologico" del movimento del software libero, che ha messo a mia disposizione per darmi idee e riferimenti che potessero tornarmi utili in questo libro. Alex Golub — un altro antropologo con un piede nel mondo del software libero, anche lui in procinto di finire la sua tesi in quel periodo — è stato un supporto prezioso sin dall'inizio, dando un grande aiuto.

Micah Anderson non si è mai mostrato particolarmente sopraffatto dal suo compagno di viaggio, che l'ha ispirato con una specie di comportamento invidioso, ma è stato sempre pronto con la sua amicizia, con le sue conversazioni, e con il suo supporto tecnico (almeno in un'occasione). Grazie, Micah!

Jon Trowbridge e Sander Striker mi hanno entrambi incoraggiato con aiuti concreti — la loro vasta esperienza nel software libero mi ha dato accesso a materiale che non avrei mai potuto ottenere in altro modo.

Grazie a Greg Stein non solo per l'amicizia e gli incoraggiamenti al momento giusto, ma anche per aver mostrato agli sviluppatori del progetto Subversion quanto sia importante la revisione periodica del codice nel costruire una comunità di sviluppatori. Grazie anche a Brian Behlendorf, che ha inculcato

sapientemente nelle nostre teste l'importanza di discutere pubblicamente un problema; spero che questo principio venga riportato lungo la trattazione di questo libro.

Grazie a Benjamin "Mako" Hill e Seth Schoen, per le belle chiacchierate sul software libero e le sue politiche; a Zack Urlocker e Louis Suarez-Potts per avermi concesso un'intervista durante il loro tempo libero; a Shane sulla lista di discussione *Slashcode* per avermi fatto riportare il suo intervento; e ad Haggen So per il suo enorme contributo di confronto tra vari servizi di *hosting* pronti all'uso.

Grazie a Alla Dekhtyar, Polina, e Sonya per il loro incoraggiamento paziente ed instancabile. Sono ben felice di non dover più finire presto la giornata (a volte senza aver concluso qualcosa) ed andare a casa a lavorare sul "Libro".

Grazie a Jack Repenning per l'amicizia, le chiacchierate, e l'ostinato rifiuto di accettare un'analisi semplicistica, quando ne intravedeva una più corretta, ma anche più difficile da spiegare. Spero che una parte della sua esperienza sia nel campo dello sviluppo software che in quello dell'industria del software, abbia dato maggior lustro a questo libro.

CollabNet si è mostrata eccezionalmente generosa nel concedermi un orario di lavoro flessibile per scrivere, e non ha avuto da ridire quando il lavoro si è protratto più a lungo del previsto. Non conosco le vie intricate attraverso le quali i dirigenti arrivano a prendere tali decisioni, ma ho il sospetto che Sandhya Klute, e più tardi Mahesh Murthy, ne sappiano qualcosa — il mio ringraziamento va ad entrambi.

L'intero gruppo di sviluppatori di Subversion è stato fonte di ispirazione negli ultimi cinque anni, e molto di quello che ho messo in questo libro proviene dalla mia esperienza lavorativa insieme a loro. Non posso ringraziarli uno per uno, perché ci vorrebbe quasi un capitolo solo per i loro nomi, ma prego ogni lettore che incontrasse uno sviluppatore di Subversione di offrirgli immediatamente qualcosa al bar — come farò sicuramente io.

Tante volte ho scocciato Rachel Scollon per informarmi sulla situazione del libro; mi ha sempre ascoltato, ed a volte ha reso i problemi più piccoli di quello che sembravano prima di parlarle. Anche questo mi è stato di grande aiuto — grazie.

Grazie (ancora) a Noel Taylor, che si sarà sicuramente chiesto perché io volessi scrivere un altro libro, visto quanto mi sono lamentato la volta precedente, ma la cua amicizia e appartenenza a Golosá mi ha aiutato ad avere un po' di buona musica e fratellanza nella mia vita, anche nei giorni più impegnati. Grazie anche a Matthew Dean e Dorothea Samtleben, amici e compagni "sofferti" di musica, che hanno saputo comprendermi man mano che le scuse per le assenze durante la pratica si accumulavano. Megan Jennings è stata un supporto costante, ed interessata in maniera spontanea all'argomento, sebbene le risultasse poco familiare — un vero tonico per uno scrittore insicuro. Grazie, cara.

Ho avuto quattro revisori competenti e precisi per questo libro: Yoav Shapira, Andrew Stellman, Davanum Srinivas, e Ben Hyde. Se fossi stato in grado di integrare tutti i loro eccellenti suggerimenti, questo sarebbe un libro migliore. Ma come capita sempre, i vincoli temporali mi hanno costretto a scegliere, sebbene i miglioramenti siano comunque significativi. Qualsiasi refuso rimasto, è imputabile esclusivamente a me.

I miei genitori, Frances ed Henry, sono stati un supporto meraviglioso come sempre, a visto che questo libro è meno tecnico del precedente, spero che lo troveranno un tantino più comprensibile.

In conclusione, vorrei ringraziare i destinatari della dedica, Karen Underhill e Jim Blandy. L'amicizia di Karen e la sua comprensione sono stati tutto per me, non solo durante la scrittura del libro, bensì negli ultimi sette anni. Semplicemente non sarei stato in grado di finire senza il suo aiuto. Analogamente per Jim, vero amico ed *hacker* di tutto rispetto, che mi ha introdotto al software libero, un po' come un uccellino che insegna ad un aereo a volare.

Esclusione di responsabilità

I pensieri e le opinioni espresse in questo libro sono del tutto personali. Non rappresentano necessariamente la visione di CollabNet o quella del progetto Subversion.

Capitolo 1. Introduzione

La maggioranza dei progetti di software libero fallisce.

Noi tendiamo a non dare molto ascolto alla notizia di questi fallimenti. Solo i progetti che hanno successo attraggono l'attenzione e ci sono tanti progetti di software libero in totale[1] che, anche se solo una piccola percentuale di essi ha successo il risultato è che a noi appare essere tuttavia una gran quantità. Inoltre noi non abbiamo notizia dei fallimenti perchè i fallimenti non fanno notizia. Non c'è un particolare momento in cui il progetto cessa di essere praticabile. La gente semplicemente sceglie di allontanarsene. Ci può essere un momento in cui un cambiamento finale viene fatto nel progetto, ma quelli che lo hanno fatto generalmente non sanno che quel cambiamento è l'ultimo. Non è facile stabilire quando il progetto si è esaurito. Quando non è stato lavorato per sei mesi? Quando la base di utilizzatori ha smesso di crescere senza aver superato la base di sviluppatori? E se gli sviluppatori di un progetto lo abbandonano perchè si rendono conto che stanno duplicando il lavoro di un altro?—e se essi si uniscono a quell'altro progetto dunque lo espandono per immettervi molto del loro sforzo primitivo? Il progetto precedente è finito o ha semplicemente cambiato casa?

A causa di una tale complessità è impossibile stabilire il numero dei fallimenti. Ma una evidenza annedotica da più di un decennio di open source, qualche ricerca su SourceForge.net e un piccolo googling tutti puntano alla stessa conclusione: il numero è estremamente alto, probabilmente dell'ordine del 90–95%. Il numero sale se includete i progetti che sopravvivono ma son disfunzionali: quelli che *stanno* producendo codice che gira ma che non hanno motivo di esistere, o che non stanno progredendo così velocemente o così affidabilmente come dovrebbero.

Questo libro parla di come evitare i fallimenti. Esso esamina non solo come fare le cose bene ma come farle sbagliate in modo che voi possiate riconoscere e correggere gli errori piuttosto in anticipo. La mia speranza è che dopo averlo letto voi abbiate un repertorio di tecniche non tanto per evitare i trabocchetti dello sviluppo open source quanto per operare per la crescita e la manutenzione di un progetto di successo. Successo non è un gioco senza perdite e questo libro non parla di come vincere o farsi strada nella competizione. Indubbiamente una importante parte dei progetti open source che girano sta funzionando regolarmente con altri progetti simili. Nel lungo periodo ogni progetto che ha successo contribuisce allo stato di benessere dell'insieme complessivo del software libero nel web.

Si sarebbe tentati di dire che i progetti di software libero falliscono per lo stesso tipo di motivi del software proprietario. Certamente il software libero non non è il solo a trovarsi di fronte a presupposti non realistici, specifiche vage, non buona amministrazione di risorse, non adeguata fase di progetto o alcuni degli altri spauracchi ben noti all'industria del software. C'è una enorme quantità di cose scritte su questi argomenti, e non tenterò di ripeterle in questo libro. Invece tenterò di descrivere i problemi peculiari del software libero. Quando i progetti di software libero vanno in secca ciò avviene perchè gli sviluppatori (o gli organizzatori) non si sono resi conto dei problemi peculiari dello sviluppo open source anche se essi possono essere ben consci delle meglio note difficoltà dello sviluppo closed source.

Uno dei più comuni errori è l'aspettativa circa i benefici dell'open source stesso. Una licenza open source non garantisce il fatto che orde di sviluppatori mettano subito a disposizione i loro tempo per il vostro progetto, nè l'open sourcing cura automaticamente i mali di un progetto che dà problemi. Di fatto è proprio l'opposto: l'aprire un progetto può aggiungere una serie di complessità, e costa nel breve periodo *più* che non mantenendolo chiuso. Aprirlo significa adattare il codice per renderlo comprensibile a persone completamente nuove, metter su un sito di sviluppo e liste email e spesso scrivere documentazione per la prima volta. Tutto ciò costa molto lavoro E certamente se alcuni sviluppatori interessati *lo aprono* c'è il carico aggiuntivo di rispondere alle domande dei nuovi arrivati

[1] SourceForge.net, sito popolare solo di hosting aveva 79.225 progetti registrati a metà Aprile 2004. Questo non è neanche lontanamente il numero dei progetti in Internet, certamente; è solo il numero di quelli che scelgono di usare SourceForge.

per un certo tempo prima di vedere i benefici della loro presenza. Come sviluppatore Jamie Zawinski sui primi giorni del progetto Mozilla disse:

> *L'open source funziona ma non è del tutto certo che sia una panacea. Se c'è una diceria che mette in guardia qui è che tu non puoi prendere un progetto che sta morendo, cospargerlo con la magica polvere dell'open source e ottenere che ogni cosa funzioni. Il software è difficile. I problemi non sono così semplici*
>
> (da **http://www.jwz.org/gruntle/nomo.html**)

Un errore correlato è quello di essere avari nella presentazione e nel confezionamento della relativa applicazione. La presentazione e il confezionamento dell'applicazione relativa richiedono una larga serie di compiti che girano tutti intorno al tema di ridurre la difficoltà di chi entra nel progetto. Rendere il progetto invitante per i non iniziati significa scrivere un documentazione per l'utente e per lo sviluppatore, metter su un sito che sia di informazione per i nuovi arrivati, automatizzare la compilazione e l'installazione quanto più sia possibile, ecc. Molti programmatori purtroppo trattano questo compito come se fosse secondario rispetto alla scrittura del codice. C'è una duplice ragione perchè questo avviene: in primo luogo questo può essere percepito come un lavoro di poco conto perchè i suoi benefici sono più visibili a coloro che hanno meno familiarità col progetto e viceversa. Dopotutto, coloro che sviluppano il progetto non hanno bisogno dell'applicazione relativa. Essi sanno già come installare, gestire e usare il software perchè lo hanno scritto. In secondo luogo le abilità richieste per fare una presentazione e confezionare la relativa applicazione sono spesso completamente differenti da quelle richieste per scrivere codice. Le persone tendono a focalizzarsi su ciò in cui essi sono brave anche se potrebbe servire di più al progetto spendere tempo su ciò a cui esse sono meno adatte. Capitolo 2, *Partenza* discute in dettaglio sulla presentazione e sul confezionamento dell'applicazione e spiega perchè è cruciale che essi abbiano la priorità in una una corretta partenza del progetto.

Poi viene la credenza errata che in un progetto open source occorra un una piccola o nessuna organizzazione o al contrario che funzionerà la stessa organizzazione di un progetto non aperto. L'organizzazione in un progetto open source non è sempre visibile, eccetto che nei progetti di successo, usualmente avviene dietro le quinte in una forma o nell'altra. Un esperimento in apparenza insignificante è sufficiente a mostrare come. Un progetto open source è fatto di un numero casuale di programmatori—una già nota categoria di liberi pensatori—che molto verosimilmente non si sono mai incontrati e che possono avere differenti obbiettivi nel lavorare al progetto. L'esperimento pensato consiste nell'immaginare cosa succederebbe ad un simile gruppo *senza* una organizzazione. A meno di un miracolo fallirebbe o scomparirebbe dalla vista molto rapidamente. Le cose semplicemente non girerebbero da se, per quanto noi vorremmo altrimenti. Ma l'organizzazione malgrado possa essere alquanto attiva è spesso irregolare, scarsa, riservata. La sola cosa che tiene unito un gruppo di sviluppatori è il pensiero condiviso che si possa fare di più in concerto che non individualmente. Così l'obiettivo dell'organizzazione è sopratutto assicurarsi che essi continuino a pensare questo, stabilendo standards di comunicazione, rendendo sicuramente utile che gli sviluppatori non siano emarginati a causa di peculiarità di comportamento, e in generale facendo in modo che per gli sviluppatori il progetto sia un luogo dove ritornare. Tecniche specifiche per far ciò sono discusse per il resto di questo libro.

Infine c'è una categoria generica di problemi che può essere chiamata "fallimenti della navigazione culturale" Dieci anni fa, anche cinque, sarebbe stato prematuro parlare di cultura globale del software libero, ma ora non più. Una cultura riconoscibile è lentamente emersa e mentre essa è certamente non monolitica è per lo meno soggetta al dissenso alla faziosità come una cultura geograficamente chiusa—essa ha un consistente nocciolo di base. Molti progetti di successo mostrano alcune o tutte le caratteristiche di questo nocciolo. Essi premiano certi tipi di comportamento e ne puniscono altri; essi creano un'atmosfera che incoraggia una partecipazione non pianificata, a volte a spese di un coordinamento centrale; essi hanno concetti di rudezza e garbo che possono differire sostanzialmente da quelli che prevalgono altrove. Ciò che è molto importante è il fatto che partecipanti veterani hanno fatto propri questi comportamenti, dimodochè hanno in comune un forte consenso sul comportamento

previsto. I progetti che falliscono deviano in modo significativo da questo nocciolo, sebbene senza intenzione, e spesso non hanno l'unanimità su ciò che costituisce un comportamento ragionevole non predefinito. Ciò significa che quando insorgono problemi la situazione può rapidamente deteriorarsi, giacchè i partecipanti non hanno quell'insieme di riflessi culturali per aiutarsi a risolvere i dissensi.

Questo libro è una guida pratica, non uno studio antropologico o una storia. In tutti i casi una valida conoscenza dell'odierna cultura del software libero è un fondamento essenziale per ogni consiglio pratico. Una persona che comprende la cultura può viaggiare in lungo e in largo per il mondo dell'open source, incontrando molte varianti in abitudini e linguaggi, sarà sempre capace di partecipare effettivamente e con agio ovunque. Invece una persona che non comprende la cultura troverà il processo dell'organizzazione e della partecipazione difficile e pieno di sorprese. Siccome il numero di persone che sviluppano software libero è sempre in crescita a balzi e rimbalzi c'è sempre gente in questa seconda categoria— questa è in maniera preponderante la cultura dei nuovi arrivati e le cose continueranno ad essere tali per qualche tempo. Se voi pensate di poter essere fra questi la prossima sezione dà le basi per discutere cose che incontrerete più tardi sia in questo libro sia in Internet. (Se invece avete lavorato con l'open source per un certo tempo forse conoscete molto della sua storia e di conseguenza vi sentirete liberi di saltare questa sezione).

Storia

La condivisione del software è esistita da quando è esistito il software . Nei primi giorni dei computers i costruttori vedevano che risultati positivi nella competizione si erano avuti principalmente nell'innovazione dell'hardware e quindi non dedicarono attenzione al software come risorsa commerciale. Molti dei primi clienti che usavano queste macchine erano scienziati o tecnici che avevano l'abilità di modificare e ampliare il software inviato con la macchina. I clienti a volte distribuirono le modifiche non solo ai costruttori ma anche agli altri possessori di macchine simili. I costruttori spesso tollerarono e incoraggiarono ciò: ai loro occhi miglioramenti al software di qualunque origine rendevano la macchina più allettante per gli altri potenziali acquirenti.

Anche se questo periodo iniziale somigliava all'attuale cultura del software libero per diverse ragioni, differiva per due principali aspetti. Uno, c'era già una certa standardizzazione dell'hardware; era il tempo di una fiorente innovazione nel progetto dei computers, ma la diversità nelle architetture di calcolo voleva dire che ogni cosa era diversa da ogni altra. Cosicchè il software scritto per una macchina non avrebbe funzionato con un'altra. I programmatori si orientarono ad acquisire abilità con una certa architettura o famiglia di architettura (laddove oggi essi sarebbero più propensi ad acquistare esperienza in un linguaggio di programmazione o famiglia di linguaggio confidando nel fatto che la propria esperienza sarebbe trasferibile a qualsiasi hardware di calcolo a cui a loro capiti di lavorare). Poiché l'esperienza di una persona tendeva a essere specifica di un tipo di computer la loro accumulazione di esperienza fece sì che fosse preferibile a sè e ai suoi colleghi quel computer. Ci fu quindi da parte dei costruttori l'interesse a che si estendesse quanto più possibile il codice e la conoscenza della macchina.

Due, non c'era Internet. Sebbene ci fossero minori restrizioni legali sulla condivisione rispetto ad oggi i limiti erano più di carattere tecnico. I mezzi per spostare dati da un posto all'altro erano sconvenienti e scomodi, relativamente parlando. C'era qualche piccolo network buono per scambiare informazione fra coloro che erano impiegati in una stessa ricerca o compagnia. Ma rimanevano barriere da superare si si voleva scambiare con chiunque, ovunque fosse. Queste barriere *furono* superate in molti casi. A volte differenti gruppi stabilirono reciproci contatti autonomamente inviando dischi o nastri tramite posta terrestre e a volte i costruttori stessi servirono come centrali di scambio per le modifiche. Fu di aiuto anche il fatto che molti dei primi sviluppatori lavoravano alle università dove era previsto che uno pubblicasse le sue conoscenze. Ma la realtà fisica della trasmissione dati diceva che c'era sempre un impedimento alla condivisione, un impedimento proporzionale al tratto (fisico o organizzativo) che il software doveva percorrere. Una condivisione larga e priva di attriti, come la conosciamo oggi era impossibile.

L'ascesa del software proprietario e del software libero

Come conseguenza del fatto che l'industria maturava avvennero molti cambiamenti. La grande varietà del software prodotto diede strada ai pochi chiari vincitori—vincitori per la tecnologia superiore, per una organizzazione commerciale superiore, o una combinazione delle due cose. Allo stesso tempo, e non completamente in simultaneità, lo sviluppo dei cosiddetti linguaggi di “alto livello” volle dire che uno avrebbe potuto scrivere un programma una volta sola in un unico linguaggio e ottenere che esso fosse tradotto (“compilato”) in modo da poter girare su differenti tipi di computer. Le implicazioni di ciò non furono perse dai costruttori di hardware. Il cliente così poteva dedicarsi a un impegno maggiore nello sviluppo del software senza necessariamente limitarsi a farlo in relazione a una specifica architettura. Quando ciò si combinò con una graduale diminuzione delle differenze di prestazione dei vari computers man mano che i progetti meno efficienti furono fatti fuori un costruttore che considerasse il suo hardware come sua unica risorsa poteva prevedere un futuro di margini di guadagno in diminuzione. Da sola la potenza di calcolo stava diventando un bene commerciale e il software faceva la differenza. Vendere software o almeno trattarlo come parte della vendita di hardware cominciò a sembrare una buona strategia.

Ciò significò che i costruttori dovettero incominciare a rafforzare i diritti d'autore sul loro codice in maniera più rigorosa. Se gli utilizzatori avessero continuato a scambiarsi il codice e a condividene i cambiamenti liberamente, essi avrebbero potuto reimplementare alcuni dei miglioramenti ora venduti come “valore aggiunto” dal fornitore. Peggio, il codice condiviso avrebbe potuto finire nella mani dei concorrenti. L'ironia è che questo stava succedendo nel tempo che Internet stava emergendo. Proprio quando il software veramente non impedito stava finalmente diventando tecnicamente possibile, i cambiamenti nell'affare dei computers lo rendeva indesiderabile almeno dal punto di vista di ogni singola compagnia. I fornitori misero un freno sia negando l'accesso agli utilizzatori al codice che girava sulle proprie macchine sia insistendo su un accordo di non apertura che rende il codice condivisibile.

Deliberata resistenza

Mentre il mondo del codice libero perdeva colore lentamente una reazione prendeva forma nella mente di almeno un programmatore. Richard Stallman aveva lavorato nell' Artificial Intelligence Lab al Massachusetts Institute of Technology negli anni 70 e nei primi anni 80 periodo che si rivelò essere un periodo d'oro e il luogo un luogo d'oro per la condivisione del codice. L'Artificial Intelligence Lab aveva una forte “etica hacker”,[2] e la gente non solo era incoraggiata ma voleva fortemente condividere ogni miglioramento avesse apportato al sistema. Come Stallman scrisse poi:

> *Noi non chiamavamo il nostro software software libero perchè quel termine non esisteva ancora; ma quello era. Quando persone di un'altra università o compagnia volevano trasferire su un'altra piattaforma e usare un programma noi eravamo contenti di permetterglielo. Se voi aveste visto uno usare un programma non familiare ed interessante avreste potuto sempre chiedere di vedere il codice sorgente in modo da poterlo leggere, cambiarlo o usare parti di esso per costruire un altro programma.*
>
> (da **http://www.gnu.org/gnu/thegnuproject.html**)

Questa comunità da paradiso terrestre si esaurì intorno a Stallmann bruscamente dopo il 1980 quando i cambiamenti che erano avvenuti nel resto dell'industria raggiunsero alla fine anche l'Artificial Intelligence Lab. Una nuova compagnia impiegò molti programmatori del Lab per farli lavorare a un sistema operativo simile a quello su cui avevano lavorato nel Lab, ora solamente sotto una licenza esclusiva. Contemporaneamente l' l'Artificial Intelligence Lab acquistò una nuova apparecchiatura che mise a disposizione di un nuovo sistema operativo proprietario.

[2] Stallman usa la parola “hacker” per indicare “chi ama programmare ed è abile nel farlo” non nel relativamente nuovo significato di “chi entra nei computers”.

Stallmann tracciò un chiaro saggio di quello che stava accadendo:

> *I moderni computers dell'epoca come il VAX o il 68020 avevano i loro sistemi operativi ma nessuno dei due era software libero: tu dovevi firmare un accordo di non rivelazione persino per prendere una copia dell'eseguibile.*
>
> *Ciò significava che la prima cosa nell'usare un computer era quella di non aiutare il tuo vicino. Una comunità operativa era proibita. La regola stabilita dai proprietari del software era "Se tu condividi col vicino sei un pirata. Se tu vuoi un cambiamento chiedi umilmente a noi."*

Per qualche bizzarria nel carattere lui decise di resistere alla tendenza. Invece di continuare a lavorare presso l'ora decimato Lab o di prendere lavoro come scrittore di codice presso una delle nuove compagnie dove i risultati di questo lavoro sarebbero stati tenuti chiusi in una scatola si licenziò dal Lab e iniziò il progetto GNU dando il via alla Free Software Foundation (FSF). Lo scopo del GNU [3] era quello di sviluppare un sistema operativo completamente libero e parte principale di software applicativo al quale a coloro che lo usavano non sarebbe stato impedito di accedere e di condividerne le modifiche. Egli stava in sostanza rimettendo in piedi ciò che era stato distrutto dall'Artificial Intelligence Lab ma in una scala mondiale e senza quella vulnerabilità che aveva reso la cultura dell'Artificial Intelligence Lab suscettibile di disintegrazione.

Oltre a lavorare al nuovo sistema operativo Stallman lasciò in eredità una licenza di copyright i cui termini garantivano che il suo codice sarebbe stato libero per sempre. La The GNU General Public License (GPL) è un pezzo geniale di judo legale: essa dice che il codice può essere copiato e modificato senza restrizioni e che le copie e i lavori derivati (cioè le versioni modificate) devono essere distribuite sotto la stessa licenza originale senza nessuna restrizione aggiuntiva. Nei fatti essa usa la legge sul copyright per raggiungere l'effetto opposto di quello del copyright tradizionale: invece di limitare la distribuzione del software faceva in modo che nessuno, compreso l'autore, potesse limitarla. Per Stallman questo era meglio che porre semplicemente il suo codice sotto pubblico dominio. Se esso fosse stato di pubblico dominio una copia particolare avrebbe potuto essere incorporata in un software proprietario come si era anche appreso avvenire sotto licenze di copyright permissive. Mentre tale incorporazione non avrebbe potuto compromettere in nessun modo la continua disponibilità del codice, avrebbe potuto avere il significato che gli sforzi di Stallman avrebbero potuto beneficiare il nemico software proprietario. La GPL può essere pensata come una forma di protezionismo per il software libero perchè impedisce al software proprietario di trarre pieno vantaggio dal codice GPL. La GPL e le sue relazioni con le altre licenze libere è discussa in dettaglio in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti*.

Con l'aiuto di molti programmatori, molti dei quali condividevano le idee di Stallman e alcuni dei quali volevano semplicemente vedere una gran quantità di software libero, il progetto GNU incominciò a rilasciare molti ricambi dei componenti più critici di un sistema operativo. A causa della ora assai diffusa standardizzazione nell'hardware e nel software dei computers fu possibile usare i ricambi GNU su sistemi per il resto non liberi. E molti lo fecero. L'editor di testi GNU (Emacs) e il compilatore C (GCC) ebbero particolare successo guadagnando seguaci numerosi e fedeli, non per le loro origini ideologiche ma semplicemente per il loro meriti tecnici. Dal 1990 circa GNU ha prodotto molti sistemi operativi, eccetto il kernel—la parte che attualmente avvia la macchina e che è responsabile della memoria operativa, dei dischi e di altre risorse di sistema.

Sfortunatamente il progetto GNU aveva scelto un progetto di kernel che si dimostrò essere più difficile da implementare di quanto ci si fosse aspettato. Il ritardo che ne risultò impedì alla Free Software Foundation di creare la prima release di un intero sistema operativo libero. Il pezzo finale fu invece messo a punto da uno studente finlandese della scienza dei computers che con l'aiuto di volontari per il mondo aveva completato un kernel libero usando un progetto più tradizionale. Lo chiamò Linux

[3] Esso sta per "GNU's Not Unix" e il "GNU" in quella espansione sta per ...la stessa cosa.

e quando questo fu combinato con i programmi GNU esistenti il risultato fu un sistema operativo completamente libero. Per la prima volta si sarebbe potuto avviare un computer e lavorare senza alcuno dei software proprietari. [4]

Molto software su questo sistema operativo non fu prodotto dal progetto GNU. Infatti GNU non era nemmeno l'unico gruppo che lavorava ad un sistema operativo libero (per esempio il codice che alla fine diventò NetBSD e FreeBSD era già in sviluppo in questo periodo). L'importanza della Free Software Foundation stava non solo nel codice che essi scrivevano ma nella loro retorica politica. Col parlare del software libero come un ideale anzichè come una convenienza essi resero difficile per i programmatori *non* avere una consapevolezza di ciò. Persino quelli che si dissociarono dalla Free Software Foundation ebbero da affrontare il problema, se prendere esclusivamente una posizione differente. L'efficacia della propaganda della Free Software Foundation poggiò sull'associare il suo codice a un messaggio per mezzo della GPL e di altri testi. Il suo messaggio si diffuse alla stessa maniera in cui si diffuse il suo codice.

Imprevista resistenza

C'erano molte altre cose che stavano succedendo sulla scena del software libero e poche erano tanto esplicitamente ideologiche quanto il progetto GNU di Stallman. Una delle più importanti era la *la Berkeley Software Distribution* (*BSD*), una reimplementazione graduale del sistema operativo Unix —che fino agli ultimi anni '70 era stato un vago progetto di ricerca all' AT&T—da programmatori dell'Università della California a Berkley. Il gruppo del BSD non fece nessuna dichiarazione per unirsi e condividere ma *praticò* l'idea con predisposizione ed entusiasmo col coordinamento di un massiccio impegno di sviluppo distribuito in cui le utility command line di Unix e le librerie di codice e lo stesso kernel del sistema operativo furono riscritti da un abbozzo per lo più da volontari. Il progetto BSD divenne un primo esempio di sviluppo di software libero non ideologico e servì anche come base di addestramento per molti sviluppatori che avrebbero voluto andare avanti per rimanere attivi nel mondo dell'open source.

Un'altra dura prova per lo sviluppo cooperativo fu il *X Window System*, un ambiente grafico di calcolo condiviso in rete sviluppato al MIT nella metà degli anni '80 in partnership coi venditori di hardware che avevano il comune interesse di poter offrire ai loro clienti un sistema a finestre. Lontano dal contrastare il software proprietario, la licenza X permetteva aggiunte proprietarie in cima a un corpo libero—ogni membro del consorzio voleva la chance di aumentare la distribuzione X di default e quindi di trarre un vantaggio nella competizione con gli altri membri. X Windows[5] di per sé era software libero ma principalmente per spianare il campo di gioco a interessi commerciali in concorrenza, non senza un qualche desiderio di porre fine al dominio del software proprietario. Ancora un altro esempio che anticipò il GNU project di alcuni anni fu TeX, sistema libero di tipocomposizione della qualità di pubblicazione di Donald Knuth. Egli la rilasciò sotto una licenza che permetteva a chiunque di modificare e distribuire il codice, ma non sotto il nome di “TeX” a meno che non avesse superato una molto rigorosa serie di tests di compatibilità (questo è un esempio della classe di licenze "a protezione del marchio" di cui si parla ancora in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti*). Knut non stava prendendo una posizione o l'altra sul problema software libero o software proprietario, egli voleva solo un sistema di tipocomposizione per completare il suo *reale* compito—un libro sulla programmazione—e non vide nessuna ragione per non rilasciare il suo sistema a cose finite.

Senza enumerare ogni progetto e ogni licenza si può dire di sicuro che dagli ultimi '80 ci fu molto software libero disponibile sotto una grande varietà di licenze. La diversità delle licenze rifletteva la diversità delle motivazioni. Anche alcuni dei programmatori che avevano scelto la GNU GPL erano molto meno spinti da motivi ideologici di quanto lo fosse il GNU project. Anche se erano contenti

[4]Tecnicamente Linux non era il primo. Un sistema operativo libero per computers IBM compatibili, chiamato 386BSD era venuto fuori poco prima di Linux. Comunque sarebbe stato molto più difficile rifinire 386BSD e farlo girare. Linux così fece tale colpo non solo perchè era libero ma perchè aveva realmente una grande possibilità di avviare il vostro computer quando lo aveste installato.
[5]Essi preferiscono essere chiamati "X Window System", ma in pratica, la gente usualmente li chiama "X Window” perchè tre parole sono proprio troppo scomode.

di lavorare al software libero molti sviluppatori non consideravano il software proprietario un male sociale. Ci fu gente che sentì l'impulso morale di liberare il mondo del "software hoarding" (termine di Stallman per software non libero) ma altri erano motivati più da sollecitazioni tecniche o dal piacere di lavorare con collaboratori della stessa opinione o persino da un un semplice umano desiderio di gloria. Eppure tutto sommato queste disparate motivazioni non interagirono in modo distruttivo. Ciò è dovuto in parte al fatto che il software, a differenza della altre forme creative come la prosa o le arti visive, deve superare dei tests per metà oggettivi per essere considerato di successo. Ciò dà a tutti i partecipanti a un progetto una specie di automatica base comune, una ragione e una struttura per lavorare insieme senza tanta preoccupazione di qualificazioni oltre quella tecnica.

Gli sviluppatori ebbero un'altra ragione per unirsi. Risultò che il mondo del software libero stava producendo qualche codice di alta qualità. In alcuni casi esso era in modo dimostrabile tecnicamente superiore all'alternativa non libera. In altri esso era almeno comparabile e certamente costava meno. Mentre solo alcuni erano motivati a far girare sul computer il software libero per motivi strettamente filosofici, una grande quantità di persone era contenta di farlo perchè svolgeva meglio un compito. E quelli che lo usavano erano spesso intenzionati a spendere tempo e mettere a disposizione capacità per rendersi utili a svolgere compiti di manutenzione e miglioramento del software.

Questa tendenza a produrre buon codice non era universale ma si stava affermando con crescente frequenza in progetti di software libero in giro per il mondo. Imprese che trattavano principalmente software incominciarono gradualmente ad averne notizia. Molte di esse scoprirono che stavano già usando software libero nelle operazioni giornaliere solo che semplicemente non lo avevano saputo (la parte superiore dell'organizzazione spesso non è a conoscenza di ciò che il dipartimento IT fa). Le grandi imprese incominciarono ad assumere un ruolo sempre più attivo nei progetti di software libero, contribuendo con tempo e mezzi e a volte persino finanziando lo sviluppo di programmi liberi. Tali investimenti nel migliore degli scenari avrebbero potuto ripagarli molte volte di seguito. Lo sponsor paga solo un piccolo numero di programmatori esperti affinchè essi si dedichino al progetto a tempo pieno ma raccoglie i benefici del contributo di ognuno incluso il lavoro non pagato dei volontari e dei programmatori pagati da altre grandi imprese.

"Free" e "open source" a confronto

Nella misura in cui le grosse imprese guardavano con sempre più attenzione al software libero i programmatori erano messi di fronte ai nuovi problemi della presentazione. Uno era la parola “free” in se stessa. In un primo momento nell'ascoltare la parola “free” molta gente pensava erroneamente che questa significava semplicemente software a "costo zero". E' vero che tutto il software libero è a costo zero, ma non tutto il software a costo zero è libero.[6] Per esempio, durante la battaglia dei browsers negli anni '90 sia Microsoft che Netscape diedero via i propri browsers in competizione gratis, nella zuffa per accaparrarsi un fetta di mercato. Nessuno dei due browsers era libero nel senso di “software libero”. Non si poteva ottenere il codice sorgente e anche si si fosse potuto ottenere non si aveva il diritto di modificarlo e redistribuirlo. [7] L'unica cosa che si poteva fare era scaricare un eseguibile e farlo girare sul computer. I browsers erano non più liberi di un software incartato comprato in un negozio. Essi avevano soltanto un prezzo abbastanza basso.

Questa confusione sulla parola “free” è dovuta a una sfortunata ambiguità nella lingua inglese. Molte altre lingue distinguono fra prezzi bassi e libertà (la distinzione fra *gratis* e *libre* è chiara immediatamente a coloro che parlano una lingua neolatina, per esempio. Ma a causa della posizione della lingua inglese come lingua ponte di fatto di Internet, un problema con la lingua inglese è in una certa misura un problema per chiunque. L'incomprensione sulla parola “free” era così prevalente che i programmatori di software libero alla fine elaborarono una formula standard nella risposta: “E'

[6]Uno può imporre un costo alle copie di software libero distribuite, ma dal momento che non può impedire al destinatario di offrire il software a costo zero dopo il prezzo è forzato a scendere completamente a zero,

[7]Il codice sorgente di Netscape Navigator *fu* alla fine rilasciato come licenza open source nel 1998 e diventò il fondamento di Mozilla. Vedere http://www.mozilla.org/.

libero nel senso di in *in regime di libertà*—il pensiero *libertà di parola*, non il pensiero *birra gratis*." Comunque doverlo spiegare ancora e ancora è stancante. Molti programmatori ebbero la sensazione, non ingiustificata, che l'ambiguità della parola “free” stava ostacolando la comprensione di questo software.

Ma il problema diventò più profondo di così. La parola “free” portò con sè una inevitabile connotazione morale. Se in effetti la libertà era un fine, non era importante il fatto che che potesse succedere che il software libero fosse migliore oppure più profittevole per certi affari in certe circostanze. Quelli erano graditi effetti secondari di una causa che non era in fondo né tecnica né mercantile, ma morale. Inoltre la collocazione di “libero nel senso di in libertà” evidenziò una chiara incoerenza delle grosse imprese che da un lato dei propri affari volevano supportare particolari programmi liberi mentre dall'altro continuavano a commerciare software proprietario.

Sopraggiunsero questi dilemmi per una comunità che si trovava già in sospeso per una crisi di identità. I programmatori che realmente *scrivevano* software libero non erano stati mai unanimi su tutti gli obiettivi, se ce n'erano, del movimento del software libero. Persino dire che che le opinioni vanno da un estremo all'altro sarebbe fuorviante, giacchè ciò farebbe erroneamente pensare che esso sia in una zona lineare invece che in una dispersione multidimensionale. Comunque si possono distinguere due categorie di pensiero, se vogliamo ignorare le sottigliezze per il momento. Un gruppo fa proprie le vedute di Stallman che la libertà di condividere e modificare è la cosa più importante e che quindi se si finisce di parlare di libertà si perde il cuore della questione. Altri sentono che il software è l'argomento più importante in favore del software stesso e trovano scomodo proclamare che il software proprietario è cattivo per il fatto di essere software proprietario. Alcuni, ma non tutti i programmatori pensano che l'autore (e chi lo impiega, nel caso di lavoro pagato) *dovrebbe* avere il diritto di controllare termini della distribuzione e che non c'è bisogno di dare nessun giudizio morale sulla scelta dei particolari termini.

Per lungo tempo non ebbero bisogno di essere esaminate e coordinate ma il nascente successo del software libero nel mondo del business rese il problema ineludibile. Nel 1998 il termine *open source* fu creato in alternativa a “libero” da una colizione si programmatori che alla fine diventarono l' Open Source Initiative (OSI).[8] L'OSI si convinse non solo che il “software libero” era potenzialmente portatore di confusione ma che la parola “libero” era appunto un sintomo di un problema generale: che il movimento aveva bisogno di un programma di marketing per agganciarsi al mondo delle grandi imprese, e che il discorso della morale e di benefici sociali non avrebbe mai preso il volo al tavolo dei consigli di amministrazione. Nelle loro parole:

> *La Open Source Initiative è un programma di marketing per il software libero. Esso è un montare per il software libero su basi pragmatiche piuttosto che su una barca ideologica che fa rumore. La sostanza prevalente non è cambiata, la tendenza a perdere e il simbolismo si. ...*
>
> *L'involucro che deve essere fatto per molte tecniche non è il concetto di open source ma il nome. Perchè non chiamarlo, come facevamo, software libero?*
>
> *Una ragione diretta è che il termine “software libero” è facilmente mal compreso in modi che portano al conflitto. ...*
>
> *Ma la vera ragione del cambiamento di etichetta e di tipo commerciale. Stiamo cercando di agganciare la nostra concezione a quella del mondo delle grosse imprese. Noi abbiamo un prodotto vincente ma il nostro posizionarsi in passato è stato spaventoso. Il termine “software libero” è stato mal interpretato dagli uomini d'affari che scambiarono il desiderio di scambiare con anti commercialismo, o peggio con imbroglio.*
>
> *Le principali corporazioni CEOs e CTOs non compreranno mai il “software libero”. Ma se noi facciamo nostra proprio la stessa tradizione, la stessa gente e e le stesse*

[8] La web home dell OSI è http://www.opensource.org/.

> *licenze di software libero e ne cambiamo l'etichetta a “open source”? Così, essi lo compreranno.*
>
> *Alcuni esperti di computer troveranno questo difficile da credere, ma ciò accade perchè essi sono tecnici che pensano in concreto, in termini di sostanza e non capiscono quanto sia importante l'immagine quando si vende qualcosa.*
>
> *Nel marketing l'apparenza è realtà. L'apparenza che noi vogliamo togliere le barricate e che stiamo volendo lavorare con il mondo delle grosse imprese conta quanto la realtà del nostro comportamento, delle nostre convinzioni e del nostro software.*
>
> (da http://www.opensource.org/. O piuttosto, *in passato* da quel sito — l' OSI ha chiaramente preso le pagine sin d'allora, sebbene esse possano essere viste ancora a http://web.archive.org/web/20021204155057/http://www.opensource.org/advocacy/faq.php and http://web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/case_for_hackers.php#marketing.)

Le punte dei molti icebergs della controversia sono visibili in questo testo. Esso si riferisce alle “nostre convinzioni” ma evita abilmente di parlare di cosa esattamente siano le nostre convinzioni. Per qualcuno potrebbe essere la convinzione che il codice sviluppato in accordo con un processo aperto sarà un codice migliore; per altri potrebbe essere la convinzione che tutte le informazioni debbano essere condivise. C'è l'uso della parola “furto” per riferirsi (presumibilmente) alla copia illegale. Uso su cui molti hanno da ridire, sulla base del fatto che non è furto se tuttavia il possessore ne ha la notizia dopo. C'è il fatto che è allettante lasciar credere che il movimento del software libero potrebbe erroneamente essere accusato di anti comercialismo, ma si lascia non attentamente esaminata la questione se una tale accusa abbia di fatto una base.

Nessuna delle quali cose voleva dire che il sito dell'OSI era incoerente o ingannevole. Piuttosto era un esempio di quello che esattamente l'OSI dichiarava essere stato perso dal movimento del software libero: il “buon marketing” dove “buon” sta per “attuabile ne mondo degli affari”. L'Open Source Initiative diade a tanta gente esattamente quello di cui erano andati in cerca: un vocabolario per parlare del software libero come metodologia di sviluppo del software e di strategia di affari, non come una crociata morale.

La comparsa della Open Source Initiative cambiò il panorama del software libero. Essa formalizzò la dicotomia che era stata non nominata a lungo e nel fare ciò forzò il movimento a prendere atto del fatto che essa aveva un politica sia interna che esterna. L'effetto oggi è che ambedue le parti hanno dovuto cercare un comune terreno dato che molti progetti includono programmatori di ambedue i campi e anche partecipanti che non si inseriscono in nessuna categoria. Ciò non significa che la gente non parla mai di motivazioni morali gli errori nell'etica tradizionale di coloro che volevano abbattere certi limiti furono nominati ad alta voce, per esempio. Ma era raro per uno sviluppatore di software libero/opensource farsi domande sulle motivazione degli altri in un progetto. Il contributo vince su chi contribuisce. Se uno scrive del buon codice tu non gli chiedi se lo fa per ragioni morali, o perchè il suo datore di lavoro lo paga, o perchè si sta costruendo un curriculum vitae o per qualcos'altro. Tu valuti il software su basi tecniche e su basi tecniche dai il responso. Anche organizzazioni esplicitamente politiche come il progetto Debian, il cui obiettivo era quello di offrire un ambiente di calcolo libero al 100% (cioè “libero nel senso di in libertà”) era abbastanza permissivo nell'integrarsi con codice non libero e nel cooperare con programmatori che non condividevano esattamente gli stessi obiettivi.

La Situazione Oggi

Quando fate girare un progetto di software libero avete bisogno di non parlare di questioni filosofiche su una base giornaliera. I programmatori non insistono sul fatto che ogni altro nel progetto sia

concorde con le proprie vedute su tutte le cose (quelli che insistono su questo si trovano rapidamente incapaci a partecipare a qualunque progetto). Ma voi avete bisogno di essere a conoscenza del fatto che la questione del software “libero” contro l' “open source” esiste, in parte per evitare di dire cose che potrebbero essere ostili a qualche partecipante, e in parte perchè capire le motivazioni dei programmatori è il miglior modo—in un certo senso, il *solo* —di portare avanti un progetto.

Il software libero è una cultura per scelta. Per operare con successo in esso dovete capire in primo luogo perchè la gente sceglie di stare in esso. Tecniche coercitive non funzionano. Se la gente non è contenta in un progetto andrà verso un altro progetto. Il software libero è degno di attenzione anche fra le comunità di volontari per la sua chiarezza di impiego. Molta della gente coinvolta non ha mai incontrato gli altri partecipanti faccia a faccia e dona ritagli di tempo quando gli va a genio. Il passaggio normale attraverso il quale gli uomini si uniscono gli uni agli altri si restringe a un sottile canale: le parole scritte che viaggiano sui fili elettrici. A causa di ciò può passare molto tempo prima che si formi un gruppo coeso e specializzato. Viceversa è molto facile che si perda un potenziale volontario nei primi momenti in cui si fanno le conoscenze. Se un progetto non dà una buona prima impressione, raramente i nuovi arrivati danno una seconda chance.

La precarietà, o piuttosto la potenziale precarietà delle relazioni è forse la più scoraggiante incombenza nel portare avanti un progetto. Cosa persuaderà tutta questa gente a tenersi unita abbastanza a lungo da poter produrre qualcosa di utile? La risposta a questa domanda è complessa abbastanza da occupare il resto di questo libro, ma se dovesse essere espressa in un modo di dire essa sarebbe questa:

> *La gente dovrebbe sentire che il suo legame col progetto e la sua influenza su di esso è direttamente proporzionale a suo contributo.*

Nessuna classe di sviluppatori o potenziali sviluppatori si sentirebbe svenduta o discriminata per ragioni non tecniche. Chiaramente i progetti con la sponsorizzazione delle grandi compagnie e/o gli sviluppatori salariati hanno bisogno di essere particolarmente attenti a riguardo come Capitolo 5, *I Soldi* discute in dettaglio. Certamente ciò non significa che se non si ha la sponsorizzazione non c'è da preoccuparsi. I soldi sono solo uno dei fattori che possono influire sul successo di un progetto. Ci sono anche domande su quale linguaggio scegliere, quale licenza, quale procedimento di sviluppo, quale tipo di infrastruttura metter su, come pubblicizzare l'inizio di un progetto nei fatti e molto altro. Partire con un progetto col piede giusto è l'argomento del prossimo capitolo. .

Capitolo 2. Partenza

Il modello classico di avvio di un progetto di software libero fu fornito da Eric Raymond su un saggio oggi famoso sui metodi dell'open source intitolato *La Cattedrale e il Bazaar*. Egli scriveva:

> *Ogni buon lavoro di software nasce dall'atto dello sviluppatore di grattarsi un prurito personale.*
>
> (da **http://www.catb.org/~esr/writings/cathedral-bazaar/**)

Da notare che Raymond non stava dicendo che l'open source si ha quando qualche individuo ha prurito. Piuttosto stava dicendo che *il buon* software nasce quando il programmatore ha interesse a vedere risolti i problemi. La rilevanza di ciò per il software libero era che il prurito personale si rivelava essere la più frequente motivazione nel far partire un progetto di software libero.

Questo è ancora oggi il modo in cui i progetti liberi partono, ma meno oggi che nel 1997, quando Raymond scriveva queste parole. Oggi abbiamo il fenomeno di organizzazioni, incluse quelle che lo fanno per profitto—che danno il via a grossi progetti open source centralizzati organizzativamente. Il programmatore solitario che batte del codice per risolvere un problema locale e che poi si rende conto che il risultato ha una larga applicabilità è ancora la sorgente di molto nuovo software libero, ma non è la sola storia.
La condizione essenziale è che i produttori abbiano un interesse diretto al suo successo perchè lo usano essi stessi. Se il software non fa quello che si suppone faccia l'organizzazione o la persona che lo produce sentirà una insoddisfazione nel suo lavoro giornaliero. Per esempio, il progetto OpenAdapte (http://www.openadapter.org/), che fu avviato dalla banca di investimenti Dresdner Kleinwort Wasserstein come base open source per integrare disparati sistemi di informazione finanziaria, può a mala pena dirsi un progetto che gratta il prurito di un programmatore solitario. Esso gratta un prurito istituzionale. Ma quel prurito vien fuori dall'esperienza dell'istituzione e dei suoi partners, e quindi se il progetto fallisce nell'alleviare il loro prurito essi lo sapranno. Questo congegno produce buon software perchè il giro di ritorno va nella giusta direzione. Il programma non viene scritto per essere venduto a qualche altro in modo che questi possa risolvere *il suo* problema. Esso viene scritto per risolvere *il proprio* problema di qualcuno, quindi viene condiviso con chiunque, più o meno come se il problema fosse una malattia e il software una medicina la cui distribuzione intende sradicare completamente l'epidemia.

Questo capitolo tratta di come mettere al mondo un nuovo software, ma molte delle sue raccomandazioni suoneranno familiari a una organizzazione della sanitÃ distributrice di medicine. Gli obiettivi sono molto simili: voi volete chiarire ciÃ² che la medicina fa, la mettete nelle mani delle persone giuste e vi assicurate che quelli che la ricevono sappiano usarla, ma col software voi volete attirare alcuni dei destinatari a unirsi al tentativo di migliorare la medicina.

Il software ha bisogno di acquisire utilizzatori e di acquisire sviluppatori. Queste due necessità non sono necessariamente in conflitto, ma aggiungono complessità alla presentazione iniziale del progetto. Qualche informazione è utile per ambedue le categorie, qualcuna è utile solo per l'una o solo per l'altra. Tutti e due i tipi di informazione dovrebbero dare un contributo al principio di una informazione adattata. Cioè il grado di dettaglio fornito ad ogni stadio dovrebbe corrispondere direttamente alla quantità di tempo e di sforzo immessovi dal lettore. Maggiore sforzo dovrebbe essere sempre uguale a maggior ricompensa. Quando le due cose non sono correlate fermamente la gente può velocemente perdere la fiducia e abbandonare il tentativo.

Il corollario a ciò è così *la questione dell'apparenza*. Ai programmatore spesso non piace pensare a ciò. Il loro amore per la sostanza più che per la forma è quasi un punto di vanto professionale. Non è un caso che così tanti programmatori mostrino una antipatia per il lavoro di marketing e di pubbliche relazioni

né che i creatori di grafica professionale siano inorriditi di fronte a quello che i programmatori fanno pensare per conto proprio.

Questo è un peccato, perchè ci sono situazioni in cui la forma *è* sostanza, e la presentazione è una di quelle. Per esempio la primissima cosa di cui un visitatore viena a conoscenza circa un progetto è l'aparenza del suo sito web. Questa informazione è assorbita prima di quello che realmente vi è contenuto—di ogni testo sia stato letto o dei links cliccati. Per quanto ingiusto possa essere ciò, la gente non può astenersi dal formarsi una prima impressione. L'apparenza di un sito web dà un segno di quanta cura è stata messa nell'organizzare la presentazione di un progetto. Gli uomini hanno hanno antenne molto sensibili nel captare l'impiego di attenzione. Molti di noi possono dire a un prima occhiata se un sito è stato messo insieme disordinatamente o se è stato pensato seriamente. Questo è il primo pezzo di informazione che il vostro progetto dà e l'impressione che esso crea si trasferirà al resto del progetto per associazione.

Così mentre molta parte di questo capitolo parla del contenuto con cui il vostro progetto potrebbe partire, ricordatevi della questione del look e anche delle impressioni. Siccome il sito del progetto ha a che fare con due tipi di visitatori—utilizzatori e sviluppatori—una attenzione speciale si deve fare alla chiarezza e a chi è diretto. Sebbene questo non sia il luogo per una trattazione generale del web design, un principio è meritevole di menzione, specialmente quando il sito serve a molti tipi di pubblico (anche se e una sovrapposizione): la gente deve avere una grezza idea di dove il link va, prima di cliccarlo. Per esempio dovrebbe essere ovvio *nel guardare ai links* che portano alla documentazione utente che essi portano alla documentazione utente, e non, per esempio, alla documentazione sviluppatore. Il mandare avanti un progetto consiste in parte nel fornire informazioni, ma anche nel dare comodità. La sola presenza di certe offerte standard, in luoghi determinati, riassicura gli utilizzatori e gli sviluppatori che stanno decidendo se essere coinvolti. Ciò vuol dire che questo progetto ha la caratteristica di procedere insieme, anticipa le domande che la gente farà e ha fatto lo sforzo di rispondere loro in modo che è richiesto un minimo sforzo da parte di chi fa le domande. Creando questa atmosfera di preparazione, il progetto dà questo messaggio: “Il vostro tempo non sarà sprecato se sarete coinvolti”, che è esattamente la cosa che la gente vuole sentirsi dire.

Ma Prima, Guardiamoci Intorno

Prima di partire con un nuovo progetto c'è un importante avvertimento:

Sempre guardarci intorno per vedere se c'è un progetto esistente che fa ciò che noi vogliamo. Sono molto buone le possibilità che un problema che volete risolvere ora, qualcun altro lo ha voluto risolvere prima di voi. Se lo hanno risolto e hanno rilasciato il loro codice sotto una licenza libera, allora non c'è motivo per voi di reinventare la ruota oggi. Ci sono eccezioni, certamente: se volete avviare un progetto per una esigenza educativa un codice preesistente non vi aiuterebbe; oppure può essere che il progetto che avete i mente è così specializzato che voi sapete che non c'è alcuna possibilità che qualcuno lo abbia fatto. Ma generalmente non c'è nessun motivo per non guardare e la ricompensa può essere enorme. Se la ricerca sui motori di ricerca di Internet non dà risultato, cercate su http://freshmeat.net/ (un sito di notizie sui progetti open source di cui si parlerà molto, più in là), su http://www.sourceforge.net/, nella Free Software Foundation's directory a http://directory.fsf.org/.

Anche se non trovate esattamente ciò che state cercando, potreste trovare qualcosa così vicina che avrebbe più senso unirsi a quel progetto e aggiungervi funzionalità, che partire da un vostro abbozzo.

Partire Da Ciò Che Si Ha

Vi siete guardati intorno, trovato che niente fuori soddisfa veramente i vostri bisogni, e avete deciso di partire con un nuovo progetto.

Cosa viene ora?

La parte più difficile nel lanciare un nuovo progetto è trasformare una visione privata in una visione pubblica. Voi nella vostra organizzazione potete conoscere alla perfezione ciò che volete, ma esprimere chiaramente al mondo l'obiettivo è una chiara quantità di lavoro. E' essenziale, comunque, che vi prendiate il tempo per farlo. Voi e gli altri fondatori dovete decidere su che cosa sarà in realtà il progetto—cioè, stabilire i suoi limiti, ciò che *non* farà allo stesso modo di ciò che farà—e scrivere una dichiarazione delle vostre intenzioni. Questa parte non è usualmente troppo difficile, sebbene essa può rivelare supposizioni non dette e anche disaccordi sulla natura del progetto, che è cosa buona: meglio risolvere queste cose ora che più tardi. Il passo successivo è confezionare il progetto per il pubblico consumo, e questa è, fondamentalmente, una vera e propria sgobbata.

Ciò che lo rende così laborioso è che esso consiste principalmente nell'organizzare e documentare cose che ognuno conosce già—“ognuno”, cioè, coloro che sono stati finora coinvolti nel progetto. Cosicchè per coloro che fanno il lavoro non c'è un immediato beneficio. Essi non hanno bisogno di un file `README` che dia una panoramica, né di un documento di progetto o di un manuale utente. Essi non hanno bisogno di un albero del codice messo in ordine con cura, conforme agli informali ma assai diffusi standards delle distribuzioni di codice sorgente. Ad ogni modo il codice sorgente messo a punto è buono per loro perchè vi sono avvezzi comunque, e se il codice non gira affatto, essi sanno come usarlo. E non ha importanza, per essi, se i presupposti fondamentali dell'architettura del progetto rimangono non documentati; essi invece sono familiari con esso.

I nuovi arrivati, d'altra parte, hanno bisogno di queste cose. Fortunatamente essi non ne hanno bisogno tutti in una volta. Non è necessario per voi fornire ogni possibile risorsa prima di rendere pubblico un progetto. In un mondo perfetto, forse, ogni nuovo progetto open source prenderebbe vita con una completa documentazione della fase di progettazione, un manuale utente completo (con speciali note su funzionalità pianificate ma non ancora implementate), bel codice confezionato in modo che sia trasportabile, capace di girare su ogni piattaforma di elaborazione, e così via. In realtà prendersi cura di tutti questi disparati fini sarebbe una proibitiva perdita di tempo e, comunque, è lavoro in cui uno può ragionevolmente sperare di essere aiutato da volontari, una volta che il progetto sia avviato.

Ciò *che è* necessario, comunque, è che un sufficiente impiego di energie venga attuato nella presentazione che i nuovi arrivati possono trovare dopo dopo la difficoltà o la non familiarità iniziale. Pensate a ciò come ad un primo passo di un processo che si sta avviando, per tenere il progetto a un specie di minima energia di attivazione. Ho sentito che questa soglia veniva chiamata la *hacktivation energy*: la quantità di energia che il nuovo arrivato deve immettere prima di incominciare ad entrare in possesso di qualcosa. Più piccola è l'energia di attivazione di un progetto, tanto meglio è. Il vostro primo compito è tenere l'energia di attivazione bassa, a un livello che incoraggi la gente a farsi coinvolgere.

Ciascuna delle sottosezioni seguenti descrive un aspetto importante nell'avvio di un nuovo progetto. Esse sono presentate approssimativamente nell'ordine in cui un nuovo visitatore le incontrerebbe, sebbene certamente l'ordine in cui voi in realtà le implementate potrebbe essere diverso. Voi potete trattarle come una lista da spuntare. Quando avviate un progetto andate fino in fondo alla lista e vi assicurate di aver incluso tutti le voci, o almeno che siate sereni sulle potenziali conseguenze di averne lasciata fuori una.

Scegliere Un Buon Nome.

Mettetevi nei panni di qualcuno che ha appena saputo del vostro progetto, forse per essersi imbattuto in esso alla ricerca di un software che risolvesse il suo problema. La prima cosa che egli incontra è il nome del progetto.

Un bel nome non renderà il vostro progetto un progetto di successo e un brutto nome non lo rovinerà;—beh, un nome *veramente* brutto potrebbe farlo, ma noi partiamo dall'ipotesi che nessuno stia cercando di far fallire il proprio progetto. Comunque, un brutto nome può rallentare l'adozione di un progetto, sia perchè la gente non lo prende seriamente, sia perchè semplicemente ha difficoltà a ricordarlo.

Un nome bello:

- : dà un'idea di ciò che il progetto fa, o almeno vi è correlato un modo chiaro, cosicchè se uno conosce il nome e conosce quello che il progetto fa, il nome verrà subito in mente dopo.
- E' facile da ricordare. Qui, ecco non c'è da girare intorno al fatto che l'inglese è la lingua predefinita di Internet: “facile da ricordare” significa ”facile da ricordare per qualcuno che sa leggere l'inglese”. Nomi che sono dipendenti da giochi di parole della pronuncia nativa, per esempio, saranno poco chiari a molti lettori non nativi di inglese là fuori. Se il gioco di parole suscita particolare interesse ed è particolarmente memorizzabile può ancora valere la pena di usarlo; solo mettetevi in testa che molta gente vedendo il nome non lo sente nella testa allo stesso modo di chi parla l'inglese come nativo.
- Non è come con altri nomi di progetti e non viola il marchio. Questa è solo una buona maniera e buona anche in senso legale. Voi non volete creare confusione di identità. E' abbastanza difficile tener traccia di ciò che è disponibile in rete già ora, senza cose differenti che hanno lo stesso nome.

  L risorse menzionate prima in sezione chiamata «Ma Prima, Guardiamoci Intorno» Sono utili a scoprire se progetto ha lo stesso nome a cui state pensando voi. Libere ricerche di marchi liberi sono disponibili a http://www.nameprotect.org/ e http://www.uspto.gov/.
- Se possibile, sia disponibile come nome del dominio `.com`, `.net`, e `.org` domini di primo livello. Voi dovreste sceglierne uno, forse `.org`, da annunciare come sito ufficiale del progetto; gli altri due doverebbero condurre lì e sono un modo semplice per impedire a terze parti di creare confusione sul nome del progetto. Anche se volete hostare il progetto su un altro sito (vedere sezione chiamata «L' Hosting In Scatola»), potete ugualmente registrare domini specifici per il progetto e redirigerli al sito ospitante. Ciò aiuta molto gli utilizzatori ad avere un URL facile da ricordare.

Avere una chiara dichiarazione di intenti

Una volta che avete trovato il sito web del progetto, la cosa successiva che la gente cercherà è una breve ma veloce descrizione, una dichiarazione di intenti in modo da poter decidere (in 30 secondi) se è interessata a saperne di più. Questa deve essere messa in evidenza in prima pagina, preferibilmente sotto al nome del progetto.

La dichiarazione di intenti deve essere concreta, limitativa, e soprattutto breve. Qui c'è n'è un esempio di una buona, da http://www.openoffice.org/:

> *Per creare, come comunità, la swite leader a livello internazionale per ufficio che gira su tutte le principali piattaforme e che fornisce accesso a tutte le funzionalità e ai dati grazie a un componente basato sulle API e a un formato di file basato sull'XML.*

Giusto in poche parole, essi hanno centrato tutti i punti principali prendendo largamente ispirazione dalla conoscenza precedente dei lettori. Dicendo "*come comunità*", essi segnalano che nessuna grossa impresa dominerà lo sviluppo; "*internazionale*" significa che il software permetterà alla gente di lavorare in molti lingue e luoghi; "*Tutte le principali piattaforme*" significa che esso sarà trasportabile su Unix, Macintosh, e Windows. Il resto segnala che le interfacce aperte e i formati facilmente comprensibili sono una parte importante dell' obiettivo. Essi non vengono direttamente allo scoperto e dicono che stanno cercando di essere una alternativa libera all'Office della Microsoft, ma molta gente può verosimilmente leggere tra le righe. Sebbene questa dichiarazione di intenti sembri ampia ad una prima occhiata, nei fatti è piuttosto circoscritta: le parole "*swite per ufficio*" hanno il significato di qualcosa di molto concreto a quelli che sono familiari con questo software .Inoltre la presunta precedente conoscenza del lettore (in questo caso probabilmente di Office della Microsoft) è usata per mantenere concisa la dichiarazione di intenti.

La natura di una dichiarazione di intenti dipende in parte da chi la sta scrivendo, non dal software che descrive. Per esempio per Openoffice.org ha senso usare le parole "*come una comunità*", perchè

il progetto fu avviato ed è ancora sponsorizzato da Sun Microsystems. Includendo quelle parole Sun manifesta la sua sensibilità alle preoccupazioni circa il fatto che essa potrebbe dominare il processo di sviluppo. Con una cosa di questo tipo, dimostrando unicamente la consapevolezza del *potenziale* per un problema va lontano nell'evitare del tutto il problema. D'altra parte i progetti che non sono sponsorizzati da una singola grande impresa non hanno bisogno probabilmente di un simile linguaggio; dopotutto lo sviluppo da parte di una comunità è la norma, sicchè non ci sarebbe ragione di metterlo in lista come parte degli intenti.

Specificare che il Progetto è Libero

Quelli che restano interessati dopo aver letto la dichiarazione di intenti vorranno poi vedere più dettagli, forse qualche documentazione utente o sviluppatore e eventualmente vorranno scaricare qualcosa. Ma prima di ognuna di queste cose essi vorranno essere sicuri che è open source.

La pagina principale deve rendere inequivocabilmente chiaro che quel progetto è open source. Ciò può sembrare ovvio, ma sareste sorpresi dal fatto di quanti progetti dimenticano di farlo. Ho visto tanti siti di progetti di software libero dove la pagina principale non solo non diceva sotto quale licenza particolare il software era distribuito, ma nemmeno specificava chiaramente che il software era libero. A volte il dato fondamentale informativo era relegato nella pagina dei downloads, o nella pagina degli sviluppatori, o in qualche altro posto che richiedeva un ulteriore clic del mouse per arrivarvi. In casi estremi la licenza non era inserita affatto nel sito web e l'unico modo per vederla era quello di scaricare il software e di guardarvi dentro.

Non fate questo errore. Tale omissione può far perdere molti potenziali sviluppatori e utilizzatori. Specificate in modo aperto, giusto sotto la dichiarazione di intenti che il progetto e “software libero” e “open source” e fornite la licenza esatta. Una guida rapida alla scelta della licenza è data in sezione chiamata «Scegliere una Licenza e Applicarla» più in là in questo capitolo , e le questioni sulle licenze sono discusse in dettaglio in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti*.

A questo punto i nostri ipotetici visitatori hanno deciso—probabilmente in un minuto o meno—che sono interessati a spendere, per esempio, almeno cinque ulteriori minuti nell' esaminare questo progetto. Le prossime sezioni descrivono ciò che essi potrebbero incontrare in quei cinque minuti.

Elenco dell Caratteristiche e dei Requisiti

Ci potrebbe essere una lista delle funzionalità del software (se qualcosa non è ancora completa potete ugualmente metterla in lista ma mettete "*pianificato*" o "*in corso*" vicino ad essa), e il tipo di ambiente di elaborazione richiesto dal software. Pensate alla lista di funzionalità/requisiti come a qualcosa che dareste a uno che chiedesse un sommario veloce del software. E' spesso giusto una logica espansione della dichiarazione di intenti. Per esempio la dichiarazione di intenti potrebbe dire:

> *Per creare un indice full-text e un motore di ricerca con una ricca API ad uso dei programmatori nel fornire servizi di ricerca per grosse quantità di files di testo.*

L'elenco delle caratteristiche e dei requisiti darebbe i dettagli, chiarendo la portata della dichiarazione di intenti :

> *Caratteristiche:*
>
> - *Testo di ricerca non formattato, HTML, e XML*
> - *Ricerca di una parola o di una frase*
> - *(pianificata) Ricerca diffusa*

- *(pianificata) Aggiornamento incrementale degli indici*
- *(pianificata) Indicizzazione dei siti web*

Requisiti:

- *Python 2.2 o superiore*
- *Sufficiente spazio su disco da contenere gli indici (approssimativamente 2x dimensione dei dati)*

Con queste informazioni i lettori possono velocemente farsi un'idea di quanto questo software abbia speranza di funzionare per loro e possono pensare di farsi coinvolgere anche come sviluppatori.

Lo Stato dello Sviluppo

La gente vuole sempre sapere quanto il progetto sta facendo. Per i nuovi progetti, essi vogliono sapere il ritardo fra le promesse del progetto e la corrente realtà. Per i progetti maturi vogliono sapere quanto è attiva è la sua manutenzione, quanto spesso escono le nuove releases, come probabilmente si reagisce ai rapporti dei bugs, ecc..

Per rispondere a queste domande dovreste metter su una pagina dello stato dello sviluppo con l'elenco degli obiettivi a breve termine del progetto e delle sue necessità (per esempio, si potrebbe essere alla ricerca di sviluppatori con una particolare esperienza). La pagina può anche fornire una storia della passate releases, con gli elenchi delle caratteristiche, cosicchè i visitatori possano farsi un'idea di come il progetto definisce l'“avanzamento” e di quanto velocemente avanza in accordo con tale definizione.

Non vi spaventate se vi vedete impreparati, e non siate tentati di esagerare sullo stato di sviluppo. Chiunque sa che il software si evolve per stadi; non c'è da vergognarsi a dire “Questo è il software alfa con bugs noti. Esso gira, e funziona almeno per qualche tempo, ma lo usate a vostro rischio” Un tale linguaggio non farà fuggire per lo spavento il tipo di sviluppatori di cui voi avete bisogno in quello stadio. Per quanto riguarda gli utilizzatori, una delle peggiori cose che un progetto può fare è attrarre utilizzatori prima che sia finito. Una reputazione per l'instabilità o per gli errori è difficile da scrollarsela di dosso, una volta che se la si è fatta. La prudenza paga a lungo andare. E' sempre meglio che il software sia *più* stabile di quanto gli utilizzatori si aspettino che meno stabile, e le sorprese piacevoli producono il più bel tipo di voci.

Alpha e Beta

Il termine *alfa* usualmente vuol dire una prima release con la quale gli utilizzatori possono accedere al lavoro reale fatto, e che ha tutte le funzionalità progettate, ma che ha anche bugs conosciuti. Il proposito principale del software alfa è quello di generare una reazione cosicchè gli sviluppatori conoscano ciò su cui lavorare. Il successivo stadio, beta, è quello in cui in cui tutti i bugs seri sono stati corretti ma non è stato testato abbastanza per certificarsi come release. Il proposito di un software beta è quello di diventare una release, nell'ipotesi che nessun bug sia stato trovato, ma anche quello di dare una risposta dettagliata agli sviluppatori in modo che essi possano giungere velocemente alla release. La differenza fra alfa e beta e molto una questione di giudizio.

Downloads

Il software dovrebbe essere scaricabile come codice sorgente in formato standard. Quando il progetto sta inizialmente prendendo avvio il pacchetto binario (eseguibile) non è necessario, amenochè il software

abbia i requisiti di costruzione o aspetti annessi talmente complicati che il semplice prelevarlo per farlo girare sarebbe un gran lavoro per molta gente. (Ma in questo caso il progetto è in un brutto periodo per attrarre sviluppatori comunque!)

Il meccanismo di distribuzione dovrebbe essere agevole, standardizzato e il meno astratto possibile. Se steste cercando di sradicare un male, non distribuireste la medicina in modo tale che essa richieda una grandezza della siringa non standard per somministrarla. Allo stesso modo il software doverebbe conformarsi a metodi standard di costruzione e di installazione; più esso devia dagli standards più i potenziali utilizzatori e sviluppatori abbandoneranno e andranno via confusi.

Il che suona ovvio, ma molti progetti non si infastidiscono a standardizzare le loro procedure di installazione prima che sia molto tardi nel gioco, dicendo che possono farlo in qualsiasi momento: *"Sistemeremo tutte quelle cose quando il codice sarà più vicino ad essere pronto."* Ciò di cui non si rendono conto è che rimandando il lavoro noioso di portare a termine le procedure di costruzione e di installazione, in realtà stanno facendo in modo che il codice tardi ulteriormente ad essere pronto—perchè scoraggiano gli sviluppatori che altrimenti avrebbero contribuito al codice. Molto insidiosamente, essi non *sanno* che stanno perdendo tutti quegli sviluppatori, perchè il processo è accumulazione di non eventi: qualcuno visita il sito, scarica il software, cerca di svilupparlo, fallisce, abbandona e va via. Chi mai saprà che ciò è accaduto, al di fuori delle stesse persone? Nessuno di quelli che lavorano al progetto si renderà conto che che l'interesse e la buona volontà di qualcuno è stata dissipata.

Un lavoro noioso con un'alta ricompensa dovrebbe essere fatto all'inizio e abbassando significativamente l'ostacolo di presentare un progetto con una buona confezione porta una ricompensa molto alta.

Quando rilasciate un pacchetto scaricabile è di vitale importanza che voi diate un numero univoco di versione alla release, così la gente può confrontare due release e sapere quale sostituisce l'altra. Una trattazione dettagliata sulla numerazione delle versioni può essere trovata in sezione chiamata «Numerazione delle Releases», e i dettagli sulle procedure di standardizzazione della costruzione e dell'installazione sono contemplate in sezione chiamata «Impacchettamento» , e anche in Capitolo 7, *Confezione, Rilascio, e Sviluppo Quotidiano* .

Controllo Versione e Accesso al Tracciamento Bug

Lo scaricamento dei pacchetti sorgente è piacevole per coloro che vogliono giusto installare e usare il programma, ma non è abbastanza per coloro che vogliono correggerne i bug e aggiungere nuove qualità. Istantanee notturne del sorgente possono aiutare, ma non c'è ancora una struttura sufficientemente a grani piccoli per una comunità che sta crescendo. La gente vuole un accesso in tempo reale agli ultimi codici sorgente e il modo per darglielo è usare un sistema di controllo versione. La presenza di versione dei sorgenti anonimamente controllate di è un segno a utilizzatori e sviluppatori—cioè questo progetto sta facendo uno sforzo per dare alla gente quello di cui ha bisogno per partecipare. Se voi non potete dare un controllo di versione subito, allora date un segno che intendete darlo presto. Della infrastruttura del controllo di versione si parla in dettaglio in sezione chiamata «Controllo di versione» in Capitolo 3, *L'Infrastruttura Tecnica* .

La stessa cosa per il tracciamento bug del progetto. L'importanza di un sistema di tracciamento bug sta non solo nella sua utilità per gli sviluppatori ma in quello che significa per chi osserva il progetto. Per molta gente una database accessibile dei bug è un fortissimo segnale che il progetto dovrebbe essere preso seriamente. Inoltre, più alto è il numero di bugs nel database, più il progetto sembra migliore. Ciò potrebbe sembrare contrario alla logica, ma ricordate che il numero dei bugs registrati in realtà dipende da tre cose: il numero in assoluto presente nel software, il numero di utilizzatori che usano quel software, e la comodità con cui questi utilizzatori possono registrare nuovi bugs. Di questi tre fattori gli ultimi due sono più significativi del primo. Un software di sufficiente complessità e grandezza ha

essenzialmente un numero arbitrario di bugs che devono essere scoperti. La vera domanda è, quanto il progetto fa per registrare e elencare in ordine di priorità questi bugs? Un progetto con un grosso e ben mantenuto database dei bugs (i bugs significativi ricevono una risposta prontamente, i bugs duplicati vengono unificati, ecc..) quindi dà una impressione migliore di un progetto senza una database dei bugs o di un database quasi vuoto.

Certo, se il vostro progetto sta giusto partendo, il database dei bugs conterrà assai pochi bugs, e non c'è molto che voi possiate fare su questo. Ma se la pagina dello stato mette in evidenza la giovinezza del progetto e se la gente guardando il database può vedere che molte archiviazioni sono avvenute di recente, può dedurre da ciò che il progetto ha ancora una salutare velocità di archiviazioni, e non sarà ingiustamente allarmata dal basso numero di bugs registrati.

Notare che i tracciatori di bug sono spesso usati per tracciare non solo i bug di software, ma anche richieste di crescita, cambiamenti di documentazione di, operazioni pendenti, e altro. I dettagli di un tracciatore di bugs in funzione sono visibili in sezione chiamata «Tracciamento dei bug» in Capitolo 3, *L'Infrastruttura Tecnica* , così io non entrerò in essi qui. La cosa importante dal punto di vista di una presentazione è giusto *avere* un tracciamento dei bugs e assicurarsi del fatto che sia visibile nella pagina principale.

I Canali di Comunicazione

I visitatori di solito vogliono sapere come raggiungere le persone coinvolte nel progetto. Fornite gli indirizzi delle mailing lists, delle chat, dei canali IRC e dei forums dove le altre persone interessate al software possano essere raggiunte. Rendete chiaro che voi e gli altri autori del progetto siete iscritti a queste mailing lists cosicchè la gente possa vedere che c'è un modo per far sapere che raggiungeranno gli sviluppatori. La vostra presenza nelle liste non implica un impegno a rispondere a tutte le domande o a implementare tutte le funzionalità future. Alla fine molti utilizzatori probabilmente non si uniranno mai ai forum comunque, ma saranno confortati dal fatto di sapere che *potrebbero* unirsi se caso mai ne avessero bisogno.

Nei primi stadi del progetto non c'è bisogno di avere forums separati per utilizzatori e sviluppatori. E' molto meglio che tutti siano spinti a parlare del software fra loro in un'unica “stanza”. Fra i primi arrivati la distinzione fra utilizzatori e sviluppatori è spesso sfocata. Nell'ambito in cui la distinzione può essere fatta il rapporto sviluppatori/utilizzatori è di solito molto più alto nei primi giorni del progetto che non in seguito. Mentre si può supporre che ognuno dei primi che aderiscono sia un programmatore che vuole cimentarsi col software, allo stesso tempo si può supporre che egli sia almeno interessato alle successive discussioni sullo sviluppo e a cogliere il senso degli indirizzi del progetto.

Sebbene questo capitolo parli solo della partenza di un progetto, è solo sufficiente dire che c'è bisogno questi forums di comunicazione esistano. Più in là, in sezione chiamata «Gestire la Crescita» in Capitolo 6, *Comunicazione* , esamineremo come e dove metter su tali forums i modi in cui essi potrebbero aver bisogno di moderazione o di altra amministrazione e come separare i forums per utilizzatori dai forums per sviluppatori, quando viene il momento, senza creare un abisso insuperabile.

Linee Guida per lo Sviluppatore

Se qualcuno sta pensando di contribuire al progetto egli cercherà le linee guida per gli sviluppatori. Le linee non sono tanto tecniche quanto sociali: esse spiegano come gli sviluppatori interagiscono fra di loro e con gli utilizzatori e infine come si portano a termine le cose.

Questo argomento è trattato in dettaglio in sezione chiamata «Metter Giù Tutto Per Iscritto» in Capitolo 4, *L'Infrastruttura Sociale e Politica* , ma gli elemento base delle linee guida sono:

- gli indici dei forums per l'interazione con gli altri sviluppatori

- istruzioni su come riportare i bugs e inviare la patches
- alcune istruzioni su *come* lo sviluppo viene generalmente fatto—il progetto è una benevola dittatura, una democrazia o qualcos'altro

Il termine “dittatura” non è inteso in senso peggiorativo, per inciso. E' un perfetto okay al realizzarsi di una tirannia dove un particolare sviluppatore ha il potere di veto su tutti i cambiamenti. Molti progetti di successo funzionano in questo modo. L'importante è che il progetto venga fuori e si fa per dire così. Una tirannia che pretenda di essere una democrazia disgusterà la gente. Una tirannia che dice di essere una tirannia andrà bene finchè il tiranno è competente e fidato.

Vedere http://subversion.apache.org/docs/community-guide/ per un esempio completo di linee guida, o http://www.openoffice.org/dev_docs/guidelines.html per linee guida che mettono a fuoco più l'amministrazione e lo spirito di partecipazione e meno le questioni tecniche.

Il problema separato di fornire al programmatore una introduzione al software è trattato in sezione chiamata «La documentazione sviluppatore» più in là in questo capitolo .

La documentazione

La documentazione è fondamentale. Bisogna che la gente abbia *qualcosa* da leggere, anche se questo qualcosa è rudimentale è incompleto. Ciò cade in pieno nella categoria “sgobbata” cui si è fatto riferimento prima ed è spesso la prima zona dove un nuovo progetto open source fallisce. Venir fuori con una dichiarazione di intenti, e con un elenco delle caratteristiche, scegliere una licenza, fare il punto sullo stato dello sviluppo—questi sono compiti relativamente piccoli che possono essere completati definitivamente e in genere non c'è bisogno di ritornarci su. La documentazione, invece, non è mai realmente finita, il che può essere un motivo per cui la gente a volte tarda proprio a iniziarla.

La cosa più insidiosa è che l'utilità della documentazione per quelli che la scrivono è inversamente proporzionale a quella di coloro che la leggeranno. La più importante documentazione per gli utilizzatori iniziali sono le basi: come installare velocemente il software, una panoramica su come funziona, magari qualche guida per eseguire alcune operazioni. Queste sono esattamente le cose che coloro che *scrivono* la documentazione conoscono troppo bene— così bene che può essere difficile per loro vedere le cose dal punto di vista del lettore e parlare con impegno dei passi che (a coloro che la scrivono) sembrano ovvi e non meritevoli di menzione.

Non esiste una soluzione magica a questo problema. Qualcuno giusto ha bisogno di sedersi e scrivere la materia, e poi farla funzionare da parte degli utilizzatori tipici per verificare la sua qualità. Usate un formato facile per le modifiche come l'html, il testo semplice, il Textinfo o qualche variante dell'XML —qualcosa che sia adatto per facilità, veloci miglioramenti a botta calda. Questo solo per rimuovere una qualsiasi cosa all'inizio che potrebbe impedire a coloro che originariamente hanno scritto la documentazione di apportare successivi miglioramenti, ma anche a coloro che si aggregano dopo al progetto e che vogliono lavorare alla documentazione.

Una maniera per assicurarsi che una documentazione di base iniziale sia portata a termine è limitare la sua portata in precedenza. In quel modo lo scriverla almeno non sembrerà come un compito indeterminato. Una buona regola pratica è quella che soddisferà i seguenti criteri:

- Dire al lettore quanta esperienza tecnica ci si aspetta da lui.
- Dire chiaramente e in modo esauriente come configurare il software e, in qualche posto vicino all'inizio della documentazione dire all'utilizzatore come attuare una sorta di test diagnostico o un semplice comando che confermi che ha configurato correttamente le cose. Una documentazione sulla configurazione è in una certa maniera più importante della documentazione sul vero e proprio utilizzo. Quanto maggiore sarà lo sforzo che uno avrà investito nell'installare e far partire il software, tanto maggiore sarà la sua tenacia nello scoprire funzionalità avanzate che non sono ben documentate.

Quando la gente abbandona, abbandona all'inizio; perciò sono i primissimi stadi, come l'installazione, che hanno bisogno del maggior supporto.

- Date un esempio stile tutorial di come svolgere una operazione comune. Ovviamente molti esempi per molte operazioni sarebbero sempre la cosa migliore ma se il tempo è limitato, prendete una sola operazione e spiegatela fino in fondo. Una volta che uno vede che il software *può* essere usato per una cosa partirà ad esplorare cos'altro può fare per sé stesso—e, se siete fortunati incomincerà a coinvolgersi nella documentazione. La qual cosa ci porta al prossimo punto...

- Segnatevi le aree in cui è noto che la documentazione è incompleta. Mostrando ai lettori che siete al corrente delle sue manchevolezze vi allineerete al loro punto di vista. La vostra capacità di comprenderli li rassicura che essi non sono di fronte al grande sforzo di convincere il progetto di quello che è importante. Non c'è bisogno che questi appunti rappresentino promesse di riempire i vuoti a partire da una data particolare—è ugualmente legittimo trattarli come aperte richieste di un aiuto di tipo volontario.

L'ultimo punto è di grande importanza, davvero, è può essere applicato all'intero progetto, non esattamente alla documentazione. Un accurato rendere conto delle manchevolezze note è la norma nel mondo dell'open source. Non dovete amplificare le manchevolezze del progetto, giusto identificatele scrupolosamente e spassionatamente quando il contesto lo richiede (nella documentazione, nel database del tracciamento dei bugs, o nelle discussioni nelle mailing lists). Nessuno prenderà ciò per disfattismo sulla parte di progetto, nè come un ordine di risolvere i problemi entro una determinata data, a meno che il progetto dia questo ordine esplicitamente. Siccome chiunque usa il software scoprirà le manchevolezze da sé, è molto meglio che sia psicologicamente preparato—allora sembrerà come se il progetto abbia una solida consapevolezza di come sta andando.

Manutenzione di una sezione FAQ

Una *FAQ* (Un documento“Domande fatte di frequente”) può essere uno dei migliori investimenti che il progetto fa in termini di ritorno didattico. Le FAQs sono molto in sintonia con le domande che gli utilizzatori e gli sviluppatori in realtà fanno—invece delle domande che voi avreste potuto *attendervi* da loro—e quindi una sezione di FAQs ben mantenuta tende a dare a coloro che la consultano esattamente quello che essi stanno cercando. La sezione FAQ è la prima cosa che guardano quando incontrano un problema, spasso anche preferendola al manuale ufficiale ed è probabilmente il documento nel vostro progetto che viene linkato dagli altri siti.

Sfortunatamente non potete fare le FAQ alla partenza del progetto. Della buone FAQs non sono scritte, esse vengono cresciute. Per definizione sono documenti di reazione, che si evolvono nel tempo in riposta all'uso che la gente fa del software giorno dopo giorno. Siccome è impossibile anticipare con successo le domande che la gente fa, è impossibile sedersi e scrivere delle utili FAQs dall'inizio.

Quindi non perdete tempo nel cercare di scriverle. Potete comunque trovare utile metter su un modello generalmente vuoto, di modo che quello sarà un posto dove la gente potrà contribuire con domande e risposte una volta che il progetto ha preso avvio. A questo punto la più importante qualità non è la completezza, ma la convenienza: se la sezione FAQ è facile da crescere, la gente la farà crescere. (Una corretta manutenzione delle FAQ è una questione non banale e avvincente ed è discussa ulteriormente in sezione chiamata «Il Manager delle FAQ» in Capitolo 8, *Gestire i Volontari* .)

Disponibilità della documentazione

La documentazione dovrebbe essere disponibile in due posti: online (direttamente dal sito), *e* e nella distribuzione scaricabile del software (vedere sezione chiamata «Impacchettamento» in Capitolo 7,

Confezione, Rilascio, e Sviluppo Quotidiano). C'è bisogno che sia online accessibile, via browser, perchè spesso la gente legge la documentazione prima di scaricare il software per la prima volta, come aiuto a decidere se scaricare punto. Ma essa dovrebbe accompagnare il software, sulla base del principio che il download dovrebbe fornire (cioè, rendere accessibile localmente) ogni cosa di cui uno ha bisogno per usare il pacchetto.

Per la documentazione online assicuratevi che ci sia un link che porti all' *intera* documentazione in una pagina html (mettete una nota tipo “monolitica” o “tutto in uno” o “singola grande pagina” vicino al link affinchè la gente sappia che potrebbe impiegare un periodo di tempo a caricarsi). Ciò è utile perchè spesso la gente vuol cercare una parola specifica o una frase nell'intera documentazione. In genere sanno già ciò che stanno cercando; essi non sanno solo in quale sezione è. Per tali persone niente è più frustrante dell'imbattersi in una pagina html per la tavola dei contenuti, poi in una pagina differente per l'istruzione, poi un'altra pagina per le istruzioni di installazione, ecc.. Quando le pagine sono spezzate così la funzione cerca del browser è inutile. Lo stile a pagine separate è utile per quelli che già sanno di quale sezione hanno bisogno, o che vogliono leggere la documentazione dall'inizio alla fine in sequenza. Ma questo *non* è il modo più comune di accedere alla documentazione. Molto più spesso chi è fondamentalmente familiare col software ritorna a cercare una specifica parola o frase. Fallire nel venire in aiuto ad essi con un unico documento ove si possa fare una ricerca renderebbe loro solo la vita più difficile.

La documentazione sviluppatore

La documentazione sviluppatore viene scritta per aiutare gli sviluppatori a capire il codice, cosicchè essi possano correggerlo ed estenderlo. Questa è in qualche modo differente dalle *linee guida sviluppatore* discusse prima, che sono più sociali che tecniche. Le linee guida sviluppatore dicono loro come andare d'accordo con il codice vero e proprio. Le due cose sono spesso impacchettate in un unico documento per convenienza (come con l' http://subversion.apache.org/docs/community-guide/ esempio dato prima), ma non hanno bisogno di esserlo.

Sebbene la documentazione sviluppatore possa essere molto utile non c'è motivo di tardare a rilasciare un permesso di farla. Finchè gli autori originari sono disponibili (e vogliono) rispondere alle domande sul codice, questo è sufficiente per partire. Infatti il dover rispondere alle stesse domande più e più volte è un motivo comune per scrivere la documentazione. Ma anche prima che essa sia scritta determinati collaboratori si daranno da fare per trovare una loro via per quanto riguarda il codice. La forza che guida la gente a spender tempo a imparare il codice base è che il codice è talvolta utile per se stessi. Se la gente ha fiducia in esso, si prenderà il tempo per capire le cose; se non ha questa fiducia nessuna quantità di documentazione li attirerà o li manterrà.

Così se avete tempo per scrivere la documentazione per un solo uditorio, scrivetela per gli utilizzatori. Tutta la documentazione utilizzatore è, in effetti, anche una documentazione sviluppatore; un programmatore che si sta accingendo a lavorare su una parte di software avrà bisogno di familiarizzare con il suo utilizzo. Più in là, quando vedrete programmatori che chiedono più e più volte le stesse cose, prendetevi il tempo per scrivere documenti separati giusto per loro.

Alcuni progetti usano un sito web, (o comunque una collezione di documenti ipertestuali) che può essere modificato dai suoi utilizzatori e i cui contenuti sono sviluppati in collaborazione da tutti coloro che ne hanno accesso, come in un forum (wiki). Nella mia esperienza ciò funziona se è attivamente aggiornato da poche persone che convengono su come la documentazione deve essere organizzata e che “voce” deve avere. Vedere in sezione chiamata «Wiki» in Capitolo 3, *L'Infrastruttura Tecnica* per maggiori ragguagli.

Emissione di dati e screenshoots di esempio

Se il progetto comporta una interfaccia utente grafica, o se produce uscite grafiche o diversamente risultati particolari, mettetene qualche esempio sul sito. Nel caso dell'interfaccia, ciò significa

screenshoots; nel caso di un risultato potrebbero essere screenshoots oppure veri e propri files. Ambedue le soluzioni fornirebbero alla gente una gratificazione istantanea: un singolo screenshoot può essere più convincente che paragrafi di testo descrittivo e di chiacchiere in mailing lists, perchè uno screenshot è una prova inconfutabile che il software *funziona*Può essere pazzo, può essere difficile da montare, può essere documentato in modo incompleto, ma questo screenshot è tuttavia una prova che se uno ci mette impegno, può ottenere che il software funzioni.

Screenshots

Dal momento che gli screenshoots possono scoraggiare finchè voi avete fatto in realtà poco, qui ci sono delle istruzioni di base per farli. Usando The Gimp (http://www.gimp.org/), aprire File->Acquisisci->Screenshot, scegliere Finestra singola o Schermo intero, poi cliccate OK. Ora il vostro prossimo click di mouse catturerà la finestra o lo schermo clicckato in una immagine in The Gimp. Tagliate e ridimensionate l'immagine se necessario usando le istruzioni in http://www.gimp.org/tutorials/Lite_Quickies/#crop.

Ci sono molte altre cose che potete mettere sul vostro sito, se ne avete il tempo, o se per una ragione o per l'altra esse sono specialmente adatte: una pagina di notizie, una pagina della storia del progetto, una pagina di links correlati, un sistema di ricerca nel sito, un link per le donazioni, ecc.. Nessuna di queste cose è una necessità all'avvio, ma tenetele in mente per il futuro.

L' Hosting In Scatola

Ci sono pochi siti che offrono un hosting gratis e una infrastruttura per i progetti open source: un'area web, il controllo di versione, un tracciatore di bugs, un'area di download, un foro di dibattito, backups regolari, ecc. I dettagli variano da sito a sito, ma vengono forniti gli stessi servizi base da tutti questi. Se usate uno di questi siti, avete molto gratis; ciò che cedete, ovviamente, è un controllo raffinato sull'esperienza degli utilizzatori. Il sevizio di hosting decide il software che gira sul sito, è può controllare o almeno influenzare l'aspetto e la percezione delle pagine web.

Vedere sezione chiamata «Canned Hosting» in Capitolo 3, *L'Infrastruttura Tecnica* per una più dettagliata discussione e per i vantaggi e svantaggi degli hosting in scatola e per un elenco di siti che lo offrono.

Scegliere una Licenza e Applicarla

Questa sezione vuol essere guida molto una veloce, molto scarna alla scelta di una licenza. Leggete Capitolo 9, *Licenze, Diritti d'Autore e Brevetti* per capire le implicazioni dettagliate di differenti licenze e come la licenza che scegliete possa avere ripercussioni sulla possibilità per la gente di mescolare il vostro software con altri software liberi.

C'è un gran numero di licenze fra cui scegliere. Non avete da prendere in considerazione molte di esse ora, dal momento che esse sono state scritte per soddisfare le esigenze legali di qualche grossa impresa o persona, e non sarebbe adatta per il vostro progetto. Restringeremo il campo solo alle più comuni licenze; nella maggior parte dei casi, voi vorrete scegliere una di esse.

Le Licenze “Fai Tutto”

Se vi fa comodo che il vostro progetto possa essere potenzialmente usato in programmi proprietari allora usate una licenza *MIT/X-style*. E' la più semplice delle licenze minimali che fanno poco più che affermare un diritto di copyright (senza nei fatti limitare la copia) e specifica che il codice arriva senza garanzia. Vedere sezione chiamata «La licenza MIT / X Window System » per i dettagli.

La GPL

Se non volete che il vostro codice sia usato in software proprietari usate la GNU General Public License (http://www.gnu.org/licenses/gpl.html). La GPL è probabilmente la più largamente apprezzata licenza nel modo oggi. Questo è già in se stesso un grosso vantaggio, perchè molti utilizzatori e collaboratori saranno già familiari con essa e quindi non avranno a spendere un tempo extra per comprendere la vostra licenza. Vedere sezione chiamata «La GNU General Public License» in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti* per i dettagli.

> Se gli utilizzatori interagiscono col vostro codice principalmente su un network—cioè, se il software è usualmente parte di un servizio su hosting—allora considerate la possibilità di usare invece la *GNU Affero GPL*. Vedere La GNU affero GPL: Una Versione della GNU GPL per codice Lato Server in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti* per maggiori dettagli.

Come Applicare Una Licenza Al Vostro Software

Dopo aver scelto una licenza dovreste dichiararla nel pagina principale del progetto. Non c'è bisogno che includiate il vero testo della licenza lì; date solo il nome della licenza e create un link che porti al testo intero della licenza in un'altra pagina.

Questo dice al pubblico sotto quale licenza *volete* che il software sia rilasciato, ma non è sufficiente ai fini legali. Per questo il software stesso deve contenere la licenza. Il classico modo di fare ciò è quello di mettere la intera licenza in un file chiamato `COPYING` (o `LICENSE`), e poi mettere una piccola nota in testa ad ogni file sorgente indicante la data del copyright, il proprietario, e la licenza e indicante dove trovare il testo intero della licenza.

Ci sono molte varianti a questo percorso, sicchè voi vedrete giusto un esempio qui. La GNU GPL dice di mettere una nota come questa in testa ad ogni file sorgente:

```
Copyright (C) <anno>  <nome dell'autore>
Il programma è un software libero; potete redistribuirlo e/o secondo i termini del
dalla Free Software Foundation; sia la versione 2,
sia (a vostra scelta) ogni versione successiva.

Questo programma è distribuito nella speranza che sia utile
ma SENZA ALCUNA GARANZIA; senza anche l'implicita garanzia di
POTER ESSERE VENDUTO o di IDONEITA' A UN PROPOSITO PARTICOLARE.
Vedere la GNU General Public License per ulteriori dettagli.

Dovreste aver ricevuto una copia della GNU General Public License
in questo programma; se non l'avete ricevuta, scrivete alla Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Non si dice specificatamente che la copia della licenza che avete ricevuto col programma è nel file `COPYING` ma che è messa usualmente lì. (Potete cambiare la nota precedente per stabilire ciò completamente.) Questo modello dà anche un indirizzo geografico dal quale richiedere una copia della licenza Un altro sistema comune è quello di dare il link alla pagina in cui si trova la licenza. Usate il vostro giudizio e puntate dove voi credete sia aggiornata la copia più stabile della licenza. Che potrebbe essere semplicemente da qualche parte del sito del progetto. In generale la nota che mettete in ogni file sorgente non deve apparire esattamente come quella di sopra, a condizione che inizi con la nota

del detentore del copyright e della data, stabilisca il nome della licenza, e chiarisca dove poter vedere l'intera licenza.

Dare il Tono

Fin qui abbiamo esaminato il compiti che non ripetuti che voi assolvete durante l'organizzazione di un progetto: scegliere una licenza, mettere e posto il sito iniziale, ecc.. Ma gli aspetti più importanti dell'avvio di un nuovo progetto sono dinamici. Scegliere l'indirizzo della mailing list è facile; assicurarsi che le conversazioni nelle lista rimangano in argomento e siano produttive è completamente un'altra cosa. Se il progetto sta per essere aperto dopo anni di chiusura, di sviluppo in casa, il suo processo di sviluppo cambierà e voi dovrete preparare gli sviluppatori per questo cambiamento.

I primi passi sono i più difficili perchè i diritti comuni e le aspettative per una condotta futura non sono stati ancora stabiliti. La stabilità per un progetto non viene da linee di condotta formali, ma da una saggezza condivisa difficile-da-definire che si sviluppa nel corso del tempo. Ci sono spesso regole scritte, ma esse tendono ad essere essenzialmente un distillato degli intangibile sempre evolventisi accordi che veramente guidano il progetto. Le linee di condotta scritte non definiscono la cultura del progetto finchè lo descrivono, e anche allora lo fanno solo approssimativamente.

Ci sono poche ragioni per le quali le cose trovano una soluzione in questo modo. La crescita e il grande cambiamento non sono così dannosi per l'accumulazione di norme sociali come uno potrebbe pensare. Finchè i cambiamenti non avvengono *troppo* rapidamente, c'è tempo per in nuovi arrivati di imparare come vanno fatte le cose, e dopo che hanno imparato, contribuiranno a rafforzare quei modi stessi. Considerate come le canzoni dei bambini sopravvivono nei secoli. Ci sono bambini oggi che cantano le stesse rime che i bambini cantavano centinaia di anni fa, anche se ora non ci sono bambini vivi che erano vivi allora. I bambini più giovani sentono le canzoni dai più vecchi e quando sono più vecchi, a loro volta le canteranno davanti ai più giovani. I bambini non si impegnano in un consapevole programma di trasmissione, certamente, ma il motivo per cui le canzoni sopravvivono è tuttavia il fatto che esse vengono trasmesse regolarmente e ripetutamente. La scala di tempo dei progetti di software libero non può essere misurata in secoli (non lo sappiamo ancora), ma le dinamiche della trasmissione sono le stesse. La velocità del cambiamento è molto alta, comunque, e può essere compensata da uno impegno nella trasmissione più attivo e intenzionale.

Questo impegno è favorito dal fatto che la gente generalmente evidenzia aspettativa e ricerca di norme sociali. Ed e proprio così che la gente è fatta. In un gruppo unificato dall'impegno comune, la gente che si unisce istintivamente ricerca comportamenti che li segnalerà come parte del gruppo. L'obiettivo di stabilire diritti comuni prima è quello di fare in modo che questi comportamenti “in gruppo” siano gli unici utili al progetto. Una volta stabiliti essi si auto- perpetueranno in gran parte.

Seguono alcuni esempi di cose specifiche che voi potete fare per stabilire buoni diritti comuni. Essi non sono interpretati come un elenco esaustivo, giusto come illustrazioni dell'idea che stabilire un modo collaborativo in principio favorisce un progetto moltissimo. Fisicamente ogni sviluppatore può essere al lavoro da solo in una stanza, ma voi potete fare molto perchè egli si senta come se stesse lavorando insieme agli altri in una sola stanza. Più essi avvertiranno ciò più vorranno spendere tempo per il progetto. Scelgo questi particolari esempi perchè essi si presentarono nel progetto di Subversion(http://subversion.tigris.org/), al quale io partecipai e osservai in esso all'inizio. Ma essi non sono singolari della Subversion; situazioni come queste si presentano in molti progetti open source, e dovrebbero essere viste come opportunità per iniziare a fare le cose col piede giusto.

Evitare discussioni private

Anche dopo che avrete reso pubblico il progetto voi e gli altri fondatori vi troverete spesso desiderosi di sistemare difficili questioni con comunicazioni private in un ambiente più ristretto. Ciò è specialmente

vero nei primi giorni del progetto quando ci sono tante decisioni da prendere, e, usualmente, pochi volontari qualificati per prenderle. Tutti gli ovvi svantaggi delle discussioni in liste pubbliche appariranno palpabili davanti a voi. Il ritardo connesso con le conversazioni per email, il bisogno di avere sufficiente tempo perchè si formi il consenso, il fastidio della negoziazione con volontari sprovveduti che ritengono di capire tutti i problemi, ma in realtà non lo fanno (ogni progetto li ha; a volte essi sono collaboratori protagonisti dell'anno prossimo, a volte restano sprovveduti per sempre), le persone che non riescono a capire che voi volete solo risolvere il problema X anche se questo è ovviamente un aspetto del problema più generale. E così via. La tentazione di prendere decisioni a porte chiuse e di presentarle a loro come *a fatti compiuti*, o almeno come i fermi consigli di un blocco votante unito e influente, sarà grande indubbiamente.

Non fatelo.

Per quanto lente e scomode possano essere le discussioni pubbliche, esse sono quasi sempre preferibili a lungo andare. Rendere private importanti discussioni è come dipingere il collaboratore come repellente per il vostro progetto. Nessun serio volontario resterebbe a lungo nei paraggi di un ambiente in cui tutte le grosse decisioni vengono prese in un consiglio segreto. Inoltre le discussioni pubbliche hanno il benefico effetto secondario che sopravviveranno per quanto effimera fosse la la questione tecnica in discussione:

- La discussione favorirà l'addestramento e l'educazione dei nuovi sviluppatori. Voi non sapete mai quanti occhi sono attenti alla conversazione; anche se la maggioranza della gente non parteciperà, molti seguiranno in silenzio racimolando informazioni sul software.

- nell'arte di spiegare le questioni a persone che non hanno familiarità con le questioni tecniche come l'avete voi. Questa è una capacità che richiede pratica, e voi non potete acquisire questa pratica parlando a gente che già sa quello che sapete voi.

- La discussione e le sue conclusioni saranno disponibili nel pubblico archivio per sempre dopo, dando la possibilità alle future discussioni di evitare di rifare gli stessi passi. Vedere sezione chiamata «Uso Ben Visibile degli Archivi» in Capitolo 6, *Comunicazione* .

Infine, c'è la possibilità che qualcuno nella lista possa dare un reale contributo alla conversazione col far sorgere un'idea che voi non vi sareste mai aspettati. E' difficile dire quanto questo sia possibile. Dipende dalla difficoltà del codice e dal grado specializzazione richiesta. Ma se è permessa una evidenza annedotica, io azzarderei che questo è più probabile di quanto uno si aspetterebbe intuitivamente. Nel progetto di Subversion, noi (i fondatori) eravamo di fronte a una profonda e complessa serie di problemi ai quali stavamo pensando con difficoltà per molti mesi e francamente dubitavamo che ognuno sulla mailing list da poco creata potesse dare un contributo effettivo alla discussione. Così prendemmo pigramente la via e incominciammo a battere su idee tecniche avanti e indietro in emails private finchè uno che osservava il progetto[1] ebbe il sentore di quello che stava avvenendo e chiese che la discussione fosse spostata su una mailing list pubblica. Roteando un poco gli occhi, lo facemmo e fummo stupiti dai commenti perspicaci e dai suggerimenti che presto ne vennero. In molti casi le persone fornirono idee che non ci erano mai venute in mente. Ciò dimostrò che c'erano persone *molto* acute nella mailing list. Esse stavano solo aspettando si essere correttamente allettate. E' vero che le discussioni che ne vennero fuori furono più lunghe di quanto lo sarebbero state se la conversazione fosse stata tenuta privata, ma esse furono talmente molto più costruttive che fu un valore il tempo in più.

Senza scendere in generalizzazioni tipo “il gruppo è sempre più acuto dell'individuo” agitando le mani (noi tutti abbiamo incontrato abbastanza i gruppi per saperlo piuttosto bene), deve darsi per acquisito che ci sono certe attività in cui il gruppo eccelle. La valutazione professionale del lavoro dei colleghi è una di queste; la produzione veloce di un gran numero di idee è un'altra. La qualità delle idee dipende dalla

[1] Non siamo ancora arrivati alla sezione dei crediti, ma giusto alla pratica che io sosterrò più tardi: il nome dell'osservatore era Brian Behlendorf, e fu lui che richiamò l'attenzione sull'importanza di mantenere le discussioni pubbliche, amenochè non ci fosse uno specifico bisogno di privatezza.

quantità del pensare che è entrato in esse, certo, ma voi non saprete che genere di pensatori è sbocciato lì finchè non li stimolerete con un problema impegnativo.

Naturalmente ci sono discussioni che devono avvenire in privato. In questo libro ne vedrete esempi. Ma la guida dovrebbe essere sempre: *Se non c'è ragione perchè esse siano essere private, dovrebbero essere pubbliche.*

Far si che questo accada richiede un'azione. Non c'è solo da assicurarsi che i vostri posts vadano nella lista pubblica. Dovete anche richiamare l'attenzione della altre persone sul fatto che inutili conversazione private vadano anche sulla lista. Se qualcuno tenta di avviare una conversazione privata, e non c'è ragione perchè essa sia privata, allora il vostro compito è quello di aprire immediatamente la relativa discussione pubblica appropriata. E anche non commentate sull'argomento originale finchè o avete condotto la conversazione in un alveo pubblico, o avete accertato che la privacy era necessaria. Se fate ciò con coerenza, le persone afferreranno abbastanza velocemente e incominceranno ad usare i forums per abitudine.

Stroncate sul Nascere la Scortesia

Stroncate sul nascere la scortesia. Dall'inizio dell'esistenza del vostro progetto dovreste mantenere una politica di tolleranza-zero verso comportamenti scortesi o offensivi nei suoi forums. Tolleranza-zero non significa forzatura tecnica di per se. Non dovete rimuovere le persone dalla mailing list quando offendono un altro iscritto o impedire loro la possibilità di inserire messaggi perchè hanno fatto commenti sprezzanti. (In teoria potreste alla fine far ricorso a tale azione, ma solo dopo che le altre vie hanno fallito—cosa che, per definizione non è il caso dell'avvio del progetto.) Tolleranza-zero semplicemente significa non permettere che cattivi comportamenti passino inosservati. Per esempio quando qualcuno posta un commento tecnico insieme ad un attacco *personale* a qualche altro sviluppatore nel progetto, è imperativo che voi rispondiate a questo attacco personale *prima*, come problema separato in se stesso, e solo dopo passiate al contenuto tecnico.

Sfortunatamente è molto facile e troppo tipico, che discussioni costruttive scadano in distruttive guerre offensive. Le persone diranno cose per email che non avrebbero mai detto faccia-a-faccia. L'argomento della discussione però amplifica questo effetto: nei problemi tecnici, le persone spesso pensano che ci sia una sola risposta giusta a molte domande, e che il disaccordo con quella risposta possa essere spiegato come ignoranza o stupidità. Il tratto è breve fra il chiamare la proposta di qualcuno stupida e il chiamare la persona stessa stupida. Infatti, spesso è difficile dire dove si lascia il dibattito e incomincia l'attacco al carattere, che è una ragione per cui reazioni drastiche o punizioni non sono una buona idea. Invece, quando pensate di vedere che questo sta succedendo inserite un post che metta l'accento sull'importanza di mantenere la discussione amichevole, senza accusare nessuno di essere deliberatamente velenoso. Tali posts di ”Politica Simpatica” hanno una sfortunata tendenza a suonare come una ramanzina a una classe di bambini sul buon comportamento:

> *Prima prego limitiamoci ai (potenziali) commenti sulla persona; per esempio, il definire il progetto di J sul livello di sicurezza “ingenuo e ignorante dei principi base della sicurezza del computer.” Ciò può essere vero o no, ma in ogni caso non è un modo con il quale si ha la discussione. J ha fatto la sua proposta in buona fede. Se questa ha delle manchevolezze facciamole notare, e noi le correggeremo o troveremo un altro abbozzo. Io sono sicuro che M non volesse insultare personalmente J, ma il modo di parlare non ha avuto fortuna, e cerchiamo di mantenere le cose costruttive qui da noi.*
>
> *Ora sulla proposta. Penso che M avesse ragione quando diceva...*

Per quanto artificiose suonino queste reazioni, esse hanno un notevole effetto. Se voi costantemente gridate i cattivi comportamenti, ma non chiedete le scuse o una ammissione da parte di chi offende, allora voi lasciate le persone libere di raffreddarsi e di mostrare il lato migliore col comportarsi meglio

la volta successiva—ed essi lo faranno. Uno dei segreti per fare ciò con successo è non trasformare la pseudo discussione nell'argomento principale. Ci dovrebbe essere sempre una breve prefazione a parte, alla parte principale della vostra risposta. Fate notare, incidentalmente, che “non facciamo le cose in questo modo qui da noi”, ma poi andate al reale contenuto in modo tale da dare alle persone qualcosa in argomento su cui rispondere. Se qualcuno obbietta che non merita il vostro rimprovero, rifiutatevi semplicemente di essere condotti sull'argomento. O non rispondete (se pensate che essi si stiano giusto sfogando e non richiedono una risposta), oppure dite che vi dispiace se avete reagito in modo esagerato e che è difficile distinguere le sfumature per email, quindi ritornate all'argomento principale. Mai, mai insistere su una ammissione, pubblica o privata, da parte di qualcuno che si è comportato in modo scorretto. Se essi da sé scelgono di postare delle scuse, ciò è una gran cosa, ma pretendere che lo facciano causerà solo risentimento.

L'obiettivo primario è far si che la buona etichetta sia visto come l'unico comportamento “in-gruppo”. Ciò aiuta il progetto, perchè gli sviluppatori possono essere allontanati (anche da progetti che essi amano e vogliono sostenere) da guerre di offese. Voi potete anche non sapere che essi si sono allontanati; qualcuno potrebbe anche nascondersi nella mailing list, vedere che si fa la pelle dura a partecipare al progetto, e decidere invece di essere coinvolto del tutto. Mantenere il forum confidenziale è una strategia di sopravvivenza, ed è più facile da attuarsi quando il progetto è ancora giovane. Una volta che ciò è parte della cultura, voi non dovrete essere la sola persona a a promuoverlo. Sarà mantenuto da ciascuno.

Praticare una Visibile Revisione del Codice

Uno dei migliori modi di allevare una comunità di sviluppo è il fatto di ottenere che le persone guardino il codice uno dell'altro. E' richiesta una qualche infrastruttura per consentire ciò effettivamente—in particolare, devono essere abilitate email di invio; vedere sezione chiamata «Email di commit» per maggiori dettagli. L'effetto delle email di invio è che ogni volta che uno invia un cambiamento al codice sorgente viene emessa una email che mostra una messaggio di log e le differenze per il cambiamento (vedere *diff*, in sezione chiamata «Vocabolario del controllo di versione»). *La revisione del codice* è la pratica di revisionare le emails di invio così come arrivano, cercando per bugs e possibili miglioramenti.[2]

La revisione del codice serve a diversi scopi simultaneamente. E' il più chiaro esempio di revisione condivisa nel mondo dell'open source, e favorisce direttamente il mantenimento della qualità del software. Ogni bug che si imbarca in un pezzo di software è preso lì per essere stato inviato e non individuato; quindi più sono gli occhi e meno bugs si imbarcheranno. Ma la revisione del codice serve anche ad uno scopo indiretto: esso conferma alle persone che ciò che essi fanno è importante, perchè uno non impiegherebbe il tempo a revisionare un invio amenochè non gli interessasse il suo effetto. Le persone fanno il loro migliore lavoro quando sanno che gli altri impiegheranno del tempo per analizzarlo.

Le revisioni dovrebbero essere pubbliche. Anche in occasioni in cui sono stato a sedere in una stanza fisica con sviluppatori, e uno di noi aveva fatto un invio, facevamo attenzione a non fare la revisione verbalmente nella stanza, ma invece la mandavamo alla mailing list dello sviluppo. Le persone seguono i commenti trovano difetti in essi, ed anche se non li trovano, ciò ricorda loro che la revisione è una attività prevista, regolare, come lavare i piatti, o tosare il prato.

Nel progetto di Subversion noi all'inizio non avevamo la buona pratica della revisione del codice. Non c'era la garanzia che ogni invio sarebbe stato revisionato sebbene uno potesse dare un'occhiata a un cambiamento se era interessato a una particolare area di codice. I bugs in essa potevano e dovevano essere trovati. Uno sviluppatore chiamato Greg Stein che conosceva il valore della revisione del codice

[2] Questo è il modo in cui vene realizzata la revisione del codice nei progetti open source, in ogni caso. In progetti più centralizzati “revisione del codice” può anche significare più persone che si siedono insieme per esaminare accuratamente una stampa del codice, per cercare specifici problemi e modelli.

per passati lavori decise di andare a fare un esempio revisionando ogni linea di *ogni singolo invio* che andava nel deposito degli invii. Ogni invio che ognuno faceva veniva immediatamente seguito da un'email alla lista degli sviluppatori da Greg, che parlava dell'invio, analizzava i possibili problemi, e occasionalmente esprimeva lode su un pezzo intelligente di codice. Subito, egli trovava bugs o pratiche non ottimali di scrivere il codice che diversamente sarebbero passate inosservate. Apparentemente egli non si lamentava di essere l'unica persona che revisionava ogni invio, anche se impiegò una gran parte del suo tempo, ma cantò le lodi delle revisione del codice ogni volta che ne ebbe l'occasione. Molto presto, altre persone, me incluso, incominciarono a revisionare gli invii con regolarità. Quale era la nostra motivazione? Non era che Greg consapevolmente ci avesse rimproverati per farlo. Ma egli aveva provato che la revisione del codice era un modo prezioso di spendere il tempo, e che uno avrebbe contribuito tanto al progetto sia revisionando i cambiamenti degli altri, sia scrivendo nuovo codice. Dopo che egli dimostrò ciò, questo diventò il comportamento scontato, al punto che ogni invio che non riceveva una reazione faceva preoccupare chi lo aveva effettuato, e lo induceva anche a chiedere alla lista se qualcuno tuttavia avesse avuto l'occasione di revisionarlo. Più tardi Greg prese un lavoro che non gli lasciò tanto tempo per Subversion, e dovette finire di fare revisioni con regolarità. Ma, da allora, l'abito si radicò a tal punto per il resto di noi che sembrò che continuasse da tempo immemorabile.

Incominciate a fare revisioni sin dal primo invio. Il tipo di problemi più facili da incontrare nella revisione delle differenze sono le vulnerabilità nella sicurezza, falle nella memoria, i commenti insufficienti o l'insufficiente documentazione, gli errori logici in una iterazione, la discordanza di disciplina chiamante/chiamato, e altri problemi che richiedono un minimo di contesto intorno da vedere. Comunque anche problemi su larga scala come i fallimenti nell'estrarre procedure ripetute in una locazione diventano osservabili dopo che una ha fatto la revisione regolarmente, perchè la memoria di passate differenze avverte sulle differenze presenti.

Non vi spaventate se non trovate niente da commentare, o se non conoscete ogni area di codice. Usualmente ci sarà qualcosa da dire su quasi ognuno degli invii; anche dove non trovate niente da chiedere, potete trovare qualcosa da apprezzare. L'importante è che sia chiaro a ogni persona che fa un invio che ogni cosa che fa è visto e capito. Certo, la revisione del codice non libera i programmatori dalla responsabilità di rivedere il proprio codice prima di inviarlo. Nessuno dovrebbe dipendere dalla revisione del codice per individuare cose che dovrebbe individuare da sé.

Quando Aprite un Progetto che era Chiuso in Passato Fate Attenzione alla Grandezza del Cambiamento

Se state aprendo un progetto esistente, un progetto che ha già sviluppatori attivi abituati a lavorare in un ambiente closed-source, assicuratevi che ognuno capisca che sta venendo un grosso cambiamento, e assicuratevi di capire come ciò sta venendo percepito dal loro punto di vista.

Cercate di capire come la situazione appare loro: in passato, tutte le decisioni riguardanti il codice e la progettazione venivano prese in un gruppo di programmatori che conoscevano il software più o meno bene in modo uguale, in cui tutti ricevevano le stesse pressioni dalla stessa organizzazione, e in cui tutti conoscevano la forza e le debolezze degli altri. Adesso voi state chiedendo loro di esporre il loro codice allo sguardo indagatore di casuali estranei, che si formeranno un giudizio solo sul codice, con nessuna consapevolezza di quante pressioni mercantili possono aver forzato certe decisioni. Questi estranei faranno un sacco di domande, domande che turberanno gli sviluppatori preesistenti fino al punto che essi si renderanno conto che la documentazione sulla quale hanno sgobbato è *tuttavia* inadeguata (questo è inevitabile). E per giunta, i nuovi arrivati sono sconosciuti, entità anonime. Se uno dei vostri sviluppatori non si sente sicuro riguardo alle sue capacità, immaginate come questa cosa si inasprirà quando i nuovi arrivati faranno notare gli errori nel codice che ha scritto, e peggio, lo faranno davanti ai suoi colleghi. Amenoché voi non abbiate un team di perfetti svilippatori, questo è inevitabile—infatti ciò accadrà probabilmente a tutti loro all'inizio. Ciò non avviene perchè essi siano dei cattivi programmatori; avviene giusto che ogni programma al di sopra di una certa grandezza ha bugs, ed ogni revisione alla

pari evidenzierà alcuni di questi bugs (vedere sezione chiamata «Praticare una Visibile Revisione del Codice» prima in questo capitolo). Allo stesso tempo, il nuovo arrivato stesso non sarà soggetto a molte revisioni alla pari all'inizio, perchè non può contribuire al codice finchè non familiarizza col progetto. Ai vostri sviluppatori potrà sembrare che tutto il criticismo sta arrivando e non andando via. Così c'è il pericolo che una mentalità di assedio assuma il controllo fra i vecchi sviluppatori.

Il miglior modo per prevenire ciò è mettere in guardia ciascuno su ciò che sta venendo, spiegarlo, dire loro che lo sconforto iniziale è perfettamente normale, e rassicurarli che sta per andare meglio. Alcuni di questi preavvisi dovrebbe aver luogo in privato, prima che il progetto venga aperto. Ma potete anche trovare utile ricordare alle persone sulle liste pubbliche che questo è un nuovo modo di sviluppare nel progetto, e che ci vorrà un pò di tempo per adattarsi. La miglior cosa che voi possiate fare è guidare con l'esempio. Se vedete che i vostri sviluppatori non rispondono a sufficienza alle domande dei novizi, allora il dire loro di rispondere meglio, non è utile. Può darsi che essi non abbiano ancora una buona idea di ciò che garantisce una reazione e di ciò che non lo fa, o potrebbe darsi che essi non abbiano idea di quanto privilegiare il codice nei confronti del nuovo carico della comunicazione esterna. Il miglior modo di farli partecipare è partecipare voi stessi. Essere sulle mailing lists e accertarsi di rispondere ad alcune domande lì. Quando non avete l'esperienza per rispondere abilmente a una domanda, allora passate in maniera visibile la palla a uno sviluppatore che lo faccia, e assicuratevi che egli dia una risposta, o almeno che abbia una reazione. Ci sarà naturalmente la tentazione per lungo tempo di abbandonarsi a discussioni private, poiché quello era ciò a cui erano abituati. Assicuratevi di essere iscritti alle mailing lists interne in cui ciò può avvenire, dimodochè possiate chiedere che tali discussioni siano spostate sulle liste pubbliche immediatamente.

Ci sono altre attività a lungo termine relative all'apertura di un progetto precedentemente chiuso. Capitolo 4, *L'Infrastruttura Sociale e Politica* esamina tecniche per mettere insieme con successo sviluppatori pagati e non pagati, Capitolo 9, *Licenze, Diritti d'Autore e Brevetti* e tratta della necessità di una osservanza del termini legali quando si apre un codice base privato che può essere stato scritto o “posseduto” da altre parti.

L'Annuncio

Una volta che il progetto è presentabile— non perfetto, giusto presentabile— siete proti per annunciarlo al mondo. Questo è in realtà un procedimento molto semplice: andate a http://freshmeat.net/, cliccate su Invio in cima alla barra di navigazione e riempite un form che annuncia il vostro nuovo progetto. Freshmeat è il posto in cui chiunque guarda per annunci di nuovi progetti. Dovete solo prendere alcune opinioni lì sul vostro progetto da diffondere a voce.

Se conoscete mailing lists o newsgroups dove un annuncio del vostro progetto sarebbe in argomento e di interesse, allora postate lì, ma con l'attenzione di inserire esattamente *un solo* post per forum, e di indirizzare le persone ai forums del vostro progetto come seguito della discussione (impostando l'intestazione `Risposta-a`). I post dovranno esse brevi e andare giusto al punto:

```
A: discuss@lists.example.org
Soggetto: [ANN] Progetto Scanley di indicizzatore full-text
Risposta-a: dev@scanley.org

Questo è il post di una volta che annunciava la creazione del progetto Scanl

Home page: http://www.scanley.org/

Funzionalità:
- Ricerche testo normale, HTML, e XML
- Ricerca di parole o frasi
```

```
- (pianificato) Ricerca diffusa
- (pianificato) Aggionamento incrementale degli indici
- (pianificato) Indicizzazione di siti remoti

Requisiti:
- Python 2.2 o superiore
- Spazio su disco sufficiente a supportare gli indici (approssimativamente 2

Per maggiori informazioni venire su scanley.org.

Grazie,
-J. Random
```

(Vedere sezione chiamata «La Pubblicità» in Capitolo 6, *Comunicazione* per avvisi o per annunci di ulteriori releases o di altri eventi del progetto.)

C'è in corso un dibattito nel mondo del software libero se incominciare col far girare il codice, o se il progetto può beneficiare dall'essere aperto anche durante la fase di progettazione/discussione. Io ero abituato a pensare che partire con il codice che gira fosse il fattore più importante, che esso fosse ciò che separa i progetti di successo dai giochi, e che gli sviluppatori seri sarebbero stati attratti da software che facesse già qualcosa di concreto.

Questo si dimostrò non essere il caso. Nel progetto di Subversion, noi partimmo con un documento di progetto, un cuore di sviluppatori interessati e ben collegati, un sacco di fanfara, e un codice che *non* girava affatto. Con mia totale sorpresa, il progetto acquisì principianti attivi giusto dall'inizio, e col tempo avemmo qualcosa che girava, c'erano parecchi sviluppatori profondamente coinvolti. Subversion non è il solo esempio; il progetto Mozilla fu lanciato senza codice che girava, ed è ora un browser popolare e di successo.

Di fronte a una tale evidenza, dovetti fare marcia indietro sull'affermazione che far girare il codice fosse assolutamente necessario per lanciare un progetto. Far girare il codice è ancora la migliore premessa per il successo, e una buona regola empirica approssimativa sarebbe aspettare di farlo girare prima di annunciare il vostro progetto. Comunque ci sono circostanze in cui fare l'annuncio presto ha un senso. Penso che almeno un documento di progetto ben sviluppato, o diversamente una sorta di struttura di codice sia necessaria—certamente può essere rivista basandosi su una pubblica risposta, ma ci deve essere qualcosa di concreto, qualcosa di più tangibile oltre a delle sole buone intenzioni, affinchè le persone possano affondarvi i denti.

In qualsiasi momento facciate l'annuncio, non aspettatevi che un'orda di volontari si aggiungano al progetto. Usualmente, il risultato dell'annuncio è che voi ricevete poche domande casuali, e inoltre poche persone si aggiungono alle vostre mailing lists, e a parte questo, ogni cosa continua ad andare abbastanza come prima. Ma col tempo, noterete un graduale aumento nella partecipazione sia di collaboratori al nuovo codice sia di utilizzatori. L'annuncio è solo un piantare il seme. Ci può essere bisogno di molto tempo perchè la notizia si diffonda. Se il progetto coerentemente ricompensa coloro che sono coinvolti, le notizie *si diffonderanno*, comunque, perchè le persone vogliono condividere quando hanno trovato qualcosa di buono. Se tutto va bene, le dinamiche di reti di comunicazione esponenziali trasformeranno lentamente il progetto in una complessa comunità, in cui non dovete conoscere necessariamente il nome di ciascuno, e non potete seguire ogni singola conversazione. I prossimi capitoli parlano del lavoro in questo ambiente.

Capitolo 3. L'Infrastruttura Tecnica

I progetti di software libero poggiano su tecnologie che supportano la cattura e l'integrazione dell'informazione. Più esperti sarete nell'usare queste tecnologie, e nel persuadere gli altri a usarle, più il vostro progetto avrà successo. Ciò diventa vero quando il progetto cresce. La buona gestione dell'informazione è ciò che previene dal collasso del progetto sotto il peso della legge di Brooks ,[1] che stabilisce che aggiungere forza lavoro a un progetto avanzato lo rende più avanzato. Fred Brooks osservò che la complessità di un progetto cresce con *il quadrato* del numero dei partecipanti. Quando sono coinvolte solo poche persone, ognuno può parlare facilmente all'altro, ma quando sono coinvolte centinaia di persone, non è ulteriormente possibile per una persona essere messa al corrente di ciò che ciascun altro sta facendo. Se una buona gestione di software libero sta facendo in modo che ognuno si senta al lavoro con gli altri nella medesima stanza, la domanda ovvia è: cosa avviene quando ognuno in una stanza affollata cerca di parlare simultaneamente?

Questo problema non è nuovo. In stanze non-metaforicamente affollate la soluzione è una *procedura parlamentare* : linee guida formali su come avere discussioni in tempo reale in grandi gruppi, come assicurarsi che importanti dissensi non si perdano nelle inondazioni dei commenti #anch'io# , come dar forma a sottocommissioni, come riconoscere quando le decisioni vengono prese, ecc.. Una parte importante della procedura parlamentare è specificare come il gruppo interagisce con il suo sistema di gestione delle informazioni. Alcuni rilievi vengono fatti #per registrazione# , altri no. La registrazione stessa è soggetta a una manipolazione diretta ed è scambiata per una trascrizione letterale di ciò che è avvenuto, non per ciò su cui il gruppo *concorda* che sia avvenuto. La registrazione non è monolitica, ma assume forme differenti a seconda del fine. Essa comprende i minuti degli incontri individuali, l'insieme di tutti i minuti di tutti gli incontri, i sommari, le agende e le loro annotazioni, i rapporti della commissione, i rapporti dei corrispondenti non presenti, gli elenchi delle azioni, ecc..

Poichè Internet non è una stanza reale, noi non dobbiamo preoccuparci di riprodurre quelle parti della procedura parlamentare che mantengono le persone ferme mentre altre stanno parlando. Invece quando fanno propria la tecnica dell'organizzazione delle informazioni, i progetti open source che vanno bene sono una forma molto amplificata di procedura parlamentare. Poiché quasi tutte le comunicazioni nei progetti open source avvengono scrivendo, i sistemi elaborati si sono evoluti per l'instradamento e l'etichettatura dei dati; per evitare ripetizioni, come per evitare divergenze spurie; per immagazzinare e recuperare dati; per correggere informazioni cattive ed obsolete; e per associare pezzi di informazione con ogni altra nella misura un cui vengono rilevate nuove connessioni. I partecipanti attivi nei progetti open source fanno proprie molte di queste tecniche e metteranno a punto complesse operazioni manuali per assicurare all'informazione di essere instradata correttamente. Ma l'intero sforzo spesso dipende da un sofisticato supporto di software. Quanto più possibile, i mezzi di comunicazione non realizzano l'instradamento, la classificazione e la registrazione, e dovrebbero rendere disponibile l'informazione agli uomini nella maniera più pratica. In pratica, gli uomini avranno ancora bisogno di intervenire in molti punti del processo, ed è importante che il software renda tali interventi anche pratici. Ma, in generale, se gli uomini hanno cura di classificare e di instradare l'informazione al suo primo ingresso nel sistema il software sarà configurato in modo da fare uso di gruppi di dati il più possibile.

Il consiglio in questo capitolo è intensamente pratico, basato su sull'esperienza con software specifici e su esempi d'uso. Ma il punto non è solo insegnare un particolare insieme di tecniche. E' anche quello di dimostrare, con l'aiuto di molti piccoli esempi, l'attitudine complessiva che che meglio incoraggerà la buona gestione dell'informazione nei vostri progetti. Questa attitudine coinvolgerà una combinazione di capacità tecniche e capacità umane. Le capacità tecniche sono perchè la gestione delle informazioni spesso richiede la configurazione, più una certa quantità di manutenzione in corso e di nuove regolate nelle misura in cui nuovi bisogni sorgono (come esempio vedere la discussione su come trattare un progetto cresciuto in sezione chiamata «Pre-Filtraggio del Bug Tracker» più in là in questo capitolo).

[1] Dal suo libro *Il mitico mese dell'uomo* , 1975. vedere http://en.wikipedia.org/wiki/The_Mythical_Man-Month and http://en.wikipedia.org/wiki/Brooks_Law.

Le abilità delle persone sono necessarie perché la comunità umana anche richiede manutenzione: non è sempre immediatamente ovvio come usare questi strumenti con profitto, e in alcuni casi i progetti hanno delle convenzioni in conflitto (per esempio vedere la discussione sul predisporre le intestazione `Rispondere-a` nei posts in uscita in sezione chiamata «Mailing Lists»). Chiunque sarà coinvolto nel progetto avrà bisogno di essere incoraggiato, al momento giusto e nella giusta maniera, per fare la sua parte nel mantenere l'informazione del progetto ben organizzata. Più sarà coinvolto il collaboratore, più saranno complesse e specializzate le tecniche che ci si aspetterà che possa imparare.

La gestione delle informazioni non ha una soluzione dal taglio secco. Ci sono molte variabili. Potete aver configurato definitivamente ogni cosa giusto nel modo che voi volevate, ed avere la maggior parte della comunità che partecipa, ma la crescita del progetto renderà alcune di quelle pratiche non accessibili. O la crescita del progetto si può stabilizzare, e le comunità degli sviluppatori e degli utilizzatori si possono comporre in una comoda relazione con l'infrastruttura, ma allora qualcuno si muoverà e inventerà un servizio di gestione delle informazioni completamente nuovo, e ben presto i nuovo arrivati si chiederanno perché voi non lo usiate —per esempio ciò sta avvenendo per un sacco di progetti di software libero che anticipano l'invenzione del wiki (vedere http://en.wikipedia.org/wiki/Wiki). Molte questioni sono materia di giudizio, riguardando un compromesso tra la convenienza di quelli che producono informazione e la convenienza di quelli che la utilizzano, o fra il tempo richiesto per configurare il software per la gestione l'informazione e i vantaggi che porta al progetto.

Guardarsi dalla tentazione di una super automazione, cioè, automatizzare cose che richiedono attenzione umana. L'infrastruttura tecnica è importante, ma ciò che fa si che il progetto di software libero funzioni è la cura e l'intelligente manifestazione di quella cura, immessavi dagli uomini. L'infrastruttura tecnica consiste nel dare agli uomini modi convenienti di realizzare ciò.

Di cosa ha bisogno un progetto

La maggior parte dei progetti open source offrono almeno un minimo insieme standard di strumenti per la gestione dell'informazione:

Sito Web
: Principalmente un canale di informazione centralizzato e a senso unico, dal progetto verso il pubblico. Il sito web può anche servire come interfaccia di amministrazione di altri strumenti del progetto.

Mailing list
: Solitamente il forum di comunicazione più attivo, e il mezzo di memorizzazione ("mezzo di registrazione").

Controllo di versione
: Permette agli sviluppatori di gestire adeguatamente le modifiche nel codice, compresi i passi indietro e il cambio di portabilità. Permette a tutti di vedere cosa sta succedendo al codice.

Tracciamento dei bug
: Permette agli sviluppatori di tenere traccia di quello su cui stanno lavorando, coordinarsi l'uno con l'altro e pianificare i rilasci. Permette a tutti di fare interrogazioni sullo stato dei bug e di memorizzare informazioni (per esempio istruzioni per la soluzione di certi tipi di problemi) riguardanti particolari bug. Può essere usato non solo per i bug ma anche per attività, rilasci, nuove funzionalità, eccetera.

Chat in tempo reale
: Un posto per scambi di botta—e—risposta e discussioni veloci e leggere. Non sempre completamente archiviata.

Ognuno di questi strumenti affronta un bisogno specifico, ma le loro funzioni sono anche correlate, e gli strumenti devono essere fatti per lavorare insieme. Più avanti esamineremo come possono farlo, e ancora

più importante, come fare in modo che la gente li usi. Il sito web non sarà discusso fino alla fine dato che funziona più come collante per le altre componenti che come uno strumento in sè.

Dovreste essere in grado di evitare molti mal di testa scegliendo e configurando questi strumenti usando un sito di cossiddetto *canned hosting*: un server che offre aree web preconfezionate, con stile coerente e con tutti gli strumenti necessari a portare avanti un progetto di software libero. Vedi sezione chiamata «Canned Hosting» più avanti in questo capitolo per una discussione su vantaggi e svantaggi del canned hosting.

Mailing Lists

Le mailing list sono il sale delle comunicazioni all'interno di un progetto. Se un utente fa parte di un forum che non siano pagine web, è molto probabile questo che sia la mailing list. Ma prima che gli utenti provino la mailing list, devono provare l'interfaccia della mailing list—cioè il meccanismo con cui si aggiungono ("subscribe to") alla mailing list. Questo ci porta alla Regola numero 1 della mailing list:

> *Non provate a gestire le mailing list a mano— usate un software di gestione.*

Si può essere tentati di farne a meno. Configuare un software di gestione di mailing list può sembrare un massacro, di primo acchito. Gestire liste, piccole e con poco traffico, a mano sembrerà seduttivamente facile: si crea un indirizzo di sottoscrizione che fa il forward delle mail su di voi, e quando qualcuno ci scrive, aggiungete (o togliete) gli indirizzi email in qualche file di testo che contiene tutti gli indirizzi della lista. Cosa c'è di più facile?

Il trucco sta nel fatto che una buona gestione di mailing list—che è cosa le persone sono arrivate ad aspettarsi—non è per niente facile. Non è solo questione di aggiungere o rimuovere utenti quando essi lo richiedono. E' anche moderare per prevenire lo spam, offrire la mailing list come compendio piuttosto che messaggio per messaggio, fornire una lista standard e informazioni sul progetto come le risposte automatiche, e varie altre cose. Un essere umano che controlla gli indirizzi di sottoscrizione può fornire appena un minimo di funzionalità, e neppure in modo affidabile e pronto come lo farebbe un software.

I moderni software di gestione di mailing list solitamente offrono almeno le seguenti funzionalità:

Iscrizione sia via email che da sito web
: Quando un utente si iscrive alla mailing list, dovrebbe *velocemente* ricevere un messaggio automatico di benvenuto in risposta, che spieghi cosa ha sottoscritto, come interagire ulteriormente con il software di mailing list e (soprattutto) come disiscriversi. Questa risposta automatica può essere personalizzata in modo da contentere informazioni specifiche del progetto, di sicuro, come il sito web del progetto, dove si trovano le FAQ, eccetera.

Sottoscrizione sia in modalità aggregata che in modalità messaggio per messaggio
: Nella modalità aggregata, il sottoscrittore riceve una mail al giorno, contenente tutta l'attività di quel giorno. Per le persone che stanno seguendo la mailing list senza parteciparvi attivamente, la modalità aggregata è spesso preferibile, perchè permette loro di vedere tutti gli argomenti in una volta sola e di evitare la distrazione causata da email in arrivo ad ogni momento.

Funzionalità di moderazione
: "Moderare" è l'attività di controllare i messaggi per assicurarsi che siano a) non spam, e b) sull' argomento, prima che vengano mandati a tutta la mailing list. L'attività di moderazione necessariamente impiega gli esseri umani, ma il software può fare molto per semplificarla. Più avanti verrà detto altro riguardo alla moderazione.

Interfaccia di amministrazione
: Tra le altre cose, permette ad un amministratore di entrare e rimuovere facilmente indirizzi obsoleti. Questo può diventare urgente quando un indirizzo destinatario inizia a mandare indietro risposte automatiche "Non sono più a questo indirizzo" alla mailing list in risposta ad ogni messaggio.

(Alcuni software per mailing list possono perfino capirlo da soli e rimuovere la persona in maniera automatica)

Manipolazione degli header
: Molta gente definisce nei propri programmi di posta sofisticate regole di filtro e risposta. I software di gestione di mailing list possono aggiungere e manipolare certi header standard per queste persone così da guadagnare vantaggio da (maggiori dettagli sotto).

Archiviazione
: Tutti i messaggi della mailing list sono memorizzati e resi disponibili sul web; come alternativa, alcuni software di gestione offrono speciali interfacce per l'integrazione di strumenti di archiviazione esterni come MHonArc (http://www.mhonarc.org/). Come sezione chiamata «Uso Ben Visibile degli Archivi» in Capitolo 6, *Comunicazione* sostiene, l'archiviazione è cruciale.

Lo scopo di tutto questo è semplicente mettere l'accento sul il fatto che la gestione delle mailing list è un problema complesso che ha dato molti problemi, la maggior parte dei quali è stata risolta. Sicuramente non avrete bisogno di diventarne degli esperti. Ma dovete essere consci del fatto che c'è sempre modo di imparare cose nuove, e che la gestione della mailing list occuperà la vostra attenzione di tanto in tanto durante la vita di un progetto open source. Più avanti esamineremo alcuni dei problemi di configurazione delle mailing list più comuni.

Prevenire lo spam

Tra quando questa frase è stata scritta e quando è stata pubblicata, il problema dello spam, grande quanto la Rete, sarà probabilmente raddoppiato nella sua gravità— o almeno sembrerà così. C'era una volta, non così tanto tempo fa, un tempo in cui uno poteva gestire una mailing list senza adottare assolutamente nessuna prevenzione per lo spam. Gli occasionali messaggi non pertinenti sarebbero comunque comparsi, ma abbastanza infrequentemente ed essere solo una piccola noia. Quell'era è andata per sempre. Oggi, una mailing list che non adotta misure di prevenzione dello spam sarà velocemente sommersa di email spazzatura, fino al punto di diventare inservibile. La prevenzione dello spam è imperativa.

Dividiamo la prevenzione dello spam in due categorie: prevenire il comparire di messaggi di spam sulla mailing list e prevenire che la vostra mailing list diventi una fonte di nuovi indirizzi email per gli spammer. La prima è più importante, quindi la esaminiamo per prima.

Filtrare i messaggi

Ci sono tre tecniche di base per prevenire i messaggi di spam, e la maggior parte dei software di gestione di mailing list li offre tutti. E' meglio usarli tutti insieme:

1. **Permettere la pubblicazione automatica di messagggi solo da parte degli iscritti della mailing list.**

   Questo è efficace finchè funziona, e comporta un piccolissimo sforzo amministrativo, dato che solitamente si tratta di cambiare una configurazione nel software di gestione di mailing list.
   Ma bisogna notare che i messaggi che non sono approvati automaticamente non devono essere semplicemente scartati. Invece, dovrebbero essere passati alla moderazione, per due ragioni. Primo, potete voler permettere ai non inscritti di pubblicare messaggi. Una persona con una domanda o un consiglio non dovrebbe avere bisogno di iscriversi alla mailing list solo per pubblicare un solo messaggio. Secondo, anche gli iscritti possono a volte pubblicare da un indirizzo diverso da quello con cui si sono iscritti. Gli indirizzi email non sono un metodo affidabile per identificare le persone, e non dovrebbero essere trattati come tali.

2. **Filtrare i messaggi attraverso software che filtrano spam.**

Se il software di gestione lo rende possibile (molti lo fanno), potete avere i messaggi filtrati da un apposito software. Il filtro automatico per lo spam non è perfetto, e non lo sarà mai, dato che ci sarà una corsa alle armi senza fine tra spammer e creatori di filtri. Comunque, può considerevolmente ridurre la quantità di spam che passa attraverso la coda di moderazione, e dato che più lunga è la coda e più tempo gli esseri umani ci mettono a controllarla, ogni quantità di filtro automatico è vantaggiosa.

Non c'è spazio qui per le istruzioni dettagliate su come configurare i filtri per lo spam. Dovrete consultare la documentazione del vostro software di gestione mailing list per questo (vedi sezione chiamata «Software» più avanti in questo capitolo). Tali software a volte arrivano con alcune funzionalità precostruite di prevenzione spam, ma potreste voler aggiungere alcuni filtri di terze parti. Ho avuto buone esperienze con questi due: SpamAssassin (http://spamassassin.apache.org/) e SpamProbe (http://spamprobe.sourceforge.net/). Questo non è un commento sui molti altri filtri open source per lo spam al di fuori di questi, alcuni dei quali sono a prima vista abbastanza buoni. Semplicemente mi è capitato di usare questi due e ne sono rimasto soddisfatto.

3. **Moderazione.**

   Per le mail che non sono automaticamente autorizzate in virtù del fatto che provengono da un iscritto alla mailing list, e che passano attraverso il software di filtraggio spam, se presente, l'ultimo passo è la *moderazione*: la mail viene mandata ad uno speciale indirizzo, dove un essere umano la esamina e la conferma o la respinge.

   Confermare un messaggio assume una tra due forme: potete accettare il messaggio solo per questa volta, o potete dire al software di ammettere questo e i prossimi messaggi dallo stesso mittente. Dovreste quasi sempre fare così, per ridurre la futura mole di moderazione. I dettagli su come fare ad accettare cambiano da sistema a sistema, ma è solitamente questione di rispondere ad uno speciale indirizzo con il comando "accept" (che significa accettazione solo per questo messaggio) o "allow" (permetti questo e i messaggi futuri).

   Si respinge solitamente ignorando la mail di moderazione. Se il software di gestione non riceve mai conferme che qualcosa è un messaggio valido, allora non lo passerà alla mailing list, quindi semplicemente ignorando la mail di moderazione ottiene l'effetto desiderato. A volte avete anche l'opzione di rispondere con un comando "rejet" o "deny", per disapprovare automaticamente le future mail dallo stesso mittente senza neppure farle passare attraverso la moderazione. Raramente c'è motivo di fare così, dato che la moderazione si occupa soprattutto di spam, e gli spammer tendono a non usare lo stesso indirizzo due volte.

Cercate di usare la moderazione *solo* per scartare spam e messaggi palesemente non attinenti, come quando qualcuno uno manda un messaggio alla mailing list sbagliata. Il sistema di moderazione solitamente vi fornirà un modo per rispondere direttamente al mittente, ma non usate questo metodo per rispondere a domande che in realtà appartengono alla stessa mailing list, anche se sapete al volo la risposta. Fare così priverà la comunità del progetto dell'immagine accurata dei tipi di domande che la gente pone, e negherà un'opportunità di rispondere alle domande e/o di vedere le risposte di altri. La moderazione di mailing list riguarda soltanto l'evitare i messaggi spazzatura e email non pertinenti, nient'altro.

Nascondere gli indirizzi presenti negli archivi

Per prevenire che la vostra mailing list diventi una fonte di indirizzi per gli spammer, una tecnica comune è nascondere gli indirizzi delle email delle persone presenti negli archivi, ad esempio sostituendo

```
jrandom@somedomain.com
```

con

```
jrandom_AT_somedomain.com
```

oppure

```
jrandomNOSPAM@somedomain.com
```

o qualche altra simile evidente (per gli esseri umani) codifica. Dato che i raccoglitori di indirizzi di spam spesso funzionano elaborando le pagine web—compresi gli archivi onlime della vostra mailing list—e cercando sequenze di caratteri contententi "@", codificare gli indirizzi è un modo di rendere gli indirizzi email della gente invisibili o inutili per gli spammer. Questo non evita che dello spam sia inviato alla mailing list stessa, certo, ma evita che la quantità di spam inviata direttamente agli indirizzi personali degli utenti aumenti.

Nascondere gli indirizzi può essere controverso. Alcune persone lo apprezzano molto e saranno sorpresi se i vostri archivi non lo fanno automaticamente. Altri pensano che sia un disturbo (perchè gli esseri umani devono anche ritradurre gli indirizzi prima di usarli). A volte la gente afferma che è inutile, perchè un raccoglitore di indirizzi potrebbe in teoria supplire ogni modo coerente di codifica. Comunque, va notato che c'è una prova empirica del fatto che nascondere gli indirizzi *è* efficace, vedi http://www.cdt.org/speech/spam/030319spamreport.shtml.

Idealmente il software di gestione dovrebbe lasciare la scelta ad ogni iscritto individualmente, o attraverso uno speciale header si/no oppure una configurazione nelle preferenze del proprio account. Comunque, non sono a conoscenza di alcun software che offre una scelta per iscritto o per messaggio, quindi per ora il gestore della mailing list deve prendere una decisione per tutti (assumendo che l'archiviazione abbia tale funzionalità, il che non accade sempre). Tendo moderatamente verso l'attivazione dell'occultamento degli indirizzi. Alcune persone sono molto attente ad evitare di pubblicare i loro indirizzi email su pagine web o in qualunque altro posto in cui un raccoglitore di indirizzi potrebbe vederlo, e sarebbero dispiaciute di vedere tutta la loro cura buttata via da un archivio di mailing list; allo stesso tempo, il disturbo che l'occultamento di indirizzi crea agli utenti è molto leggero, dato che è banale ritrasformare un indirizzo oscurato in uno valido se avete bisogno di contattare quella persona. Ma tenete a mente che, alla fine, è sempre una corsa alle armi: nel tempo in cui avete letto questo, i raccoglitori di indirizzi potrebbero essersi evoluti al punto in cui possono riconoscere i metodi più comuni di occultamento, e dovremo pensare a qualcos'altro.

Identificazione e gestione degli header

Gli iscritti spesso vogliono mettere le mail della mailing list in una cartella specifica per il progetto, separata dalle altre mail. Il loro software per la lettura di email può fare ciò automaticamente esaminando gli *header* delle email. Gli header sono i campi in cima della mail che indicano mittente, destinatario, oggetto, data, e varie altre cose riguardo al messaggio. Alcuni header sono noti e anche obbligatori:

```
From: ...
To: ...
Subject: ...
Date: ...
```

Altri sono opzionali, anche se abbastanza standard. Per esempio, alle email non è strettamente richiesto di avere il seguente header

```
Reply-to: sender@email.address.here
```

ma molti ce l'hanno, perchè dà ai destinatari un modo per raggiungere l'autore assolutamente sicuro (è specialmente utile quando l'autore ha dovuto inviare da un indirizzo diverso da quello a cui le risposte andrebbero inviate).

Alcuni software per la lettura delle email offrono semplici interfacce per riempire le email basandosi sul modello dell'header dell'oggetto della mail. Questo porta la gente a richiedere che la mailing list aggiunga automaticamente un prefisso a tutti gli oggetti delle email, così che loro possano configurare i loro software per cercare tale prefisso e mettere le email nelle giusta cartella in modo automatico. L'idea è che l'autore scriva:

```
Oggetto: fare la release 2.5.
```

ma l'email inviata nella mailing list sarà del tipo:

```
Oggetto: [discuss@lists.example.org] fare la release 2.5.
```

Anche se la maggior parte dei software di gestione offre l'opzione di farlo, raccomando fortemente di non abilitare questa opzione. Il problema che risolve può essere facilmente affrontato in altri modi meno intrusivi, e il costo dello spazio sprecato nell'header dell'oggetto è davvero troppo alto. Gli utenti esperti di mailing list scorrono gli oggetti delle email in arrivo per decidere cosa leggere e/o a cosa rispondere. Aggiungere il nome della mailing list all'inizio dell'oggetto delle email può spingere la parte destra dell'oggetto fuori dallo schermo, invisibile. Ciò nasconde l'informazione di cui la gente ha bisogno per decidere quali email aprire, riducendo quindi per tutti la funzionalità complessiva della mailing list.

Invece di usare l'header dell'oggetto, insegnate ai vostri utenti ad usare a loro vantaggio gli altri header standard, a cominciare dall' header del destinatario (To), che dovrebbe dire il nome della mailing list:

```
To: <discuss@lists.example.org>
```

Ogni software di lettura mail che può applicare filtri sull'oggetto dovrebbe essere in grado di filtrare altrettanto facilmente l'header del destinatario.

Ci sono alcuni altri header opzionali ma standard che sono previsti nelle mailing list. Applicare filtri su questi è persino più affidabile che usare gli header "To" o "Cc" (Carbon copy, Copia carbone); dato che questi header sono aggiunti dal software di gestione ad ogni nuovo messaggio pubblicato, alcuni utenti potrebbero contare sulla loro presenza:

```
list-help: <mailto:discuss-help@lists.example.org>
list-unsubscribe: <mailto:discuss-unsubscribe@lists.example.org>
list-post: <mailto:discuss@lists.example.org>
Delivered-To: mailing list discuss@lists.example.org
Mailing-List: contact discuss-help@lists.example.org; run by ezmlm
```

Per la maggior parte, sono auto-esplicativi. Vedi http://www.nisto.com/listspec/list-manager-intro.html per ulteriori spiegazioni, o se avete bisogno della specifica formale e veramente dettagliata, http://www.faqs.org/rfcs/rfc2369.html.

Va notato che questi header implicano che se avete una mailing list che si chiama "list", allora avete anche gli indirizzi amministrativi "list-help" e "list-unsubscribe" a disposizione. Oltre a questi, è

normale avere "list-subscribe", per iscriversi, e "list-owner", per contattare gli amministratori della mailing list. A seconda del software di gestione che usate, questi e/o vari altri indirizzi amministrativi possono essere creati; la documentazione conterrà dettagli su questo. Solitamente una spiegazione completa di tutti questi indirizzi speciali è inviata ad ogni nuovo utente come parte di una email automatica di benvenuto quando si iscrive. Voi stessi probabilmente avrete una copia di questa mail di benvenuto. Se non l'avete, allora chiedetene una copia a qualcun altro, così saprete cosa gli utenti vedono quando si iscrivono alla mailing list. Tenete questa copia a portata di mano così da poter rispondere alle domande riguardo alle funzionalità della mailing list, o ancora meglio mettetela su una pagina web da qualche parte. In questo modo quando la gente perde la loro copia delle istruzioni e scrive per chiedere "Come mi disiscrivo dalla mailing list?", possiate semplicemente passar loro l'URL.

Alcuni software di gestione offrono come opzione l'appendere le istruzioni per la disiscrizione al fondo di ogni messaggio. Se tale opzione è disponibile, usatela. Provoca solo qualche riga in più ad ogni messaggio, in un posto che non dà problemi, e può risparmiarvi molto tempo, riducendo il numero di persone che scrivono—o peggio, scrivono alla mailing list!—chiedendo come disiscriversi.

Il grande dibattito sul 'Rispondi A'

Prima, in sezione chiamata «Evitare discussioni private», ho sottolineato l'importanza di fare in modo che le discussioni avvengano sui forum pubblici, e ho parlato di come misure sono a volte necessarie per prevenire che le conversazioni finiscano in flussi di email private; inoltre, questo capitolo tratta di come configurare il software di comunicazione del progetto per fargli fare al vostro posto la maggior parte di lavoro possibile. Quindi, se il software di gestione offre un modo di far rimanere in modo automatico le discussioni nella mailing list, potreste pensare che abilitare questa funzionalità sia la scelta più ovvia.

Be', non proprio. Questa funzionalità esiste, ma ha alcuni importanti svantaggi. La domanda riguardante il suo uso o meno è uno dei dibattiti più accesi nella gestione di mailing list—magari non la notizia che compare sui giornali della sera, ma può comparire di tanto in tanto nei progetti di free software. Più avanti, descriverò questa funzionalità, darò le argomentazioni a sostegno di entrambe le fazioni, e darò il miglior consiglio che posso.

La funzionalità in sè è molto semplice: il software di gestione può, se volete, configurare automaticamente l'header Reply-to (Rispondi A) di ogni messaggio in modo da ridirigere le risposte sulla mailing list. Cioè, nonostante cosa il mittente originale ha messo nell'header Reply-to (o anche se non è stato incluso del tutto), al momento in cui gli iscritti vedono il messaggio, l'header conterrà l'indirizzo della mailing list:

```
Reply-to: discuss@lists.example.org
```

A prima vista, sembra essere una buona cosa. Dato che virtualmente ogni software di lettura mail presta attenzione all'header Reply-to, quando chiunque risponde ad un messaggio, la risposta sarà automaticamente mandata all'intera mailing list, non solo al mittente del messaggio a cui si è risposto. Sicuramente, chi risponde può ancora cambiare a mano il destinatario, ma la cosa importante è che *per default* le risposte siano mandate alla mailing list. E' un esempio perfetto dell'uso della tecnologia per incoraggiare la collaborazione.

Sfortunatamente, ci sono alcuni svantaggi. Il primo è noto come il problema del *Non Riesco a Trovare la Strada di Casa*: a volte il mittente originale potrà mettere il proprio "vero" indirizzo email nel campo Reply-to, perchè per una qualche ragione manda la mail da un indirizzo diverso da quello dove lo riceve. La gente che legge e manda sempre dallo stesso indirizzo non ha questo problema, e potrebbe essere sorpresa della sua esistenza. Ma per chi ha configurazioni email particolari, o chi non può controllare come il suo indirizzo mittente sarà composto (magari perchè scrivono dal lavoro e non hanno influenze sul dipartimento IT), usare il Reply-to potrebbe essere l'unico modo che hanno per assicure che la

risposta li raggiunga. Quando una persona di questo genere scrive ad una mailing list a cui non è iscritto, la sua configurazione del Reply-to diventa un'informazione essenziale. Se il software di gestione lo sovrascrive, potrebbe non vedere mai la risposta al suo messaggio.

Il secondo svantaggio ha a che fare con le aspettative, e secondo la mia opinione è l'argomento più potente contro l'occultamento del Reply-to. La maggior parte degli utenti esperti di email sono abituati a due modi basilari di risposta:*rispondi a tutti (reply-to-all)* e *rispondi all'autore (reply-to-author)*. Tutti i moderni software di lettura mail hanno comandi separati per queste due azioni. Gli utenti sanno che per rispondere a tutti (quindi, includendo la mailing list), devono scegliere reply-to-all, e che per rispondere privatamente all'autore, devono scegliere reply-to-author. Anche se volete incoraggiare le persone a rispondere alla mailing list ogni volta sia possibile, ci sono certe circostanze in cui una risposta privata è prerogativa di chi risponde— per esempio, potrebbero volere dire qualcosa di confidenziale all'autore del messaggio originale, qualcosa che potrebbe essere non appropriato per la mailing list pubblica.

Considerate ora cosa succede quando la mailing list sovrascrive il Reply-to del mittente originale. Chi risponde schiaccia il tasto reply-to-author, aspettandosi di rispondere con un messaggio privato all'autore originale. Dato che questo è il comportamento atteso, potrebbe non preoccuparsi di guardare con attenzione all'indirizzo di destinazione nel nuovo messaggio. Scrive il suo messaggio privato e confidenziale, uno in cui magari dice cose imbarazzanti riguardo a qualcuno della mailing list, e schiaccia il tasto di invio. Inaspettatamente, pochi minuti dopo il suo messaggio appare *sulla mailing list!*. D'accordo, in teoria dovrebbe avere controllato al campo destinatario, e non dovrebbe aver presunto nulla riguardo all'header Reply-to. Ma gli autori praticamente sempre mettono nel Reply-to il loro indirizzo personale (o meglio il software di posta lo fa per loro), e molti utilizzatori di email di lunga data se lo aspettano. Infatti, quando una persona mette deliberatamente nel Reply-to qualche altro indirizzo, come la mailing list, solitamente lo scrive nel corpo del messaggio, così che la gente non sarà sorpresa per cosa succede quando rispondono.

A causa delle conseguenze anche serie di questo comportamento inaspettato, la mia personale preferenza è la configurazione del software di gestione della mailing list in modo che non tocchi mai l'header Reply-to. Questo è un caso in cui usare la tecnologia per incoraggiare la collaborazione ha, mi sembra, effetti collaterali potenzialmente pericolosi. In ogni caso, ci sono anche alcuni buoni argomenti nell'altra fazione del dibattito. Qualunque modo sceglierete, ogni tanto troverete persone che scrivono sulla mailing list chiedendo perchè non abbiate scelto l'altro modo. Dato che questo non è qualcosa che desiderate avere come argomento principale di discussione, potrebbe essere conveniente avere una risposta pronta preconfezionata, del tipo che sia incline a fermare la discussione piuttosto che incoraggiarla. Cercate di *non* insistere sul fatto che la vostra decisione, qualunque sia, è ovviamente l'unica giusta e sensata (anche se pensate che sia così). Cercate invece di precisare che è un vecchio dibattito, che ci sono buone argomentazioni a favore di entrambe le fazioni, che nessuna scelta soddisferà tutti gli utenti e quindi avete solo preso la miglior decisione che potevate. Educatamente chiedete che l'argomento non sia ripreso a meno che qualcuno abbia qualcosa di davvero nuovo da dire, quindi stare fuori dalla discussione e sperate che muoia di morte naturale.

Qualcuno potrebbe suggerire una votazione per scegliere un modo piuttosto che un altro. Potete farlo se volete, ma personalmente non penso che votare sia una soluzione soddisfacente in questo caso. La penalità per qualcuno che è sorpreso da questo comportamento è così grande (mandare accidentalmente una email privata ad una mailing list pubblica) e l'inconveniente per gli altri è talmente leggero (doversi ricordare qualche volta di rispondere a tutta la mailing list invece che solo a voi), che non è chiaro perchè la maggioranza, anche se rimane la maggioranza, dovrebbe poter esporre la minoranza a tale rischio.

Non ho trattato tutti gli aspetti del problema qui, solo quelli che sembravano di primaria importanza. Per una discussione completa, vedete questi due documenti 'canonici', che sono quelli che la gente cita sempre quando affrontano questo dibattito:

- **Lasciate Reply-to solo**, *by Chip Rosenthal*

http://www.unicom.com/pw/reply-to-harmful.html

- **Mettete Reply-to alla mailing list**, *di Simon Hill*

  http://www.metasystema.net/essays/reply-to.mhtml

Nonostante la debole preferenza indicata sopra, non penso che ci sia una risposta "giusta" a questo problema, e participo felicemente a molte mailing list che *configurano* il Reply-to. La cosa più importante che possiate fare è di implementare un modo o l'altro da subito, cercando di non essere coinvolti in questo dibattito in seguito.

Due fantasie

Un giorno, qualcuno avrà la brillante idea di implementare un tasto *reply-to-list* nel software di lettura posta. Userebbe qualcuno degli header menzionati prima per capire l'indirizzo della mailing list, e quindi invierebbe la risposta direttamente alla mailing list soltanto, lasciando fuori ogni altro indirizzo di destinatazione, dato che la maggior parte dei quali è comunque iscritto nella lista. Infine, altri software di email adotteranno tale funzionalità e tutta questa discussione scomparirà. (Veramente, il sofware di posta Mutt [http://www.mutt.org/] offre questa funzionalità.[2])

Una soluzione persino migliore per l'occultamento del Reply-to sarebbe una preferenza dell'iscritto. Coloro che vogliono la mailing list con il Reply-to nascosto potrebbero chiederlo, e coloro che non lo vogliono potrebbero chiedere di lasciare il Reply-to com'è. Comunque, non sono a conoscenza di nessun software di gestione di mailing list che offra questa opzione per l'iscritto. Per ora, pare di essere fermi ad un'impostazione di tipo globale. [3]

Archiviazione

I dettagli tecnici della configurazione della mailing list sono specifici rispetto al software che è in uso, e sono fuori dall'ambito di questo libro. Al momento della scelta o della configurazione dell'archiviatore, considerate queste caratteristiche:

Aggiornamento veloce
: La gente a volte vuole far riferimento ad un messaggio in archivio inviato nell'ultima ora o due. Se possibile, l'archiviatore dovrebbe archiviare ogni messaggio istantaneamente, così che per il momento in cui il messaggio compare nella mailing list, è già presente negli archivi. Se questa opzione non è disponibile, cercate almeno di provare a fare in modo che l'archiviatore si aggiorni all'incirca ogni ora. (per default, alcuni archiviatori compiono i loro processi di aggiornamento una volta ogni notte, ma in pratica è un tempo troppo grande per una mailing list attiva)

Stabilità dei riferimenti
: Una volta che un messaggio è memorizzato ad una particolare URL, dovrebbe rimanere accessibile a quella stessa URL per sempre, o per il maggior tempo possibile. Anche se gli archivi saranno ricostituiti, ristabiliti da backup o messi a posto in qualche altro modo, tutte le URL che sono già state rese pubblicamente disponibili dovrebbero rimanere invariate. Riferimenti stabili rendono possibile per i motori di ricerca Internet di indicizzare gli archivi, che è una manna per gli utenti in cerca di risposte. Referimenti stabili sono anche importanti perchè il bug tracker spesso fa riferimento ai messaggi della mailing list (vedi sezione chiamata «Tracciamento dei bug») più avanti in questo capitolo o in documenti da altri progetti.

[2]Poco dopo la pubblicazione di questo libro, Michael Bernstein [http://www.michaelbernstein.com/] mi scrisse per dirmi : "Ci sono altri software di mail che implementano la funzionalità reply-to-list oltre a Mutt. Per esempio, Evolution ha questa funzionalità come scorciatoia da tastiera, ma non come pulsanti (Ctrl+L)."

[3]Da quando ho scritto ciò, ho saputo che c'è almeno un sistema di gestione che offre tale funzionalità: Siesta [http://siesta.unixbeard.net/]. Vedete anche questo articolo a riguardo: http://www.perl.com/pub/a/2004/02/05/siesta.html

Idealmente, il software di gestione della mailing list dovrebbe includere in un header quando distribuisce il messaggio ai destinatari, una URL di archiviazione del messaggio, o almeno una porzione della URL specifica per il messaggio. In questo modo la gente che ha una copia del messaggio possa essere in grado di sapere la sua collocazione nell'archivio senza dovere veramente visitare l'archivio, il che sarebbe d'aiuto dato che ogni operazione che coinvolge il proprio web browser automaticamente occupa del tempo. Non so se ogni software di gestione include tale funzionalità; sfortunatamente, quelli che ho usato no. Comunque, è qualcosa da cercare (o, se scrivete software per mailing list, è una funzionalità la cui implementazione è, per favore, da considerare).

Backup
: Dovrebbe essere ragionevolmente ovvio come fare il backup degli archivi, e il modo di ristabilirli non dovrebbe essere troppo difficile. In altre parole, non trattate il vostro archiviatore come una scatola nera (black box). Voi (o qualcuno nel vostro progetto) deve sapere dove i messaggi sono memorizzati, e come rigenerare l'archivio corrente dal backup se dovesse essere necessario. Questi archivi sono dati preziosi—un progetto che li perde, perde una buona parte della sua memoria collettiva.

Supporto ai thread
: Dovrebbe essere possibile andare da ogni singolo messaggio al *thread* (gruppo di messaggi correlati) di cui il messaggio originale fa parte. Ogni thread dovrebbe avere anche la propria URL, diversa dalle URL dei messaggi singoli che lo compongono.

Ricercabilità
: Un archiviatore che non supporta la ricerca—sui contenuti dei messaggi, così come sugli autori e oggetti—è praticamente inutile. Va notato che alcuni archiviatori supportano la ricerca semplicemente appoggiandosi per l'elaborazione su di un motore di ricerca esterno come Google [http://www.google.com/]. Questo è accettabile, ma il supporto alla ricerca diretta è solitamente meglio rifinito, perchè permette a chi cerca, per esempio, di specificare che la chiave di ricerca appaio nell'oggetto piuttosto che nel corpo del messaggio.

Cosa scritto sopra è solo una lista di punti tecnici per aiutarvi a valutare e configurare un archiviatore. Fare in modo che la gente davvero *usi* l'archiviatore per il bene del progetto, è discusso nei prossimi capitoli, in particolare sezione chiamata «Uso Ben Visibile degli Archivi».

Software

Qui di seguito ci sono alcuni strumenti open source per gestire mailing list e archiviazione. Se il sito dove è ospitato il vostro progetto ha già una configurazione di default, allora potreste non avere mai bisogno di decidere sull'uso di nessun strumento. Ma se dovete installarne uno da voi, ci sono alcune possibilità. Quelli che ho effettivamente usato sono Mailman, Ezmlm,MHonArc, e Hypermail, ma questo non vuol dire che altri non siano pure buoni (e di sicuro ci saranno altri strumenti là fuori che non ho trovato, quindi non prendete questa come una lista completa).

Software di gestione di mailing list:

- **Mailman** — http://www.list.org/

  (contiene un archiviatore, e ha la possibilità di aggiungerne di esterni.)

- **SmartList** — http://www.procmail.org/

  (Fatto per essere usato con il sistema di elaborazione email Procmail.)

- **Ecartis** — http://www.ecartis.org/

- **ListProc** — http://listproc.sourceforge.net/
- **Ezmlm** — http://cr.yp.to/ezmlm.html

  (Progettato per lavorare con il sistema di consegna mail Qmail [http://cr.yp.to/qmail.html])
- **Dada** — http://mojo.skazat.com/

  (Nonostante i bizzarri tentativi di nasconderlo, questo è free software, rilasciato sotto la licenza GNU General Public License. Contiene anche un archiviatore.)

Software di archiviazione mailing list:

- **MHonArc** — http://www.mhonarc.org/
- **Hypermail** — http://www.hypermail.org/
- **Lurker** — http://sourceforge.net/projects/lurker/
- **Procmail** — http://www.procmail.org/

  (Software complementare a SmartList, questo è un sistema generale di processamento email che può, apparentemente, essere configurato con archiviatore.)

Controllo di versione

Un *sistema di controllo di versione* (version control system) (o *sistema di controllo di revisione*) è una combinazione di tecnologie e procedure per tenere traccia e controllare i cambiamenti dei file di un progetto, in particolare del codice sorgente, della documentazione e delle pagine web. Se non avete mai usato il controllo di versione prima, la prima cosa che dovreste fare è cercare qualcuno che l'abbia usato e coinvolgerlo nel vostro progetto. In questi tempi, chiunque si aspetterà che almeno il codice sorgente del vostro progetto sia sotto controllo di versione, e probabilmente non prenderà il progetto seriamente se questo non usa il controllo di versione con almeno le minime competenze.

La ragione per cui il controllo di versione è così universale è che aiuta potenzialmente ogni aspetto della gestione di un progetto: comunicazioni tra sviluppatori, gestione dei rilasci, gestione dei bug, stabilità del codice e tentativi di sviluppo sperimentali, e attribuzione e autorizzazione di cambiamenti da parte di alcuni sviluppatori. Il controllo di versione mette a disposizione un centro di coordinamento per tutte queste aree. Il cuore del controllo di versione è la a central coordinating force among all of these areas. The core of *gestione dei cambiamenti* (change management) che identifica ogni cambiamento apportato ai file del progetto, annotando ogni cambiamento con metadati quali la data di modifica e l'autore e quindi replicando questi fatti a chiunque chieda, in qualunque modo sia chiesto. E' un meccanismo di comunicazione dove un cambiamento è l'unità base di informazione.

Questa sezione non discute tutti gli aspetti dell'uso del controllo di versione. E' una cosa così pervasiva che deve essere trattata per argomento lungo tutto il libro. Qui ci concentreremo sulla scelta e sulla configurazione di un sistema di controllo di versione in modo da incoraggiare lo sviluppo cooperativo.

Vocabolario del controllo di versione

Questo libro non può insegnarvi come usare il controllo di versione se non l'avete mai usato prima, ma sarebbe impossibile discuterne senza alcune parole chiave. Queste parole sono utili indipendentemente da ogni sistema di controllo di versione: sono i nomi e i verbi di base della collaborazione in rete, e saranno usati in modo generico lungo il resto di questo libro. Anche se non ci fossero sistemi

di controllo di versione, il problema della gestione delle modifiche rimarrebbe, e queste parole ci forniscono un linguaggio per parlare concisamente di questo problema.

> **"Versione" contro "Revisione"**
>
> La parola *versione* è a volte usata come un sinonimo di "revisione", ma non la userò in questo libro, perchè è troppo facilmente confusa con "versione" nel senso di una versione di un software —cioè, il rilascio o numero di edizione, come in "Versione 1.0". Tuttavia, dal momento che la frase "controllo di versione" è già standard, continuerò ad usarla come sinonimo di "controllo di revisione" e "controllo delle modifiche".

commit
: Fare una modifica al progetto; più formalmente, archiviare una modifica al progetto nel database del controllo di versione in modo che possa essere incluso nei futuri rilasci del progetto. "Commit" può essere usato come nome e come verbo. Come nome, è essenzialmente un sinonimo di "cambiamento". Per esempio: "Ho appena commitato una fiz per il crollo del server che gli addetti ai bug hanno segnalato su Mac OS X. Jay, potresti per favore rivedere il commit e controllare che lì non stia usando male l'allocatore?"

messaggio di log
: Un po' di commento allegato ad ogni commit, descrivendo la natura e lo scopo del commit. I messaggi di log sono tra i documenti più importanti di ogni progetto: sono il ponte tra il linguaggio altamente tecnico dei cambiamenti individuali di codice e il linguaggio,più orientato all'utente, delle funzionalità, del fissaggio di bug e del progresso del progetto. Più avanti in questa sezione, vedremo modi di distribuire messaggi di log alla audience adeguataa; inoltre, sezione chiamata «La Tradizione della Codifica» in Capitolo 6, *Comunicazione* discute i modi per incoraggiare i contributori a scrivere messaggi di log concisi e utili.

update
: Richiedere che le modifiche (commit) degli altri siano inclusi nella vostra copia locale del progetto; cioè rendere la vostra copia aggiornata. Questa operazione è molto comune; la maggior parte degli sviluppatori aggiornano il loro codice diverse volte al giorno, così da sapere che stanno usando all'incirca la stessa cosa degli altri sviluppatori, e così che se vedono un bug, possano essere abbastanza sicuri che non sia stato già fissato. Per esempio:"Hey, ho notato che il codice di indicizzazione scarta sempre l'ultimo byte. Questo è un bug?" "Si, ma è stato fissato la scorsa settimana—prova ad aggiornarti, dovrebbe sparire."

repository
: Un database in cui sono memorizzate le modifiche. Alcuni sistemi di controllo di versione sono centralizzati: c'è un solo repository principale, che memorizza tutte le modifiche del progetto. Altri sono decentralizzati: ogni sviluppatore ha il proprio repository, e le modifiche sono scambiate arbitrariamente avanti e indietro tra i repository Il sistema di controllo di versione mantiene traccia delle dipendenze tra le modifiche, e quando è il momento del rilascio, un particolare insieme di modifiche viene approvato per il rilascio. La domanda su quale sia migliore tra i sistemi centralizzati e decentralizzati è una delle lunghe guerre sante dello sviluppo software; cercate di non cadere nella trappola di discuterne sulle mailing list del vostro progetto.

checkout
: Il processo di ottenere una copia del progetto da un repository. Un checkout solitamente produce un albero di directory chiamato "copia di lavoro" (vedi sotto), da cui le modifiche possono essere committate indietro al repository originale. In alcune versioni di sistemi di controllo decentralizzati, ogni copia di lavoro è essa stessa un repository, e i cambiamenti possono essere presi o richiesti da ogni repository che vuole accettarli.

copia di lavoro
: L'albero di directory privato di uno sviluppatore contenente i file di codice sorgente del progetto, e possibilmente le sue pagine web e altri documenti. Una copia di lavoro contiene anche un po' di metadati gestiti dal sistema di controllo di versione, che dice alla copia di lavoro da quale repository viene, quali "revisioni" (vedi sotto) dei file sono presenti eccetera. Generalmente ogni sviluppatore ha la sua copia di lavoro, in cui fa e prova modifiche, e da cui fa commit.

revisione, *modifica*, *insieme di modifiche*
: Una "revisione" è solitamente un'incarnazione specifica di un particolare file o directory. Per esempio, se il progetto presenta la revisione 6 del file F, e poi qualcuno fa il commit di una modifica a F, ciò crea la revisione 7 di F. Alcuni sistemi usano anche "revisione","modifica", "insieme di modifiche" per riferirsi ad un insieme di modifiche comittate insieme come una unità concettuale.

  Questi termini hanno occasionalmente significati tecnici distinti in diversi sistemi di controllo di versione, ma l'idea generale è sempre la stessa: danno modo di parlare in modo preciso di punti esatti nella storia di un file o di un insieme di file (per dire, immediatamente prima e dopo la riparazione di un bug). Per esempio : "Si, ha fissato il bug nella versione 10" oppure "ha fissato il bug nella versione 10 del file foo.c"

  Quando si parla di un file o di una collezione di file senza specificare la revisione, generalmente si assume che si intenda la revisione più recente disponibile.

diff
: Una rappresentazione testuale di una modifica. Una diff mostra quali righe sono state cambiate e come, più alcune righe del contesto attorno on entrambi i lati. Uno sviluppatore che è già familiare con un po' di codice può solitamente leggere una diff rispetto a quel codice e capire cosa ha fatto la modifica, e persino scoprire bug.

tag
: Un'etichetta per un particolare insieme di file ad una specifica revisione. I tag sono solitamente usati per preservare "istantanee" interessanti del progetto. Per esempio, un tag è solitamente fatto per ogni pubblico rilascio, così che one possa ottenere, direttamente dal sistema di controllo di versione, l'insieme esatto dei file/revisioni compresi in quel rilascio. Nomi di tag comuni sono cose del tipo `Release_1_0`, `Delivery_00456`, eccetera.

Ramo (branch)
: Una copia del progetto, sotto controllo di versione ma isolata, così che i cambiamenti fatti nel ramo non interferiscono con il resto del progetto e viceversa, tranne quando i cambiamenti sono deliberatamente uniti da un lato all'altro (vedi sotto). I rami sono anche noti come "linee di sviluppo". Anche se un progetto non ha rami specifici, lo sviluppo è comunque considerato accadere nel "ramo principale", detto anche "linea principale" o "*tronco*" (trunk).

  i rami offrono un modo per isolare le diverse linee di sviluppo l'una dall'altra. Per esempio, un ramo può essere usato per sviluppo sperimentale che sarebbe troppo destabilizzante sul tronco principale. O al contrario, un ramo può essere usato come luogo per stabilizzare un nuovo rilascio. Durante il processo di rilascio, lo sviluppo regolare dovrebbe continuare ininterrotto nel tronco principale del repository; nel frattempo, nel ramo del rilascio, nessuna modifica è permessa tranne quelle approvate dal manager del rilascio. In questo modo, fare un rilascio non ha bisogno di interferire con lo sviluppo in corso. Vedi See sezione chiamata «Use branch per evitare colli di bottiglia» più avanti in quesot capitolo per una discussione più dettagliata della ramificazione.

Unire(merge)
: Spostare una modifica da un ramo ad un altro. Ciò include l'unione(merging) tra il tronco principale e qualche altro ramo o viceversa. Infatti, questi sono i tipi più comuni di union; è raro portare una modifica tra due rami non principali. Vedi sezione chiamata «Singolarità dell'informazione» per più dettagli su questo tipo di unione.

"Unire" ha altro significato correlato: è ciò che il sistema di controllo di versione fa quando vede che due persone hanno cambiato lo stesso file ma in modo non contrastante. Dato che le due modifiche non interferiscono l'una con l'altra, quando una delle persone aggiorna la propria copia del file (che già contiene le proprie modifiche), le modifiche dell'altra persona saranno unite automaticamente. Questo è molto comune, specialmente in progetti dove molte persone lavorano sullo stesso codice. Quando due modifiche diverse contrastano, il risultato è un "conflitto"; vedere sotto.

conflitto
: Cosa succede quando due persone provano a fare modifiche diverse nello stesso punto del codice. Tutti i sistemi di controllo di versione trovano automaticamente i conflitti, e notificano ad almeno uno degli umani coinvolti che i suoi cambiamenti sono in conflitto con quelli di qualcun altro. E' quindi lasciato all'umano *risolvere* il conflitto, e comunicare la soluzione al sistema di controllo di versione.

Blocco
: Un modo di dichiarare un'intenzione esclusiva di cambiare un certo file o directory. Per esempio, "Non posso fare il commit di tutti i cambiamenti alle pagine web ora. Sembra che Alfredo le abbia bloccate tutte mentre mette a posto le immagini di sfondo." Non tutte le versioni di sistemi di controllo offrono anche questa possibilità di blocco, e quelli che lo fanno, non tutti richiedono di usare questa funzionalità. Questo perchè lo sviluppo parallelo, simultaneo è la norma, e negare alla gente l'accesso ai file è (solitamente) contrario a questo ideale.

  I sistemi di controllo di versione che richiedono il blocco per fare i commit sono detti usare il modello *blocca-modifica-sblocca* (lock-modify-unlock). Un'ottima e profonda spiegazione e confronto dei due modelli può essere trovata qui http://svnbook.red-bean.com/svnbook-1.0/ch02s02.html. In generale, il modello copia-modifica è migliore per lo sviluppo open source, e tutti i sistemi di controllo di versione discussi in questo libro supportano tale modello.

Scegliere un sistema di controllo di versione

I due sistemi di controllo di versione più popolari nel mondo del free software sono *Concurrent Versions System* (*CVS*, http://www.cvshome.org/) e *Subversion* (*SVN*, http://subversion.tigris.org/).

CVS c'è da molto tempo. La maggior parte degli sviluppatori esperti ne hanno già familiarità, fa più o meno cosa di cui avete bisogno, e dal momento che è stato popolare per molto tempo, probabilmente non finirete in nessun lungo dibattito sul fatto che sia la scelta giusta o meno. Comunque, CVS ha alcuni svantaggi. Non fornisce un modo semplice per riferirsi a modifiche che coinvolgono diversi file; non vi permette di rinominare o copiare file sotto il controllo di versione (quindi se avete bisogno di riorganizzare il vostro albero del codice dopo l'inizio del progetto, può essere una vera sofferenza); ha un mediocre supporto al merging (unione); non maneggia molto bene grossi file o file binari; e alcune operazioni sono lente quando un gran numero di file è coinvolto.

Nessuna delle crepe di CVS è fatale ed è tuttora abbastanza popolare. Ad ogni modo, negli ultimi anni il più recente Subversion ha guadagnato terreno, soprattutto nei progetti più nuovi.[4]. Se state iniziando un nuovo progetto, raccomando Subversion.

Dall'altro lato, dato che sono coinvolto nel progetto Subversion, la mia oggettività può essere giustamente messa in discussione. E negli ultimi anni alcuni nuovi sistemi di controllo open source hanno fatto la loro comparsa. Appendice A, *Sistemi di Controllo di Versione Liberi* elenca tutti quelli di cui sono a conoscenza, in un ordine approssimativo di popolarità. Come la lista mostra, decidere su un sistema di controllo di versione può facilmente diventare un progetto di ricerca lungo una vita. Magari

[4]Vedi http://cia.vc/stats/vcs e http://subversion.tigris.org/svn-dav-securityspace-survey.html per prove di questa crescita.

vi risparmierete la decisione perchè sarà presa per voi dal sito che vi ospita. Ma se dovete scegliere, consultatevi con gli altri sviluppatori, chiedete in giro per vedere con quale la gente ha esperienza, infine prendetene uno e andate avanti con quello. Ogni sistema di controllo di versione stabile e pronto alla produzione andrà bene; non dovete preoccuparvi troppo di fare una scelta totalmente sbagliata. Se proprio non riuscite a farvi un'idea, allora usate Subversion. E' abbastanza facile da imparare e probabilmente rimmarrà standard per almeno un po' di anni.

Usare un sistema di controllo di versione

I consigli di questa sezione non sono indirizzati a particolari versioni di sistemi di controllo di versione, e dovrebbero essere facili da implementare in ognuno di loro. Consultate la documentazione specifica del vostro sistema di versione per i dettagli.

Tenere tutto sotto controllo di versione

Tenere non solo il codice sorgente del vostro progetto sotto controllo di versione, ma anche le sue pagine web, la documentazione, FAQ, note di progetto, e qualsiasi altra cosa che la gente potrebbe voler modificare. Teneteli vicino al codice sorgente, nello stesso albero di repository. Ogni brandello di informazione che valga la pena scrivere vale la pena mettere sotto controllo—cioè ogni brandello di informazione che potrebbe cambiare. Cose che non cambiano dovrebbero essere archiviate e non messe sotto controllo di versione. Per esempio, le email, quando pubblicate, non cambiano (a meno che diventino parte di un documento più grande e in evoluzione).

La ragione per cui è importante mettere tutto insieme cio che è sotto controllo di versione è che in questo modo la gente deve imparare solo un meccanismo per inviare le modifiche. Qualche volta un contributore inizierà con l'apportare modifiche a pagine web o alla documentazione, spostandosi su piccoli contributi di codice in un secondo momento, per esempio. Quando il progetto usa lo stesso sistema per ogni tipo di contribuzione, la gente deve imparare solo una volta. Inoltre le nuove funzionalità possono essere contribuite insieme con le modifiche alla documentazione, fare il branching del codice farà fare il branch anche alla documentazione eccetera.

Non tenete i *file generati*(generated files) sotto controllo di versione. Non sono dati veramente editabili, dato che sono generati programmaticamente da altri file. Per esempio, alcuni sistemi di build creano dei `configure` basati sul template `configure.in`. Per fare un cambiamento al `configure`, bisognerebbe cambiare il `configure.in` e poi rigenerare; quindi solo il template `configure.in` è un "file editabile". Mettete sotto controllo di versione solo i template—se lo fate anche con i file generati, la gente si dimenticherà inevitabilmente di rigenerare quando fanno il commit di una modifica al template, e le inconsistenze risultanti causeranno confusione a non finire.[5]

La regola che tutti i dati editabili vadano tenuti sotto controllo di versione ha una sfortunata eccezione: il tracciatore dei bug. I database di bug contengono una gran quantità di dati editabili, ma per ragioni tecniche generalmente non possono memorizzare questi dati nel sistema di controllo principale. (Comunque, alcuni tracciatori hanno funzionalità di versionamento primitive, indipendenti dal repository principale del progetto.)

Navigabilità

Il repository del progetto dovrebbe essere navigabile sul web. Questo non significa solo la possibilità di vedere le ultime revisioni dei file del progetto, ma di andare indietro nel tempo e vedere le versioni precedenti, vedere le differenze tra le revisioni, leggere i messaggi di log per le modifiche selezionate eccetera.

[5]Per una differente opinione sulla questione dei file `configure`, vedete il post di Alexey Makhotkin "*configure.in and version control*" qui http://versioncontrolblog.com/2007/01/08/configurein-and-version-control/.

La navigabilità è importante perchè è un portale leggero verso i dati del progetto. Se il repository non può essere visto attraverso un browser web, allora qualcuno che vuole esaminare un file particolare (per esempio, per vedere se una certa fix è stata fatta in quel codice) dovrebbe prima installare localmente il software del client del controllo di versione, il che muterebbe la semplice richiesta da un'attività di due minuti in una da mezz'ora o più.

La navigabilità implica anche URL canoniche per vedere le specifiche revisioni dei file, e per vedere le ultime revisioni in ogni momento. Questo può essere molto utile nelle discussioni tecniche o quando si rimanda la gente alla documentazione. Per esempio, invece di dire "Per suggerimenti sul debugging del server. vedi il file www/hacking.html nella tua copia di lavoro," uno può dire "Per suggerimenti sul debugging del server, vedi *http://subversion.apache.org/docs/community-guide/*," dando una URL che punti sempre all'ultima revisione del file `hacking.html`. L'URL è meglio perchè è totalmente non ambigua, ed evita la questione sul fatto che il destinatario abbia o meno una copia di lavoro aggiornata.

Alcuni sistemi di controllo di versione includono meccanismi precostitutiti di navigazione del repository, mentre altri si affidano a strumeti di terze parti. Tre di tali strumenti sono *ViewCVS* (http://viewcvs.sourceforge.net/), *CVSWeb* (http://www.freebsd.org/projects/cvsweb.html), e *WebSVN* (http://websvn.tigris.org/). Il primo funziona sia con CVS che con Subversion, il secondo con CVS soltanto, e il terzo solo con Subversion.

Email di commit

Ogni commit sul repository dovrebbe generare un email che mostra chi ha fatto le modifiche, quando, quali file e directory sono cambiati, e come sono cambiate. L'email dovrebbere essere mandata su una speciale mailing list dedicata alle email di commit, separata dalle altre mailing list a cui la gente scrive. Gli sviluppatori e le altre parti interessate dovrebbero essere incoraggiati a sottoscrivere la mailing list dei commit, dato che è il modo più efficace per stare dietro a cosa sta succedendo nel progetto al livello del codice. Oltre dagli ovvi benefici tecnici della revisione di pari (peer review) (see sezione chiamata «Praticare una Visibile Revisione del Codice»), le email di commit aiutano a creare un senso di comunità perchè stabiliscono un ambiente condiviso in cui la gente può reagire agli eventi (i commit) che sanno essere visibili anche ad altri.

Le specifiche su come creare le email di commit cambieranno a seconda della vostra versione di sistema di controllo di versione, ma solitamente c'è uno script o qualcosa di già preparato per farlo. Se state avendo problemi a trovarlo, provate a guardare la documentazione sugli *hooks* (uncini), in particolare il *post-commit hook*, detto anche il *hook loginfo* in CVS. Gli hook post-commit sono strumenti generali per il lancio di attività automatiche in risposta ai commit. L'hook è lanciato da un commit individuale, viene fornito di tutta l'informazione sul commit, ed è poi libero di usare questa informazione per fare qualcosa—per esempio, mandare un'email.

Con sistemi di email di commit pre-confezionata, potreste volere modificare qualcuno dei loro comportamenti di defult:

1. Alcuni sistemi non includono i diff nella mail, ma invece forniscono una URL per vedere le modifiche sul web usando il sistema di navigazione del repository. Mentre è bene fornire la URL, così che si possa fare riferimento alle modifiche anche in seguito, è anche *molto* importante che le email di commit includano i diff stessi. Leggere le email è già parte delle abitudini della gente, quindi se il contenuto della modifica è visibile proprio lì nella email di commit, gli sviluppatori esamineranno il commit al volo, senza lasciare il loro programma di lettura di posta. Se devono cliccare su di una URL per esaminare le modifiche, la maggior parte non lo farà, perchè richiede una nuova azione invece di continuare quella che stanno già facendo. Inoltre, se il revisore vuole chiedere qualcosa riguardo alla modifica, è molto più semplice premere il tasto di risposta con testo e annotare il diff in questione rispetto alla visita di una pagina web e laboriosamente fare copia e incolla di parti del diff dal browser web al programma di email.

(Ovviamente, se il diff è grande, come quando una gran quantità di nuovo codice è stato aggiunto al repository, allora ha senso omettere il diff e offrire solo la URL. La maggior parte dei sistemi possono fare automaticamente questo tipo di limitazione. Se il vostro non può, allora è in ogni caso meglio includere i diff, e vivere con l'occasionale mail smisurata, piuttosto che lasciare del tutto fuori i diff. La revisione e il commento sono la pietra angolare dello sviluppo cooperativo, troppo importante per farne a meno.)

2. Le email di commit devono avere l'header Reply-to impostato sulla normale mailing list di sviluppo, non sulla mailing list dei commit. Cioè quando qualcuno esamina un commit e scrive una risposta, tale risposta deve essere automaticamente diretta alla mailing list di sviluppo, dove le questioni tecninche sono solitamente discusse. Ci sono alcune ragioni per questo. Primo, vorrete tenere tutte le discussioni tecniche su una mailing list, perchè è ciò che la gente si aspetta, e per mantenere un solo archivio di ricerca. Secondo, ci potrebbero essere parti interessate non iscritte alla mailing list di commit. Terzo, la mailing list di commit si presenta come un servizio per vedere i commit, non per vedere commit *e* occasionali discussioni tecniche. Chi ha sottoscritto la mailing list di commit l'ha fatto per nient'altro che le email di commit; mandare loro altro materiale attraverso la mailing list violerebbe un contratto implicito. Quarto, la gente spesso scrive programmi che leggono le email di commit e processano i risultati (per esempio per mostrarli su una pagina web). Questi programmi sono fatti per gestire email di commit formattate in maniera consistente, non email scritte senza formattazione da umani.

   Notate che questo consiglio di impostare il Reply-to non contraddice le raccomandazioni fatte in sezione chiamata «Il grande dibattito sul 'Rispondi A'» precedentemente in questo capitolo. Va sempre bene per il *mittente* di un messaggio impostare il Reply-to. In questo caso, il mittente è il sistema stesso di controllo di versione, e imposta il Reply-to per indicare che il posto adeguato per rispondere è la mailing list di sviluppo e non quella di commit.

CIA: un altro meccanismo di pubblicazione delle modifiche

Le email di commit non sono il solo modo di propagare le notizie di modifica. Recentemente, è stato sviluppato un altro meccanismo chiamato CIA (http://cia.navi.cx/). CIA è un aggregatore e distributore di statistiche di commit in tempo reale. L'uso più diffuso di CIA è l'invio di notifiche di commit ai canali IRC, così che la gente loggata in questi canali vede succedere i commit in tempo reale. Anche se una utility in qualche modo meno tecnica delle email di commit, dato che gli osservatori possono o meno essere lì quando una notifica compare in IRC, questa tecnica è una grandissima *social* utility. La gente ha la sensazione di essere parte di qualcosa di vivo e attivo e sente che possono vedere fare il progresso proprio di fronte ai loro occhi.

Il modo in cui funziona è l'invocazione di CIA dal vostro hook di post-commit. Il notificatore formatta le informazioni di commit in un messaggio XML, e invia ad un server centrale (tipicamente `cia.navi.cx`). Questo server quindi distribuisce le informazioni di commit agli altri forum.

CIA può anche essere configurato per mandare feed RSS [http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html]. Vedete la documentazione qui http://cia.navi.cx/ per dettagli.

Per vedere un esempio di CIA in azione, impostate il vostro client IRC a `irc.freenode.net`, canale `#commits`.

Use branch per evitare colli di bottiglia

Gli utenti non esperti del controllo di versione sono a volte un po' spaventati da branch e merge. Questo è probabilmente uno degli effetti secondari della popolarità di CVS: la sua interfaccia per branch e

merge è in qualche modo controintuitiva, così molta gente ha imparato ad evitare totalmente queste operazioni.

Se siete tra queste persone, decidetevi ora a sconfiggere ogni paura che possiate avere e prendetevi tempo per imparare come fare branch e merge. Non sono operazioni difficili, una volta che vi siete abituati, e diventano sempre più importanti man mano che il progetto acquista più sviluppatori.

I branch sono preziosi perchè fanno diventare una risorsa scarsa —una stanza di lavoro nel codice del progetto—in una abbondante. Normalmente, tutti gli sviluppatori lavorano assieme nello stesso ambiente, costruendo lo stesso castello. Quando qualcuno vuole aggiungere un nuovo ponte levatoio, ma non può convincere tutti gli altri che sia un miglioramento, fare un branch render possibile spostarsi in un angolo e provare. Se lo sforzo ha successo, può invitare altri sviluppatori ad esaminare il risultato. Se tutti sono d'accordo che il risultato è buono, possono dire al sistema di controll di versione di spostare (merge) il ponte levatoio dal castello branch a quello principale.

E' facile vedere come questa possibilitò aiuti lo sviluppo collaborativo. La gente ha bisogno della libertà di provare nuove cose senza sentirsi come se stessero interferendo con il lavoro degli altri. Altrettanto importante, ci sono volte in cui il codice ha bisogno di essere isolato dalla solita pentola, per fissare un bug o stabilizzare una release (vedi sezione chiamata «Stabilizzare una Release» e sezione chiamata «Avere in Manutenzione più di Una Linea di Release» in Capitolo 7, *Confezione, Rilascio, e Sviluppo Quotidiano*) senza preoccuparvi del tracciamento di un bersaglio mobile.

Usate liberamente i branch, e incoraggiate gli altri ad usarli. Ma accertatevi anche che un dato branch sia solo attivo soltanto per il tempo necessario. Ogni branch è una piccola falla nell'attenzione della comunità. Anche coloro che non stanno lavorando in un branch mantengono comunque una visione periferica di cosa stia succedendo. Tale visione è sicuramente desiderabile, e le email di commit dovrebbero essere mandate per i commit del branch come per ogni altro commit. Ma i branch non devono diventare un meccanismo per dividere la comunità di sviluppo. Con rare eccezioni, il fine di molti branch dovrebbe essere qiello di fare il merge delle modifiche con il tronco principale e scomparire.

Singolarità dell'informazione

Fare merge ha un importante corollario: non bisogna mai fare commit della stessa modifica due vole. Cioè, una certa modifica deve entrare nel sistema di controllo di versione esattamente una volta sola. La revisione (o insieme di revisioni) in cui la modifica è entrata a fare parte è il suo unico identificatore da ora in poi. Se ha bisogno di essere applicata a branch diversi da quello in cui è entrata, allora dovrebbe essere unita (merge) dal suo punto originale di ingresso a queste altre destinazioni—in maniera opposta al commit di una modifica testualmente identica, che avrebbe lo stesso effetto sul codice, ma renderebbe l'archiviazione e la gestione dei rilasci impossibili.

Gli effetti pratici di questo consiglio cambiano da un sistema di controllo di versione all'altro. In alcuni sistemi, i merge sono eventi speciali, fondamentalmente diversi dai commit, e portano con loro particolari metadati. In altri, i risultati del merge sono committati allo stesso modo delle altre modifiche, tanto che lo strumento principale per distinguere un merge da un modifica è il messaggio di log. Nel messaggio di log di un merge, non ripetete il messaggio log della modifica originale. Invece, indicate solo che questo è un merge, e fornite la revisione che identifica la modifica originale, con al massimo un sommario di una frase sul suo effetto. Se qualcuno vuole vedere il messaggio di log intero, dovrà consultare la revisione originale.

La ragione per cui è importante evitare la ripetizione dei messaggi di log è che questi messaggi sono spesso modificati dopo che il commit è stato fatto. Se il messaggio di log di una modifica è ripetuto ad ogni destinazione di un merge, allora anche se qualcuno modificato il messaggio originale, lascierà tutte le ripetizioni non corrette—il che causerebbe solo confusione strada facendo.

Lo stesso principio vale per il disfacimento di una modifica. Se una modifica viene tolta dal codice, allora il messaggio di log per la revisiona deve solo dire che qualche specifica revisione è stata disfatta, *non* descrivere i cambi nel codice che risultano dalla revisione, dato che la semantica della modifica può essere ottenuta leggendo il messaggio di log e la modifica originali. Di certo il messaggio di log della revisione deve anche spiegare la ragione per cui la modifica è stata annullta, ma non deve riprendere nulla del messaggio di log originale della modifica. Se possibile, tornate indietro e modificate il messaggio di log originale della modifica per puntualizzare che è stata annullata.

Tutto ciò di cui sopra implicca che dovreste usare una sintassi consistente per fare riferimento alle revisioni. Ciò è utile non solo nei messaggi di log, ma anche nelle email, nel tracciatore di bug, e da ogni altra parte. Se state usando CVS, suggerisco "`path/to/file/in/project/tree:REV`", dove REV è il numero di revisione CVS tipo "1.76". Se state usando Subversion, la sintassi standard per la revisione 1729 è "r1729" (i path dei file non sono necessari perchè Subversion usa numeri di revisione globali). In altri sistemi, cìè solitamente una sintassi standard per esprimere il nome degli insiemi di modifiche. Qualunque sia la sintassi appropriata per il vostro sistema, incoraggiate la gente ad usarla quando fa riferimento alle modifiche. Espressioni consistenti di nomi delle modifiche rende la gestione del progetto molto più facile (come vedremo in Capitolo 6, *Comunicazione* e Capitolo 7, *Confezione, Rilascio, e Sviluppo Quotidiano*), e dato che buona parte della gestione sarà svolta da volontari, deve essere il più semplice possibile.

Vedi anche sezione chiamata «Le Releases e Lo Sviluppo Quotidiano» in Capitolo 7, *Confezione, Rilascio, e Sviluppo Quotidiano*.

Autorizzazione

La maggior parte dei sistemi di controllo di versione offre una funzionalità dove a certe persone può essere permesso o vietato fare il commit in specifiche sotto-aree del repository. Seguendo il principio che quando la gente quando ha un martello in mano, si guarda attorno per cercare i chiodi, molti progetti usano questa opzione senza remore, prudentemente lasciando accesso alla gente solo alle aree in cui un loro commit è approvato, e cercando di essere sicuri che non facciano commit da nessun'altra parte. (Vedi sezione chiamata «Quelli Che Fanno gli Invii» in Capitolo 8, *Gestire i Volontari* per come i progetti decidono chi può fare commit dove.)

E' probabilmente un po' doloroso esercitare un così stretto controllo, ma una potica più rilassata va anche bene. Alcuni progetti semplicemente usano un sistema d'onore: quando ad una persona viene riconosciuto l'accesso per il commit, anche per sotto-aree del repository, cosa questa veramente riceve è una password che permette di fare commit ovunque nel progetto. Gli è solamente rischiesto di limitare i commit a quell'area. Ricordate che non c'è nessun rischio qui: in un progetto attivo, tutti i commit sono esaminati in ogni caso. Se qualcuno fa un commit che non deve, gli altri lo vedranno e diranno qualcosa. Se una modifica deve essere disfatta,è abbastanza semplice— tutto è comunque sotto controllo di versione quindi basta invertire.

Ci sono diversi vantaggi nell'approccio rilassato. Primo, dato che gli sviluppatori si allargono in altre aree (cosa che solitamente succede se rimangono nel progetto), non c'è nessuno sforzo amministrativo per garantire loro privilegi più ampi. Una volta che la decisione è fatta, la persona deve solo iniziare a fare commit nella nuova area.

Secondo, questo allargamento può essere fatto in modo più fine. Generalmente, uno che fa commit in area X che vuole allargarsi nell'area Y inizierò pubblicando patch su Y e chiedendone revisione. Se qualcuno che già ha l'accesso per fare commit in Y vede la patch e la approva, si può semplicemente dire a chi l'ha sottomessa di fare il commit della modifica direttamente (menzionando il nome di chi l'ha ricevuta/approvata nel messaggio, ovviamente). In questo modo, il commit verrà dalla persona che ha fisicamente scritto la modifica, il che è preferibile sia dal punto di vista di gestione dell'informazione che dal punto di vista del riconoscimento.

Infine, e forse più importante, usare il sistema d'onore incoraggia un'atmosfera di fiducia e mutuo rispetto. Dare a qualcuno l'accesso per il commit in un sottodominio è un'affermazione sulla sua preparazione tecnica—come dire: "Vediamo che hai le conoscenze per fare commit in un certo dominio, quindi fallo". Ma imporre uno stretto controllo di autorizzazione dice: "Non solo stabiliamo un limite alle tue capacità, ma siamo anche un po' sospettosi sulle tue *intenzioni*." Questo non è il tipo di affermazione che volete fare se potete evitarlo. Portare qualcuno che fa commit nel progetto è un'opportunità per introdurli al circolo della fiducia reciproca. Un buon modo per fare ciò e dare loro più potere di quello che dovrebbero usare, e poi informarli che sta a loro rimanere nei limiti accordati.

Il progetto Subversion ha lavorato con il sistema d'onore per più di quattro anni, con 33 persone che potevano fare commit ovunque e 43 limitatamente. L'unica distinzione che il sistema applicava è tra chi fa commit e chi no; ulteriori sottodivisioni sono mantenute solo dagli umani. Comunque non abbiamo mai avuto un problema con qualcuno che deliberatamente faceva commit fuori dal proprio dominio. Una o due volte ci sono stati degli innocenti equivoci riguardo ai limiti dei privilegi di commit, ma sono sempre stati risolti velocemente e amichevolmente.

Ovviamente in situazioni in cui l'autoregolamentazione non è possibile, dovete affidarvi a severi controlli di autorizzazione. Ma tali situazioni sono rare. Anche quando ci sono milioni di righe di codice e centinaia o migliaia di sviluppatori, un commit su qualunque modulo del codice dovrebbe comunque verificato da chi lavora su quel modulo, e possono riconoscere se qualcuno ha fatto il commit senza esserne autorizzato. Se la regolare revisione dei commit *non* funziona, allora il progetto ha problemi più grandi da affrontare piuttosto che il sistema di autorizzazione.

Riassumendo, non spendete troppo tempo armeggiando con il sistema di autorizzazione del controllo di versione, a meno che non abbiate delle specifiche ragioni per farlo. Solitamente non poterà molti vantaggi tangibili, mentre invece ci sono vantaggi nell'affidarsi al controllo umano.

Di sicuro niente di ciò vuole dire che le limitazioni in sè non sono importanti. Sarebbe male per un progetto incoraggiare le persone a fare commit in aree in cui non sono qualificate. Inoltre, in molti progetti un accesso pieno (non limitato) ai commit ha uno stato speciale: implica diritti di voto su questioni riguardanti il progetto intero. Questo aspetto politico del commit è discusso sezione chiamata «Chi Vota?» in Capitolo 4, *L'Infrastruttura Sociale e Politica*.

Tracciamento dei bug

Il tracciamento dei bug è un argomento ampio; vari suoi aspetti sono discussi lungo questo libro. Qui cercherò di concentrarmi principalmente su setup e considerazioni tecniche, ma per arrivarci, dobbiamo iniziare con una domanda di politica: esattamente quale tipo di informazione deve essere mantenuta in un tracciatore di bug?

Il termine *tracciatore di bug* (bug tracker) è fuorviante. Sistemi di tracciamento di bug sono anche usati frequentemente per tenere traccia delle richieste di nuove funzionalità, one-time task, patch non sollecitate— veramente ogni cosa che abbia stati di inizio e di fine distinti, con stati di transizione opzionali nel mezzo, e che accumuli informazioni durante il suo tempo di vita. Per questa ragione, i bug tracker sono anche chiamati *tracciatori di problemi* (issue trackers), *tracciatori di difetti* (defect trackers), *artifact trackers*, *tracciatori di richieste* (request trackers), *sistemi di segnalazione di problemi*, (trouble ticket systems) eccetera. Vedi Appendice B, *Bug Tracker Liberi* per un elenco di software.

In questo libro continuerò ad usare "bug tracker" per il software che fa il tracciamento, perchè è come la maggior parte della gente lo chiama, ma userò *problema* (issue) per riferirmi ad un singolo elemento nel database del bug tracker. Questo ci permette di distinguere tra l'aspetto, anche sbagliato, che l'utente ha riscontrato (cioè il bug stesso) e gli *elementi* (record) del tracker della scoperta del bug, della diagnosi e dell'eventuale soluzione. Ricordate che anche sono la maggior parte dei problemi sono veramente bug, possono essere usati per tracciare anche altri tipi di attività.

Il tipico ciclo di vita di un problema è tipo:

1. Qualcuno identifica il problema. Fornisce un'introduzione, una descrizione iniziale (includendo se possibile, una descrizione di come riprodurlo; vedi sezione chiamata «Trattate Ogni Utilizzatore Come un Potenziale Volontario» in Capitolo 8, *Gestire i Volontari* per come incoraggiare delle buone segnalazioni di bug). e qualsiasi altra informazione il tracker richieda. La persona che scopre il bug potrebbe essere completamente sconosciuta al progetto —le segnalazioni di bug e le richieste di funzionalità plausibilmente arrivano dalla comunità degli utenti così come dagli sviluppatori.

   Una volta scoperto, il problema entra nel cosiddetto stato *aperto*. Dato che nessuna azione è ancora stata intrapresa, alcuni tracker lo marcano come *non verificato* (unverified) e/o *non iniziato* (unstarted). Non è assegnato a nessuno; o, in alcuni sistemi è assegnato ad un qualche utente fittizio che rappresenta la mancanza di una vera assegnazione. A questo punto, è in un'area controllata: il problema è stato memorizzato, ma non ancora integrato nella coscienza del progetto.

2. Altri leggono il problema, ci aggiungono commenti, e magari chiedono chiarimenti su alcuni punti allo scopritore.

3. Il bug viene *reprodotto*. Questo potrebbe essere il momento più importante nel suo ciclo di vita. Anche se il bug non è stato riparato, il fatto che qualcuno diverso dallo scopritore è stato in grado di riprodurlo prova che è genuino e, non meno importante, conferma al primo scopritore che ha contribuito al progetto riportando un vero bug.

4. Il bug viene *diagnosticato*: la sua causa viene identificata e se possibile, viene estimato lo sforzo per ripararlo. Siate certi che queste cose vengano registrate nel problema; se la persona che l'ha diagnosticato deve improvvisamente uscire dal progetto per un po' (come spesso accade con sviluppatori volontari), qualcun altro dovrebbe essere in grado di considerare il problema quando se ne va.

   In questo stato, o in qualche momento prima, uno sviluppatore può prendere "possesso" del problema e *assegnare* tale problema a se stesso (sezione chiamata «Distinguere chiaramente fra richiesta e assegnazione» in Capitolo 8, *Gestire i Volontari* esamina il processo di assegnamento in maggior dettaglio). La *priorità* del problema può anche essere impostata a questo punto. Per esempio, se così grave che potrebbe ritardare il prossimo rilascio, questo fatto ha bisogno di essere identificato presto e il tracker deve avere qualche modo di notarlo.

5. Viene pianificata la soluzione del problema. Pianificare non significa necessariamente menzionare una data entro cui sarò risolto. A volte significa solo decidere in quale futuro rilascio (non necessariamente il prossimo) il bug deve essere risolto, o decidere che non ha bisogno di bloccare nessuna release particolare. Ci si può risparmiare la pianificazione, se il bug è veloce da riparare.

6. Il bug viene riparato (o il task completato, o la patch applicata o qualsiasi altra cosa). La modifica o l'insieme di modifiche che l'hanno riparato devono essere registrate in un commento nel problema, dopo di che il problema è *chiuso* (closed) e/o marcato come *risolto* (resolved).

Ci sono alcune comuni variazioni a questo ciclo di vita. A volte un problema è chiuso molto presto dopo essere stato scoperto, perchè viene fuori che non è per niente un bug ma piuttosto un equivoco dalla parte dell'utente. Quando un progetto acquista più utenti, falsi problemi arriveranno sempre in maggior numero, e gli sviluppatori li chiuderanno con tempi di risposta sempre più brevi. Non fa bene a nessuno, dato che il singolo utente in ogni caso non è responsabile per tutti i falsi problemi precedenti; l'andamento statistico è visibile solo dal punto di vista degli sviluppatori, non da quello degli utenti. (In sezione chiamata «Pre-Filtraggio del Bug Tracker» più avanti in questo capitolo, affronteremo le tecniche per ridurre il numero dei falsi problemi.) Inoltre, se utenti diversi sperimentano continuamente lo stesso equivoco , potrebbe voler dire che tale aspetto del software ha bisogno di essere ridisegnato. Questo tipo di tendenza è più facile da notare quando c'è un manager dei problemi che tiene d'occhio il database dei bug; vedi sezione chiamata «Il Manager di Problemi» in Capitolo 8, *Gestire i Volontari*.

Un'altra tipica variazione è la chiusura di un problema perchè *duplicato* subito dopo il primo stato. Un duplicato è quando qualcuno registra un problema che è già noto al progetto. I duplicati non sono confinati nei problemi aperti: è possibile per un bug ritornare dopo essere stato riparato (ciò è noto con il nome di *regressione*), nel qual caso la soluzione preferita è solitamente riaprire il problema originale e chiudere ogni nuova segnalazione come duplicato di quello originale. Il sistema di tracciamento di bug deve tenere traccia di questa relazione in modo bidirezionale, così che l'informazione della rifattorizzazione nei duplicati è disponibile al problema originale e viceversa.

Una terza variazione è per gli sviluppatori di chiudere il problema, pensando di averlo riparato, solo per fare in modo che il segnalatore originale rifiuti la riparazione e lo riapra. Questo accade solitamente perchè gli sviluppatori semplicemente non hanno accesso all'ambiente necessario per riprodurre il bug, o perchè non hanno testato la riparazione usando esattamente la stessa procedura di generazione del bug del segnalatore.

Oltre a queste variazioni, ci possono essere altri piccoli dettagli di ciclo di vita che cambiano a seconda del software di tracking. Ma la forma di base è la stessa, e mentre il ciclo di vita in sè non è specifica al software open source, ha implicazioni su come i progetti open source usano i loro bug tracker.

Come il primo stato implica, il tracker è un aspetto pubblico del progetto tanto quanto le mailing list o le pagine web. Chiunque può registrare un problema, ognuno può guardare un problema, e chiunque può esaminare la lista dei problemi aperti al momento. Ne consegue che non potete mai sapere quanta gente sta aspettando di vedere il progresso di un certo problema. Mentre la grandezza e la capacità della comunità di sviluppo limita la velocità a cui i problemi sono risolti, il progetto deve almeno prendere coscienza del problema nel momento in cui compare. Anche se il problema sta fermo per un po', una risposta incoraggia chi l'ha segnalato a rimanere coinvolto, perchè sente che un qualche umano ha preso nota di quello che ha fatto (ricordate che registrare un problema solitamente comporta maggiore sforzo di, per esempio, mandare una mail). Inoltre, una volta che il problema è visto da uno sviluppatore, entra nella coscienza del progetto, nel senso che lo sviluppatore può essere all'erta per altre istanze del problema, parlarne con altri sviluppatori eccetera.

Il bisogno di pronte risposte implica due cose:

- Il tracker deve essere collegato con una mailing list, in modo che ogni cambiamento di un problema, includendo la sua registrazione iniziale, causa la spedizione di una email che descrive cosa è successo. Questa mailing list è solitamente diversa rispetto a quella normale di sviluppo, dato che non tutti gli sviluppatori potrebbero voler ricevere email automatiche di bug, ma (proprio come con le email di commit) l'header di Reply-to deve essere impostato alla mailing list di sviluppo.

- Il form per la registrazione dei problemi deve poter captare l'indirizzo email di chi fa la segnalazione, così che possa essere contattato per maggiori informazioni. (Comunque, non deve *richiedere* l'indirizzo, dato che alcune persone preferiscono segnalare i problemi in maniera anonima. Vedi sezione chiamata «Anonimità e coinvolgimento» più avanti in questo capitolo per maggiori informazioni sull'importanza dell'anonimato.)

Interazione con le mailing list

Fate in modo che il bug tracker non diventi un forum di discussione. Anche se è importante mantenere una presenza umana nel bug tracker, non è fondamentalmente fatto per le discussioni in tempo reale. Pensatelo piuttosto come un archiviatore, un modo per organizzare fatti e riferimenti ad altre discussioni, principalmente quelli che avvengono sulle mailing list.

Ci sono due motivi per fare questa distinzione. Primo, il bug tracker è più difficoltoso da usare rispetto alla mailing list (o rispetto ai forum di chat in tempo reale, per questo ambito). Questo non è solo perchè i bug tracker hanno un cattivo design dell'interfaccia grafica, è solo che le loro interfacce sono state

disegnate per catturare e presentare stati discreti, non discussioni in libera espansione. Secondo, non tutti quelli che dovrebbero essere coinvolti nella discussione su un particolare problema sta necessariamente guardando il bug tracker. Parte di una buona gestione del problema (vedi sezione chiamata «Suddividete i Compiti di Management e i Compiti Tecnici» in Capitolo 8, *Gestire i Volontari*) è fare in modo che ogni problema sia portato all'attenzione delle giuste persone, piuttosto che richiedere ad ogni sviluppatore di controllare tutti i problemi. In sezione chiamata «Nessuna Conversazione nel Tracciatore di Bug» in Capitolo 6, *Comunicazione*, vedremo i modi per fare in modo che le persone non incappino accidentalmente in discussioni al di fuori degli appropriati forum e nel bug tracker.

Alcuni bug tracker possono controllare le mailing list e fare il log automatico di tutte le email che riguardano un problema noto.Tipicamente lo fanno riconoscendo il numero identificativo del problema nell'oggetto della email, da che è parte di una stringa particolare; gli sviluppatori imparano ad includere queste stringhe nelle loro email per attrarre l'attenzione del bug tracker. Il bug tracker può o salvare la email per intero o (ancora meglio) solo registrare un riferimento alla mail nel normale archivio della mailing list. In entrambi i modi, questa è una funzionalità molto utile; se il vostro tracker ce l'ha, fate in modo sia di attivarla e sia di ricordare alla gente di sfruttarla.

Pre-Filtraggio del Bug Tracker

La maggior parte dei database dei problemi alla fine soffre dello stesso problema: un distruttivo carico di problemi duplicati o non validi registrati da ben intenzionati ma inesperti o malinformati utenti. Il primo passo nel combattere questa tendenza è solitamente mettere un messaggio in evidenza sulla pagina iniziale del bug tracker, spiegando come dire se un bug è davvero tale, come fare ricerche per vedere se è già stato registrato, e infine, come effettivamente riportarlo se si è ancora convinti che sia un nuovo bug.

Questo abbasserà il livello del rumore per un po', ma non appena il numero di utenti crescerà, il problema infine ritornerà. Nessun singolo utente può essere incolpato di questo. Ognuno sta solo provando a contribuire alla buona riuscita del progetto, e anche se la sua prima segnalazione di bug non è di grande aiuto, volete comunque ancora incorraggiare il coinvolgimento e la segnalazione di problemi migliori in futuro. Nel frattempo, comunque, il progetto ha bisogno di tenere il database dei problemi il più possibile libero da spazzatura.

Le due cose che faranno il massimo per prevenire questo problema sono: essere certi che ci sia gente a guardare il bug tracker che abbia abbastanza competenze per chiudere i problemi perchè invalidi o duplicati al momento in cui arrivano, e richiedendo (o incoraggiando fortemente) agli utenti di confrontare i loro bug con altra gente prima di aggiungerli al tracker.

La prima tecnica pare essere universalmente impiegata. Anche progetti con database di problemi enormi (per dire, il bug tracker di Debian a http://bugs.debian.org/, che conteneva 315,929 problemi nel momento di questa scrittura) ancora fa in modo che *qualcuni* veda ogni problema che arriva. Potrebbe essere una persona differente a seconda della categoria del problema. Per esempio, il progetto Debian è una collezione di pacchetti di software, quindi Debian instrada automaticamente ogni problema ai manutentori del pacchetto appropriato. Certo, gli utenti possono a volte identificare male la categoria di un problema, con il risultato che il problema è inizialmente inviato alla persona sbagliata, che deve poi reinstradarla. Comunque, la cosa importante è che tale fardello è ancora condiviso—sia che l'utente sia nel giusto o nel torto quando segnala, il controllo del problema è ancora più o meno equamente distribuito tra gli sviluppatori, così che ogni problema sia in grado di ricevere una pronta risposta.

La seconda tecnica è meno diffusa, probabilmente perchè è più difficile da automatizzare. L'idea fondamentale è che ogni nuovo problema viene "affiancata" nel database. Quando un utente pensa di aver trovato un problema, gli è richiesto di descriverlo su una delle mailing list, o in un canale IRC, e di ottenere una conferma da qualcuno che si tratti davvero di un bug. Coinvolgere subito questo secondo paio di occhi può prevenire molte segnalazioni non genuine. A volte la seconda parte riesce a capire che

non si tratta di un bug o che è stato riparato in un rilascio recente. O può essere familiare con i sintomi di un problema precedente, e può prevenire una duplicazione di segnalazioni indicando all'utente il vecchio problema. A volte è sufficiente solo chiedere all'utente "Hai cercato nel bug tracker per vedere se è già stato segnalato?" Molta gente semplicemente non ci pensa, e comunque sono contenti di fare la ricerca una volta che sanno che qualcuno si *aspetta* che lo facciano.

Il sistema di affiancamento può davvero tenere il database dei problemi pulito, ma ha anche alcuni svantaggi. Molto gente registrerà da sola comunque, o non guardando o infrangendo le istruzioni per trovare un compagno per il nuovo problema. Quindi è comunque necessario per i volontari guardare il database dei problemi. Inoltre, siccome la maggior parte dei nuovi segnalatori non capisce quanto sia difficile il compito di mantenere il database dei problemi, non è giusto rimproverarli troppo duramente per avere ignorato le linee guida. Quindi i volontari devono essere vigili, e comunque praticare influenza su come rimbalzano indietro i problemi non affiancati ai loro segnalatori. Lo scopo è di educare ogni segnalatore ad usare il sistema di affiancamento nel futuro, così che ci sia un insieme sempre in crescita di persone che capiscono il sistema di filtraggio dei problemi. Vedendo un problema non affiancato, i passi ideali sono:

1. Rispondere immediatamente al problema, ringraziando educatamente l'utente per la segnalazione, ma rimandandolo alle linee guida per l'affiancamento (che devono, certamente, essere visibilmente messe sul sito web).

2. Se il problema è chiaramente valido e non un duplicato, approvatelo comunque, e iniziatene il normale ciclo di vita. Dopo tutto, il segnalatore è stato ora informato dell'affiancamento, quindi non c'è motivo nello sprecare il lavoro fatto finora chiudendo un problema valido.

3. Altrimenti, se il problema non è chiaramente valido, chiudetelo, ma chiedete al segnalatore di riaprirlo se riceve conferma da chi lo affianca. Quando ciò accade, deve mettere un riferimento al thread di conferma (ad esempio, la URL nell'archivio della mailing list).

Ricordate che anche se questo sistema migliorerà il rapporto tra segnale e rumore nel database dei problemi nel tempo, non fermerà completamente le segnalazioni non corrette. L'unico modo per prevenirle del tutto è di limitare l'accesso al bug tracker ai soli programmatori—una cura che è quasi sempre peggio della malattia. E' meglio accettare che la pulizia di problemi non validi sarà sempre parte della periodica manutenzione del progetto, cercare di avere il maggior numero possibile di persone che diano una mano.

Vedi anche sezione chiamata «Il Manager di Problemi» in Capitolo 8, *Gestire i Volontari*.

IRC / Sistemi di Chat in tempo reale

Molti progetti offrono chat in tempo reale usando *Internet Relay Chat* (*IRC*), forum dove utenti e sviluppatori possono farsi reciprocamente domande e avere risposte istantanee. Anche se *potreste* fornire un server IRC dal vostro sito web, non è generalemnte il caso. Fate invece ciò che tutti gli altri fanno: fate andare il vostro canale IRC su Freenode (http://freenode.net/). Freenode vi da il controllo di cui avete bisogno per amministrare i canali IRC del vostro progetto, [6] liberandovi dal non insignificante problema di mantenervi da voi un server IRC.

La prima cosa da fare è scegliere un nome per il canale. La scelta più ovvia è il nome del vostro progetto —se disponibile su Freenode, allora usatelo. Altrimenti, cercate di scegliere qualcosa di simile al nome del progetto, e il più facile da ricordare possibile. Pubblicizzate la disponibilità del canale sul sito web del progetto, così che un visitatore con una domanda veloce lo veda subito. Per esempio, questo appare in modo visibile alla cima della pagina principale di Subversion:

[6]Non c'è nessun obbligo nè aspettativa che facciate una donazione a Freenode, ma se voi o il vostro progetto potete permettervelo, per favore considerate questa possibilità. Sono donazioni esentasse USA e loro forniscono un importante servizio.

> *Se state usando Subversion, vi consigliamo di unirvi alla mailing list* `users@subversion.tigris.org` *,e di leggere il libro di Subversion [http://svnbook.red-bean.com/] e FAQ [http://subversion.tigris.org/faq.html]. Potete anche fare domande su IRC a* `irc.freenode.net` *canale* `#svn`.

Alcuni progetti hanno molti canali, uno per sottoargomento. Per esempio, un canale per i problemi di installazione, un altro per le domande sull'uso, un altro per la chat dello sviluppo, eccetera (sezione chiamata «Gestire la Crescita» in Capitolo 6, *Comunicazione* discute su come suddividere i vari canali Quando il vostro progetto è agli inizi, ci dovrebbe essere un solo canale, con tutti quanti a parlare insieme. Poi, quando il rapporto di utenti per sviluppatore cresce, canali separati può diventare necessario.

Come farà la gente a sapere tutti i canali disponibili, a scegliere il canale in cui parlare, come sapranno quali sono le convenzioni locali?

La risposta è dire loro di settare l'*argomento del canale*. [7] L'argomento del canale è un breve messaggio che l'utente vede quando entra per la prima volta nel canale. Forinisce una guida rapida per i nuovi arrivati, e riferimenti ad altre informazioni. Per esempio:

```
Stai parlando su #svn

L'argomento per #svn è Forum per le domande degli utenti di Subversion, vedi anche
http://subversion.tigris.org/. || La discussione sullo sviluppo avviene in
#svn-dev. || Per favore non incollate qui lunghe trascrizioni, usate invece
un sito come http://pastebin.ca/. || NEWS: Subversion 1.1.0
è stata rilasciata, vedi http://svn110.notlong.com/ per dettagli.
```

E' scarno, ma dice ai nuovi arrivati cosa hanno bisogno di sapere. Dice esattamente per cosa esiste il canale (nel caso qualcuno vaghi nel canale senza essere prima stato sul sito del progetto), menziona un canale correlato, e da alcune direttive sulla funzionalità di incolla (paste).

Siti di Paste

Un canale IRC è uno spazion condiviso: tutti possono vedere ciò che tutti gli altri stanno dicendo. Normalmente, questa è una buona cosa, dato che permette alle persone di entrare in una conversazione quando pensano di poter contribuire, permettendo alla gente di imparare guardando. Ma diventa problematico quando qualcuno deve fornire una grossa quantità di informazione in una volta, per esempio la trascrizione di una sessione di debug, perchè incollare troppe righe di output nel canale interromperebbe le altre conversazioni.

La soluzione è usare un sito di Paste, detto anche *pastebin* o *pastebot*. Quando viene richiesto da qualcuno una grande mole di dati, chiedetegli di non fare il paste nel canale, ma di andare invece (per esempio) http://pastebin.ca/, incollare i loro dati nel form, e comunicare la nuova URL risultante al canale IRC. Chiunque può quindi visitare la URL e vedere i dati.

Ci sono molti siti di paste gratuiti, troppi per farne una lista comprensiva, ma qui ci sono alcuni di quelli che ho usato: http://www.nomorepasting.com/, http://pastebin.ca/, http://nopaste.php.cd/ http://rafb.net/paste/ http://sourcepost.sytes.net/, http://extraball.sunsite.dk/notepad.php, e http://www.pastebin.com/.

[7] Per settare l'argomento di un canale, usate il comando `/topic` (argomento). Tutti i comandi su IRC iniziano con "`/`". Vedi http://www.irchelp.org/ se non siete famigliari con l'uso e l'amministrazione di IRC; in particolare, http://www.irchelp.org/irchelp/irctutorial.html è un ottimo tutorial.

Bot

Molti canali IRC tecnici hanno un membro non umano cosiddetto *bot*, che è capace di archiviare e rigurgitare informazioni in risposta a specifici comandi. Tipicamente, il bot è interpellato come ogni altro membro del canale, cioè i comandi sono inviati "parlando" al bot. Per esempio:

```
<kfogel> ayita: learn diff-cmd = http://subversion.tigris.org/faq.html#diff-cmd
<ayita>  Grazie!
```

Questo dice al bot (che è loggato nel canale come ayita) di ricordare una certa URL come risposta alla domanda "diff-cmd". Ora possiamo interpellare ayita, chiedendogli di dire ad un altro utente di diff-cmd:

```
<kfogel> ayita: tell jrandom about diff-cmd
<ayita>  jrandom: http://subversion.tigris.org/faq.html#diff-cmd
```

La stessa cosa può essere fatta usando una conveniente scorciatoia:

```
<kfogel> !a jrandom diff-cmd
<ayita>  jrandom: http://subversion.tigris.org/faq.html#diff-cmd
```

L'insieme di comandi e comportamenti può cambiare da bot a bot. L'esempio sopra è con `ayita` (http://hix.nu/svn-public/alexis/trunk/), di cui un'istanza di solito è presente in `#svn` su Freenode. Altri bot sono Dancer (http://dancer.sourceforge.net/) e Supybot (http://supybot.com/). Da notare che non c'è bisogno di particolari privilegi sul server per far andare un bot. Un bot è un programma client; tutti possono impostarne uno e metterlo in ascolto su un particolare server o canale.

Se il vostro canale tende ad ospitare domande ripetute molte volte, raccomando vivamente di impostare un bot. Solo una piccola percentuale di utenti del canale avrà la capacità di manipolare il bot, ma questi utenti risponderanno ad una percentuale sproporzionatamente alta di domande, perchè il bot permette loro di rispondere in modo più efficiente.

Archiviazione di IRC

Anche se è possibile archiviare tutto ciò che succede in un canale IRC, non è necessariamente atteso. Le conversazioni IRC possono essere nominalmente pubbliche, ma molte persone le considerano informali, semi-private. Gli utenti possono essere incuranti della grammatica, e a volte esprimere opinioni (per esempio, su altro software o altri programmatori) che non vorrebbero salvati in un archivio online.

Certo, ci saranno a volte *estratti* che dovrebbero essere preservati, e va bene. La maggior parte dei client IRC può fare il log di una conversazione su file su richiesta dell'utente, o non potendo, si può sempre fare semplice copia e incolla della conversazione da IRC in un forum più permanente (molto spesso il bug tracker). Ma tenere traccia indiscriminata di tutto può mettere a disagio alcuni utenti. Se archiviate tutto, fate in modo di dirlo chiaramente nell'argomento del canale, e fornite la URL dell'archivio.

Wiki

Una *wiki* è un sito web che permette ad ogni visitatore di cambiare o estendere il suo contenuto; il termine "wiki" (da una parola hawaiana che significa "pronto" o "veloce") è anche usato per riferirsi al software che permette tali modifiche. Le wiki sono state inventate nel 1995, ma la loro popolarità è davvero iniziata a decollare dal 2000 o 2001, spinta in parte dal successo di Wikipedia (http://

www.wikipedia.org/), una enciclopedia basata su wiki dal contenuto gratutito. Pensate ad una wiki come qualcosa tra IRC e le pagine web: le wiki non sono in tempo reale, così le persone hanno la possibilità di pensare e rifinire le loro contribuzioni, ma sono anche molto facili da ampliare, coinvolgendo meno sforzo di interfaccia che il cambiamento di una pagina web regolare.

Le wiki non sono ancora un equipaggiamento standard per i progetti open source, ma probabilmente lo saranno presto. Dato che sono una tecnologia relativamente nuova, e la gente sta ancora sperimentando i diversi modi per usarle, qui userò poche parole di cautela—a questo punto, è più facile analizzare le debolezze delle wiki che analizzarne i successi.

Se decidete di avere una wiki, mettete molto impegno nell'avere una organizzazione di pagina pulita e di presentazione piacevole, così che i visitatori (cioè i potenziali editori) sapranno istintivamente come fare le loro contribuzioni. Ugualmente importante, pubblicate questi standard sulla wiki stessa, così la gente ha un posto per consultare come guida. Troppo spesso, gli amministratori della wiki sono vittime della fantasia che dal momento che orde di visitatori stanno individualmente aggiungendo contenuto di alta qualità al sito, la somma di tutte queste contribuzioni deve quindi anche essere di alta qualità. Non è così che un sito web funziona. Ogni singola pagina o paragrafo deve essere buono quando considerato in sè, ma non sarà buono se incluso in un insieme disordinato o confuso. Troppo spesso, le wiki soffrono di:

- **Mancanza di principi di navigazione.** Un sito web ben organizzato fa sentire i visitatori come se sapessere dove sono in qualunque momento. Per esempio, se le pagine sono ben disegnate, le persone possono intuitivamente dire la differenza tra una parte di introduzione da una di contenuto. Anche i contributori di una wiki rispetteranno tale differenze, ma solo se le differenze sono presenti per cominciarci.

- **Duplicazione dell'informazione.** Le wiki spesso arrivano ad avere pagine diverse che dicono cose simili, perchè i contributori singoli non hanno notato le duplicazioni. Questo può in parte essere conseguenza della mancanza di principi di navigazione indicata sopra, nel caso le persone non trovino il contenuto duplicato se non è dove si aspettano che sia.

- **Gruppo non consistente di destinatari** In qualche modo questo problema è inevitabile quando ci sono così tanti autori, ma può essere limitato se ci sono linee guida scritte su come creare nuovi contenuti. Aiuta all'inizio anche modificare in modo aggressivo le contribuzioni, come un esemio, così da iniziare a restringere gli standard.

La soluzione comune a tutti questi problemi è lo stesso, avere standard editoriali, e dimostrarli non solo pubblicandoli, ma modificando le pagine per renderle conformi. In generale, le wiki amplificheranno ogni errore nel loro materiale originale, dato che i contributori imitano qualunque esempio vedono davanti a loro. Non impostate soltanto la wiki sperando che tutto vada a posto. Dovete anche iniziarla con contenuti ben scritti, così che le persone abbiano esempi da seguire.

L'esempio splendente di una run ben gestita è Wikipedia, anche se questo è in parte perchè il contenuto (voci di enciclopedia) è naturalmente adatto per il formato wiki. Ma se esaminate Wikipedia da vicino, vedrete che i suoi amministratori hanno costruito una fondazione per la cooperazione *molto* efficente. C'è una vasta documentazione su come scrivere nuove voci, come conservare un adeguato punto di vista, che tipo di modifiche fare, quali evitare, un processo di risoluzione per le dispute su voci contestate (che comprende molti stati, includendo un giudizio finale) e così via. Hanno anche controlli di autorizzazione, così che se una pagina è bersaglio di ripetute modifiche non appropriate, possono bloccarla fino a quando il problema è risolto. In altre parole, non hanno solo buttato qualche template su un sito web e sperato in bene. Wikipedia funziona perchè i suoi fondatori hanno pensato accuratamente su come migliaia di estranei adeguino la loro scrittura ad una visione comune. Anche se potreste non avere bisogno dello stesso livello di preparazione per far andare un wiki per un progetto di software libero, vale la pena emularne lo spirito.

Per maggiori informazioni sulle wiki, vedi http://en.wikipedia.org/wiki/Wiki. Inoltre, la prima wiki rimane viva e vegeta, e contiene molte discussioni su come far andare le wiki: vedi http://www.c2.com/

cgi/wiki?WelcomeVisitors, http://www.c2.com/cgi/wiki?WhyWikiWorks, e http://www.c2.com/cgi/wiki?WhyWikiWorksNot per vari punti di vista.

Web Site

Non c'è molto da dire sulla costituzione del sito web del progetto da un punto di vista tecnico: tirare su un server web e scrivere pagine web sono attività abbastanza semplici, e la maggior parte delle cose importanti da dire sulla presentazione e organizzazione sono stati trattati nel precedente capitolo. La funzione principale del un sito web è presentare un'introduzione al progetto chiara e invitante, e di integrare gli altri strumenti (il sistema di controllo di versione, il bug tracker eccetera). Se non avete le conoscenze per tirare su il web server, di solito non è difficile trovare qualcuno che le ha e che vuole darvi una mano. Ciononostante, per risparmiare tempo e sforzo, la gente spesso preferisce usare uno dei siti di canned hosting.

Canned Hosting

Ci sono due grandi vantaggi nell'uso di siti preconfezionati (canned site). Il primo è la capacitò dei server e la larghezza di banda: i loro server sono grosse scatole sedute su tubi davvero capienti. Non importa quanto il vostr progetto diventerà di successo, non farete finire lo spazio su disco nè sovraccaricare la connessione di rete. Il secondo vantaggio è la semplicità. Hanno già scelto un bug tracker, un sistema di controllo di versione, un gestore di mailing list, un archiviatore, e ogni altra cosa di cui avete bisogno per fare andare avanti un sito. Hanno configurato gli strumenti, e si prendono cura dei backup per tutti i dati memorizzati in tali strumenti. Non avete bisogno di prendere alcuna decisione. Tutto quello che dovete fare è riempire un form, schiacciare un bottone e improvvisamente avete un sito web del progetto.

Questi sono vantaggi abbastanza significativi. Gli svantaggi, certo, sono il fatto di dover accettare le *loro* scelte e configurazioni, anche se qualcosa di diverso sarebbe stato meglio per il vostro progetto. Solitamente i siti preconfezionati sono personalizzabili in certi secondari parametri, ma non avrete mai il controllo dettagliato che avreste se creaste il sito da voi e aveste un pieno controllo amministrativo sul server.

Un perfetto esempio di questo è la gestione dei file generati. Alcune pagine web del progetto potrebbero essere file generati—per esempio, ci sono sistemi per mantenere i dati delle FAQ in un formato facile da modificare, da cui HTML,PDF e altri formati di presentazione possono essere generati. Come spiegato in sezione chiamata «Tenere tutto sotto controllo di versione» prima in questo capitolo, non vorreste mettere sotto controllo di versione i formati generati, solo il file di origine. Ma quando il vostro sito web è ospitato sul server di qualcun altro, potrebbe essere impossibile generare un hook personalizzato per rigenerare la versione HTML onlinea delle FAQ in qualunque momento il file di origine sia modificato. L'unico modo di aggirare questo è di mettere sotto controllo di versione anche i formati generati, così come appaiono sul sito web.

Ci possono anche essere conseguenze più importanti. Potreste non avere abbastanza controllo sulla presentazione quanto vorreste. Alcuni siti di canned hosting vi permettono di personalizzare le vostre pagine web, ma il layout di default del sito solitamente finisce per uscir fuori nei modi peggiori. Per esempio, alcuni progetti che ospitano se stessi su SourceForge hanno completamente personalizzato le pagine principali, ma rimandano ancora gli sviluppatori alle loro pagine SourceForge per maggiori informazioni. La pagina SourceForge è cosa sarebbe la pagina principale del progetto, se questo non avesse usato la pagina personalizzata. La pagina SourceForge ha link al bug tracker, al repository CVS, download eccetera. Sfortunatamente, la pagina ourceForge contiene anche una grande quantità di rumore estraneo. La cima è un banner di pubblicità, spesso un'immagine animata. Il lato sinistro è un insieme verticale di link di scarsa rilevanza per qualcuno interessato al progetto. Il lato destro è spesso un'altra pubblicità. Solo il centro della pagina è dedicato al vero materiale specifico del progetto, e anche

questo è organizzato in un modo confusionario che spesso rende i visitatori insicuri su cosa cliccare dopo.

Dietro ogni aspetto del design di SourceForge, c'è senza dubbio una buona ragione—buona dal punto di vista di SourceForge, come le pubblicità. Ma dal punto di vista di un progetto individuale, il risultato può essere una pagina web meno che ideale. Non intendo bacchettare SourceForge; gli stessi problemi si trovano in molti dei siti di canned hosting. Il punto è che c'è un compromesso. Voi guadagnate il sollievo dal fardello tecnico di far andare il sito del progetto, ma solo al prezzo di accettare il modo di farlo di qualcun altro.

Solo voi potete decidere se il canned hosting è la scelta migliore per il vostro progetto. Se scegliete un sito preconfezionato, lasciate aperta l'opzione di trasferirvi su un vostro server più avanti, usando un nome di dominio personale per l'"indirizzo di casa". Potete fare il forward della URL al sito preconfezionato, o avere un home page completamente personalizzata alla URL pubblica e mandare gli utenti al sito preconfezionato per le funzionalità sofisticato. Fate solo in modo di arrangiare le cose in modo che se più avanti decidiate di usare una diversa soluzione di hosting, l'indirizzo del progetto non abbia bisogno di cambiare.

Scegliere un sito di canned hosting

Il più grande e conosciuto sito di hosting èSourceForge [http://www.sourceforge.net/]. Altri due siti che forniscono un servizio uguale o simile sono savannah.gnu.org [http://savannah.gnu.org/] e BerliOS.de [http://www.berlios.de/]. Alcune organizzazioni come la Apache Software Foundation [http://www.apache.org/] e Tigris.org [http://www.tigris.org/][8], forniscono hosting gratuito a progetti open source che ben si adattano al loro ambito e ai progetti esistenti della loro comunità.

Haggen So ha fatto una esauriente valutazione di vari siti di canned hosting, come parte della ricerca per la sua tesi di dottorato, *Construction of an Evaluation Model for Free/Open Source Project Hosting (FOSPHost) sites*(Costruzione di un modello di valutazione per l'hosting di progetti free o open source). I risultati sono in http://www.ibiblio.org/fosphost/, e in particolare vedete il leggibilissimo grafico di paragone a http://www.ibiblio.org/fosphost/exhost.htm.

Anonimità e coinvolgimento

Un problema che non è strettamente limitato ai siti preconfezionati, ma che vi si trova più di frequente, è l'abuso della funzionalità del login utente. La funzionalità in sè è abbastanza semplice: il sito permette ad ogni visitatore di registrarsi con username e password. Da lì in poi mantiene un profilo per tale utente, gli amministratori del progetto possono assegnare all'utente certi permessi, per esempio, il diritto di fare commit sul repository.

Questo può essere estremamente utile, e infatti è one dei primi vantaggi del canned hosting. Il problema è che a volte il login dell'utente finisce per essere richiesto per compiti che dovrebbero essere permessse ai visitatori non registrati, in particolare la possibilità di registrare problemi nel bug tracker, e di commentare problemi esistenti. Richiedendo uno username valido per tali azioni, il progetto alza l'indicatore di coinvolgimento per cosa dovrebbero essere attività veloci e facili. Di sicuro, si vorrebbe essere in grado di contattare qualcuno che ha immesso dei dati nel bug tracker, ma avere un campo dove (volendo) si può inserire l'indirizzo email, è sufficiente. Se un nuovo utente trova un bug e vuole riportarlo, sarà soloannoiato dal dovere completare una creazione di account prima che possa immettere il bug nel tracker. Potrebbe semplicemente decidere di non segnalare del tutto il bug.

I vantaggi della gestione degli utente solitamente sovrastano gli svantaggi. Ma se potete scegliere quali azioni possono essere fatte in modo anonimo, siate certo non solo che *tutte* le azioni di sola lettura siano

[8]Disclaimer: sono dipendente di CollabNet [http://www.collab.net/], che sostiene Tigris.org, e uso Tigris regolarmente.

permesse a visitatori non loggati, ma anche alcune azioni di immissione dati, in particolare nel bug tracker e, se le avete, nelle pagine wiki.

Capitolo 4. L'Infrastruttura Sociale e Politica

Le prime domande che la gente fa sul software libero sono: Come funziona? Cosa mantiene il progetto in funzione? Chi prende le decisioni? Io sono sempre insoddisfatto dalle risposte blande sulla meritocrazia, sullo spirito di collaborazione, sul fatto che il codice parla da sé, ecc.. Il fatto è che, alla domanda non è facile rispondere. La meritocrazia, e il codice funzionante sono parte di esso, ma essi fanno poco nello spiegare come gira il progetto sulla base del giorno per giorno, e non dicono nulla su come i conflitti vengono risolti.

Questo capitolo cerca di mostrare i puntellamenti che progetti di successo hanno in comune. Io intendo #di successo# non solo in termini di qualità tecniche, ma anche di salute operativa e di capacità di sopravvivere. La salute operativa è la capacità di incorporare strada facendo nuovi contributi di codice e di sviluppatori, e di essere reattivi ai rapporti di bugs che arrivano. La capacità di sopravvivere in un progetto è la capacità di esistere indipendentemente da un partecipante individuale o sponsor—tpensate ad essa come alla probabilità che il progetto continuerebbe anche se tutti i suoi membri fondatori si spostassero su un'altra cosa. Il successo tecnico non è difficile da conseguire, ma senza una robusta base di sviluppo e un fondamento sociale, un progetto può essere incapace di gestire la crescita che il successo iniziale porta, alla partenza di individualità carismatiche.

Ci sono molti modi per raggiungere questo tipo di successo. Alcuni riguardano una struttura di amministrazione formale, con la quale le discussioni sono risolte, nuovi sviluppatori sono invitati (o talvolta estromessi), nuove funzionalità sono pianificate, e così via. Altri riguardano strutture meno formali, ma un più conscio auto contenimento, per produrre un'atmosfera essere piacevole, cosa su cui la gente può contare come una forma di amministrazione *de facto*. Tutte e due le vie portano allo stesso risultato: un senso di stabilità istituzionale, supportata da comportamenti e procedure che possono essere ben compresi da chiunque partecipi. Queste caratteristiche sono anche più importanti in sistemi di auto organizzazione, che in sistemi controllati dal centro, perché in sistemi di auto organizzazione ognuno è conscio che poche mele marce possono rovinare l'intero canestro, almeno per un certo tempo.

La Possibilità di Diramazione

L'ingrediente indispensabile che tiene legati gli sviluppatori in un progetto di software libero, e li rende desiderosi di arrivare a un compromesso quando necessario, è*la possibilità di una diramazione* del codice: l'abilità di ciascuno di prendere una copia del codice sorgente e partire con un progetto concorrente, cosa nota come *forchetta*. La cosa paradossale è che la *eventualità* di forchette nei progetti di software libero è usualmente una forza molto più grande che le forchette reali, che sono molto rare. Poichè una forchetta è un male per chiunque (per le ragioni esaminate in dettaglio in sezione chiamata «Le Diramazioni» in Capitolo 8, *Gestire i Volontari*), più seria diventa la minaccia di una forchetta, più le persone sono desiderose di un compromesso per evitarla.

Le forchette, o piuttosto il potenziale delle forchette, sono la ragione per cui non vi sono veri dittatori nei progetti di software libero. Questo può sembrare una affermazione sorprendente, considerato quanto è comune ascoltare qualcuno chiamato #dittatore# o #tiranno# in un dato progetto open source. Ma questo tipo di tirannia è speciale, completamente differente dall'intendere convenzionale della parola. Immaginate un re i cui sudditi potessero copiare il suo intero regnare in ogni momento e muoversi a copiarne il ruolo nella misura in cui lo vedano giusto. Non governerebbe un tale re molto diversamente da uno i cui sudditi fossero soggetti a sottostare al suo dominio, qualunque cosa facesse?

Questo è il perché anche progetti non formalmente organizzati come democrazie, in pratica, sono democrazie quando ci si trova davanti a importanti decisioni. La replicabilità suggerisce possibilità di diramazione; la possibilità di diramazione suggerisce consenso. Può ben essere che ognuno voglia

far riferimento a un unico leader (l'esempio più famoso è Linus Torvalds nello sviluppo del kernel di Linux), ma ciò avviene perchè essi *scelgono* di fare così in un modo completamente non cinico e sinistro. Il dittatore non ha un magico potere sul progetto. Una proprietà chiave di tutte le licenze open source è che esse non danno a una parte un maggior potere che a qualche altra di decidere come il codice possa essere cambiato o usato. Se il dittatore stesse improvvisamente incominciando a prendere cattive decisioni, ci sarebbe un'agitazione, seguita probabilmente da una rivolta e da una scissione. A meno che, indubbiamente, le cose raramente vanno così lontano, il dittatore venga prima a un compromesso.

Ma appunto perché la possibilità di diramazione mette un limite superiore al potere che uno può esercitare su un progetto non significa che non ci siano importanti differenze su come il progetto viene condotto. Voi non avete bisogno del fatto che ogni decisione venga presa alla richiesta di ultima spiaggia di chi sta prendendo in considerazione una forchetta. Cosa che seccherebbe molto rapidamente e toglierebbe energia al lavoro reale. Le prossime due sezioni esaminano differenti modi di organizzare un progetto in modo tale che la maggior parte delle decisioni vadano per il verso giusto. Questi due esempi sono in qualche modo idealizzati come casi limite. Molti progetti cadono con una certa continuità fra di essi.

I Dittatori Benevoli

Il modello di *dittatore benevolo* è esattamente ciò che suona come: l'autorità delle decisioni finali di pende da un'unica persona, che, in virtù della personalità e dell'esperienza, si prevede che la usi saggiamente.

Sebbene #dittatore benevolo# (o *BD*) sia il termine standard per questo ruolo, è bene pensare ad esso come #arbitro approvato dalla comunità# o #giudice#. Generalmente i dittatori benevoli, in realtà non prendono tutte le decisioni, e nemmeno la maggioranza delle decisioni. Non è verosimile che un'unica persona potrebbe avere la necessaria esperienza per prendere costantemente buone decisioni lungo tutta l'area del progetto, e comunque, gli sviluppatori di qualità non starebbero intorno se non avessero qualche influenza sull'orientamento del progetto. Quindi i dittatori benevoli non dettano molto. Invece essi lasciano che le cose vadano avanti da sole attraverso discussioni ed esperimenti ogni volta che sia possibile. Essi partecipano a tutte le discussioni di persona, ma coma regolari sviluppatori, ma facendo riferimento a un reggente di area che ha più esperienza. Solo quando è chiaro che non può essere raggiunto il consenso, e che la maggior parte del gruppo *vuole* che qualcuno guidi le decisione in modo che lo sviluppo vada avanti, puntano i pedi e dicono #questo è il modo in cui deve andare#. La riluttanza a prendere decisioni per decreto è una prerogativa condivisa virtualmente da tutti i dittatori benevoli di successo; è una delle ragioni per cui essi riescono a mantenere il ruolo.

Chi Può Essere un Dittatore Benevolo?

Essere un BD richiede una combinazione di caratteristiche. C'è bisogno, prima di tutto, di una sensibilità ben affilata per quanto riguarda la propria influenza nel progetto, che di volta in volta porta a un auto controllo. Nei primi stadi di una discussione un singolo non esprimerebbe opinioni e conclusioni con così tanta certezza che altri potrebbero percepire come inutile il dissentire. La gente deve esprimere pubblicamente e liberamente le idee, anche sciocche idee. E' inevitabile che il BD anche esprimerà una idea sciocca di volta in volta, certamente, e quindi il ruolo richiede anche una abilità a rendersi conto e a riconoscere quando uno ha preso una cattiva decisione sebbene questo sia un caratteristica che *ogni* buon sviluppatore dovrebbe avere, specialmente se rimane col progetto per un lungo tempo. Ma la differenza è che i BD può permettersi un lapsus di volta in volta senza preoccuparsi per sua credibilità a lungo termine. Gli sviluppatori con minore anzianità possono non sentirsi così sicuri, così il BD dovrebbe esprimere critiche o decisione contrarie con una certa sensibilità per quanto riguarda il peso che hanno le sue parole, sia tecnicamente che psicologicamente.

Il BD *non* deve aver bisogno di avere la più evidente esperienza di tutti nel progetto. Egli deve aver sufficiente esperienza sul suo codice, e capire e commentare ogni cambiamento in considerazione, ma

questo è tutto. La posizione di BD è né acquisita né mantenuta per virtù di una abilità nello scrivere codice che intimidisce. Quello che *è* importante è l'esperienza e il senso dell'insieme nella progettazione non necessariamente la capacità di produrre una buona progettazione su richiesta, ma la capacitò di riconoscere la buona progettazione, qualunque sia la sua origine.

E' comune che il dittatore benevolo sia il fondatore del progetto. Ma questa è più una correlazione che una causa. Il tipo di qualità che rende uno capace di avviare un progetto con successo—competenza tecnica, capacità di persuadere altri ad unirsi ad esso, ecc...—sono esattamente le qualità di cui ogni BD avrebbe bisogno. E, certamente, i fondatori incominciano con una sorta di anzianità automatica, che può spesso essere sufficiente a far si che la dittatura benevola appaia il percorso di minor difficoltà per tutti gli interessati.

Ricordate che la possibilità per la forchetta esiste in entrambi i casi. Un BD può fare una diramazione dal progetto appunto facilmente come ciascun altro, e alcuni hanno occasionalmente fatto così, quando hanno visto che la direzione che essi volevano che prendesse il progetto era diversa da quella che gli altri sviluppatori volevano. A causa della possibilità di diramazione, non ha importanza se il dittatore benevolo ha la radice (i privilegi di amministratore del sistema) sui principali servers del progetto. La gente a volte parla del controllo del server come se essa fosse la fonte principale del potere in un progetto, ma nei fatti ciò è irrilevante. Il fatto di aggiungere o rimuovere le password di invio su un particolare server riguarda solo la copia del progetto che è sul server. Un prolungato abuso di questo potere, da parte del BD o di qualche altro, spingerebbe solamente lo sviluppo a spostarsi su un altro server.

Se il progetto dovrebbe avere un dittatore benevolo o se andrebbe meglio con qualche sistema meno centralizzato, dipende largamente da chi è disponibile a ricoprire il ruolo. Come regola generale, se è semplicemente ovvio per ognuno chi dovrebbe essere il BD, allora quella è la strada da prendere. Ma se non c'è un candidato per il BD immediatamente scontato, allora il progetto dovrebbe usare un processo di di presa delle decisioni decentralizzato, come descritto nella prossima sezione.

Democrazia Basata sul Consenso

Quando un progetto diventa più vecchio, tende a spostarsi altrove rispetto alla benevola dittatura. Questa non è necessariamente insoddisfazione nei riguardi di un particolare BD. E' semplicemente che il comando basato sul gruppo è #più stabile dal punto di vista dell'evoluzione#, per prendere in prestito una metafora dalla biologia. Ogni volta che un dittatore benevolo si ritira, un sistema non dittatoriale—stabilisce una costituzione, per così dire. TIl gruppo può non cogliere questa occasione la prima volta, o la seconda, ma alla fine lo farà; una volta fatto, la decisione è improbabile che sia mai revocata. Il senso comune spiega il perché: se un gruppo di N persone fosse per l'investire una persona di un potere speciale, ciò significherebbe che N - 1 persone stessero accettando una diminuzione della loro personale influenza. Le persone usualmente non vogliono fare ciò. Anche se lo facessero, la dittatura risultante sarebbe ancora condizionata. Il gruppo consacrò il BD, chiaramente il gruppo potrebbe deporre il BD. Perciò, una volta che il progetto ha abbandonato la leadership da parte di un individuo carismatico per un più formale sistema basato sul gruppo, è raro che torni indietro.

I dettagli su come questo sistema funziona variano largamente, ma ci sono due elementi comuni: uno, il gruppo lavora per consenso la maggior parte del tempo; due: c'è un formale meccanismo di voto con cui aiutarsi quando il consenso non può essere raggiunto.

Consenso ha solamente il significato di un accordo con il quale ognuno vuol vivere. Non è uno stato ambiguo: un gruppo ha raggiunto il consenso quando qualcuno propone che quel consenso è stato raggiunto, e nessuno contraddice questa affermazione. La persona che propone il consenso dovrebbe, certamente, stabilire di quale consenso di tratta, e quali azioni possono essere intraprese in conseguenza di esso, se non sono scontate.

La maggior parte delle conversazioni nel progetto sono su argomenti tecnici, come il giusto modo per correggere un bug, se o non aggiungere una funzionalità, quanto particolareggiatamente documentare le interfacce, ecc.. La conduzione basata sul consenso funziona bene perché si armonizza senza problemi con la discussione tecnica stessa. Alla fine di una discussione c'è generalesse l'accordo su che via intraprendere. Qualcuno può usualmente fare un post che è allo stesso tempo un sommario su quello che è stato deciso e una implicita proposta di consenso. Ciò fornisce anche ad ognuno un'ultima occasione per dire: #Aspettate, io non sono d'accordo su questo. Dobbiamo sviscerare ciò ancora per un po'#.

Per piccole, non controverse decisioni, la proposta di consenso è implicita. Per esempio, quando uno sviluppatore invia spontaneamente un correzione di bug, l'invio stesso è una proposta di consenso: #Io prendo per buono che tutti concordino sul fatto che questo bug debba essere corretto, e che questo sia il modo di correggerlo#. Ovviamente lo sviluppatore non dice in realtà ciò; egli semplicemente invia la correzione, e gli altri nel progetto, non si infastidiscono a dare il loro ok, perché il silenzio è consenso. Se qualcuno invia un cambiamento che non riscuote il consenso, il risultato è semplicemente che il progetto discute il cambiamento come se non fosse stato ancora inviato. Il motivo per cui ciò funziona nell'argomento della prossima sezione.

Controllo di Versione Significa Che Vi Potete Rilassare

Il fatto che il codice sorgente è tenuto sotto il controllo di versione significa che si può tornare indietro dalla maggior parte delle decisioni. Il più comune modo in cui ciò avviene è che ognuno invia un cambiamento pensando erroneamente che ognuno sarebbe contento di esso, solo che si trova di fronte a obiezioni dopo il fatto. E' tipico che tali obiezioni incomincino con un obbligatorio encomio per aver evitato una precedente discussione, anche se questo si può evitare se chi fa l'obiezione non ricorda di una tale discussione negli archivi della mailing list. In un modo o nell'altro, non c'è ragione per la quale il tono della discussione debba essere differente se l'invio del cambiamento è avvenuto prima o dopo. Ogni modifica può essere annullata, almeno fino a quando modifiche dipendenti da essa siano introdotte (cioè la nuova codifica così verrebbe fermata se la modifica originaria fosse subito rimossa). Il sistema del controllo di versione dà al progetto il modo di annullare gli effetti di giudizi cattivi o frettolosi. Questo, a sua volta, libera la gente dall'affidarsi al proprio istinto su quante conferme siano necessarie prima di fare qualcosa.

Ciò significa anche che il processo dello stabilire il consenso non ha bisogno di essere molto formale. La maggior parte dei progetti con esso si comportano a sensazione. Le modifiche minori possono andare avanti confidenzialmente senza discussione, o con un minimo di discussione seguito da pochi cenni di accordo. Per le modifiche più significative, specialmente quelle che possono destabilizzare una gran quantità di codice, le persone dovrebbero aspettare un giorno o due prima di dare per acquisito che c'è il consenso, essendo razionale che nessuno dovrebbe essere marginalizzato in una importante conversazione semplicemente perché non ha controllato la posta abbastanza frequentemente.

Così, quando qualcuno confida di sapere quello che deve essere fatto, dovrebbe procedere e farlo. Questo non si applica solo alla correzione del software, ma all'aggiornamento del sito, alle modifiche della documentazione, e a ogni altra cosa, a meno che la cosa non sia controversa. Usualmente ci saranno solo poche occasioni in cui una azione avrà bisogno di essere annullata, e queste saranno trattate sulla base del caso per caso. Certamente, uno non dovrebbe incoraggiare la gente ad essere testarda. C'è ancora una differenza psicologica fra una decisione sotto discussione e una che ha già avuto effetto, anche se questa è tecnicamente reversibile. La gente pensa sempre che la velocità è alleata dell'azione e sarà leggermente più riluttante ad annullare un cambiamento che a prevenirlo innanzitutto. Se uno sviluppatore abusa di questo fatto inviando modifiche potenzialmente controverse troppo velocemente, comunque, la gente può e dovrebbe protestare e costringere quello sviluppatore a uno standard più stringente finché le cose migliorino.

Quando Il Consenso Non Può Essere Raggiunto, Votate

Inevitabilmente alcuni dibattiti potranno arrivare al consenso. Quando tutte le altre vie per venir fuori da un punto morto falliscono, la soluzione è votare. Ma prima che sia raccolto il voto ci deve essere una chiara serie di scelte sulla scheda. Qui, di nuovo, il normale procedimento della discussione tecniche si unisce in modo benefico col procedimento del progetto di presa della decisioni. Il tipo di questione che vengono al voto spesso comprendono complessi e sfaccettati problemi. In ogni discussione così complessa ci sono usualmente una o due persone che occupano il ruolo di *onesti mediatori*: che postano sommari periodici dei vari argomenti e che tengono traccia di dove sono i punti di disaccordo (e di accordo). Questi sommari aiutano ciascuno a misurare quanto progresso è stato fatto e quanti problemi rimangono da essere elencati. Questi medesimi sommari possono servire come prototipo di scheda elettorale, dovrebbe diventare necessario un voto. Se gli onesti mediatori hanno fatto bene il loro lavoro, saranno capaci di chiamare con credibilità per il voto quando viene il momento, e il gruppo sarà contento di usare una scheda elettorale basata sui loro sommari di problemi. I mediatori stessi potranno esser partecipi al dibattito; non è necessario che essi rimangano superiori alla lite, fino a quando essi possono capire e rappresentare in modo imparziale i punti di vista, e non permettere che i propri sentimenti di parte gli impediscano di riassumere lo stato del dibattito in una maniera neutrale.

Il reale contenuto della scheda non è oggetto di controversie. Col tempo le materie vengono al voto, i disaccordi vengono condensati in pochi problemi chiave, con etichette riconoscibili e brevi descrizioni. Occasionalmente lo sviluppatore farà obiezione alla forma della scheda stessa. A volte il suo interessamento è legittimo, per esempio, che una scelta importante è stata lasciata fuori o non è stata descritta accuratamente. Ma altre volte uno sviluppatore può star cercando si evitare l'inevitabile, forse sapendo che il voto non andrà come lui vuole. Vedere sezione chiamata «Gente Difficile» in Capitolo 6, *Comunicazione* su come affrontare questo tipo do ostruzionismo.

Ricordatevi di specificare il sistema di voto, se ci sono molti differenti modi, e se la gente potrebbe potrebbe fare delle cattive supposizioni sulla procedura che viene usata. Una buona scelta nella maggior parte dei casi è il *voto per approvazione*, dove ogni votante può votare per quante scelte vuole nella scheda. Il voto per approvazione è semplice da spiegare a da contare, e diversamente da altri metodi, richiede un solo giro di voto. Vedere http://en.wikipedia.org/wiki/Voting_system#List_of_systems per maggiori dettagli sul voto di approvazione ed altri sistemi di voto, ma cercate si evitare di entrare in una lunga discussione su quale sistema usare (perché, certamente, allora voi stessi vi troverete in una discussione su quale sistema di voto usare per decidere quale sistema di voto usare). Una ragione per cui il voto di approvazione è una buona scelta è che è molto difficile per ognuno farvi obiezione;—è all'incirca tanto giusto quanto un sistema di voto può esserlo.

Infine portate il voto in pubblico. Non c'è bisogno di segretezza o anonimato nel voto su questioni che avete dibattuto pubblicamente comunque. Ogni partecipante deve postare il suo voto nella mailing list, in modo che ogni osservatore possa registrare e controllare il risultato per se stesso, e in modo tale che ogni cosa sia registrata negli archivi.

Quando Votare

La cosa più difficile nel voto è determinare quando farlo. In generale passare al voto dovrebbe essere molto raro—un ultimo ricorso quando tutte le altre opzioni sono fallite. Non pensate che il voto sia un gran modo di risolvere i dibattiti. Non lo è. Esso mette fine alle discussioni e quindi mette fine a un modo creativo di pesare ai problemi. Nella misura in cui la discussione continua, c'è la possibilità che qualcuno venga fuori con una nuova soluzione che piace a tutti. Ciò avviene sorprendentemente spesso. Un dibattito acceso può produrre un nuovo modo di pensare ai problemi, e portare a proposte che possono soddisfare tutti. Anche quando non vengono fuori nuove proposte, è usualmente ancora meglio venire a un compromesso che sostenere un voto. Dopo un compromesso, ognuno e un po' scontento, mentre dopo un voto, alcuni sono scontenti mentre altri sono contenti. Da un punto di vista politico è

preferibile la prima situazione: almeno ciascuno può avvertire di aver ricavato una ricompensa per la sua scontentezza. Egli può essere insoddisfatto, ma così è anche per gli altri.

Il principale vantaggio del voto è che finalmente sistema una questione in modo che ognuno possa andare avanti, ma esso la sistema con un conto delle teste, invece che con un dialogo razionale che porti tutti alla medesima conclusione. Più le persone hanno esperienza con i progetti open source, meno desiderose le trovo di sistemare le cose col voto. Invece essi cercheranno di esplorare soluzione non precedentemente considerate, o arrivare a un compromesso più forte di quanto avessero precedentemente pianificato. Sono disponibili varie tecniche per impedire di arrivare ad un voto prematuro. La più ovvia è semplicemente dire #Io penso che non siete ancora pronti per il voto#, e spiegare perché no. Un'altra è chiedere una informale (non vincolante) alzata di mano. Se il responso tende chiaramente da una parte o dall'altra, ciò spingerà alcuni a volere subito un compromesso, ovviando al bisogno di voto. Ma il modo più efficace è semplicemente offrire una nuova soluzione, o un nuovo punto di vista su una vecchia proposta, in modo che le persone si occupino nuovamente dei problemi invece di ripetere solamente i vecchi argomenti.

In certi rari casi ognuno può convenire che tutte le soluzioni di compromesso sono peggiori di quelle di non compromesso. Quando ciò avviene, votare è meno spiacevole, sia perché è più probabile arrivare a una soluzione migliore sia perché la gente non si dispiacerà troppo non importa quale sia il risultato. Anche allora il voto non dovrebbe essere affretto. La discussione che porta al voto è ciò che educa l'elettorato, cosicché fermare quella discussione può prematuramente abbassare la qualità del risultato.

(Notate che questo avvertimento a non chiamare il voto non si applica al voto di cambio inclusione descritto in sezione chiamata «Stabilizzare una Release» in Capitolo 7, *Confezione, Rilascio, e Sviluppo Quotidiano*. Lì il voto è più di un meccanismo di comunicazioni, un mezzo per registrare il coinvolgimento di uno nel processo di revisione del cambiamento, così che ognuno può dire quanta revisione ha avuto un dato cambiamento.)

Chi Vota?

Avere un sistema di voto solleva la questione dell'elettorato: chi accede al voto? Ciò ha la potenzialità di essere un problema sensibile, perché obbliga il progetto a riconoscere alcune persone come più coinvolte, o come aventi più giudizio di altre.

La miglior soluzione è assumere una distinzione esistente, l'accesso all'invio, e annettere il privilegio di votare ad esso. In progetti che offrono sia il pieno che il parziale accesso all'invio, la questione se coloro che hanno l'invio parziale possano votare dipende largamente dal procedimento col quale è concesso l'accesso all'invio. Se il progetto lo distribuisce liberalmente, per esempio come un modo di mantenere molti mezzi conferiti da terze parti nel deposito, allora dovrebbe essere chiaro che l'accesso all'invio parziale è in realtà solo per l'invio, non per il voto. Naturalmente permane la implicazione inversa: poiché i possessori del pieno invio *avranno* privilegi di voto, essi devono essere scelti non solo nella qualità di programmatori, ma anche in quella di membri dell'elettorato. Se qualcuno mostra tendenze distruttive o ostruzioniste nella mailing list, il gruppo dovrebbe esser molto prudente sul fatto di renderlo uno che può fare gli invii, anche se la persona è tecnicamente competente.

Lo stesso sistema di voto dovrebbe essere usato per scegliere nuove persone che possono fare l'invio, sia pieno che parziale. Ma qui c'è uno dei rari casi in cui la segretezza è appropriata. Non potete ricevere voti sulle potenziali persone con diritto di voto postati in una pubblica mailing list, perché la sensibilità (e la reputazione) dei candidati potrebbero essere ferite. Invece, il modo usuale è che uno con diritto di invio posti su una mailing list privata costituita solo da altri con diritto di invio, che propongono che a qualcuno sia concesso l'accesso all'invio. Gli altri con diritto di invio parlano di quello che pensano liberamente, sapendo che la discussione è privata. Spesso non ci sarà disaccordo, e quindi non ci sarà necessità del voto. Dopo aver aspettato pochi giorni per essere sicuri che tutti quelli con diritto di voto abbiano avuto modo di rispondere, il proponente manda una email al candidato, e gli offre

l'accesso all'invio. Se c'è disaccordo, ne deriva una discussione come per ogni altra questione, con la possibilità che si arrivi a votare. Perché questo procedimento sia aperto e franco, il solo fatto che la discussione sta avendo luogo dovrebbe punto essere segreto. Se la persona in considerazione sapesse che essa sta andando avanti, e che quindi non gli è offerto l'accesso all'invio, potrebbe concludere che ha perso il voto, e presumibilmente ne sarebbe dispiaciuto. Certamente se qualcuno fa richiesta di accesso all'invio, allora non c'è scelta al di fuori di quella di prendere in considerazione la proposta e apertamente accettarla o respingerla. Se la seconda, allora la cosa dovrebbe essere fatta quanto più educatamente possibile, con una chiara spiegazione: #A noi piacciono le tue rettifiche, ma non ne abbiamo viste a sufficienza# o #Noi apprezziamo tutte le tue rettifiche, ma esse richiedevano considerevoli miglioramenti prima di poter essere applicate, così non ci sentiamo a nostro agio nel darti già l'accesso all'invio. Speriamo che ciò cambierà, tuttavia, col tempo# Ricordate, ciò che state dicendo potrebbe riuscire come un colpo, a seconda del livello e della confidenza con la persona. Cercate di vederla dal loro punto di vista quando gli mandate l'email.

Poiché aggiungere una nuova persona che possa fare gli invii è più conseguenziale che la maggior parte delle decisioni di una volta, alcuni progetti hanno speciali requisiti per il voto. Per esempio essi possono richiedere che la proposta riceva almeno *n* voti positivi e nessun voto negativo, che una super maggioranza voti a favore. Il parametro esatto non è importante; l'idea base è far si che il gruppo sia cauto nell'aggiungere nuove persone che possono fare l'invio. Simili o più stringenti requisiti possono applicarsi ai voti per *rimuovere* uno che può fare gli invii, sebbene si speri che ciò non sarà mai necessario. Vedere sezione chiamata «Quelli Che Fanno gli Invii» in Capitolo 8, *Gestire i Volontari* di più sugli aspetti del non voto per aggiungere o rimuovere le persone che possono fare gli invii.

Sondaggi Contro Voti.

Per certi tipi di voto può essere utile espandere l'elettorato. Per esempio, se gli sviluppatori semplicemente non possono capire se la scelta di una data interfaccia si adatta al modo in cui la gente in realtà usa il software, un modo è chiedere a tutti gli iscritti alla mailig list del progetto di votare. Questi sono in realtà *sondaggi* piuttosto che voti, ma gli sviluppatori possono scegliere di trattare il risultato come vincolante. Come con ogni sondaggio, assicuratevi che sia chiaro ai partecipanti che c'è una opzione inclusa: se qualcuno pensa che non sia offerta una migliore scelta nelle domande del sondaggio, la sua risposta può riuscire come il più importante risultato del sondaggio.

I Veti

Alcuni progetti permettono una specie di voto conosciuto come *veto*. Un veto è un modo per lo sviluppatore per mettere un alt a un frettoloso o mal considerato cambiamento. Pensate a un veto come a qualcosa fra una forte obiezione e una ostruzione. Il suo esatto significato varia da un progetto all'altro. Alcuni progetti rendono molto difficile ignorare un veto; altri consento loro di essere ignorati da un regolare voto della maggioranza, magari dopo un rallentamento forzato per una ulteriore discussione. Ogni veto dovrebbe essere accompagnato da una accurata spiegazione; un veto senza una tale spiegazione dovrebbe essere considerato invalido o una comparsa.

Con il veto viene l'abuso del veto. A volte gli sviluppatori sono troppo desiderosi di sollevare steccati per sbarazzarsi di un veto, quando in realtà quello per cui erano stati chiamati era un maggior discussione. Potete prevenire un abuso di veto con l'essere molto riluttanti verso il veto voi stessi e richiedendolo quando qualcun altro usa il suo veto troppo spesso. Se necessario potete ricordare al gruppo che i veti sono vincolanti solo finché il gruppo è d'accordo che lo siano dopotutto se una chiara maggioranza di sviluppatori vuole X, allora X sta per succedere in un modo o nell'altro. O gli sviluppatori che pongono i veti fanno marcia indietro, o il gruppo deciderà di sminuire il significato di un veto.

Potete vedere gente scrivere #-1# per esprimere un veto. Questo uso proviene dall'Apache Software Foundation che ha estremamente strutturato il procedimento di voto e di veto descritto a http://

www.apache.org/foundation/voting.html. Gli standard Apache si sono diffusi agli altri progetti, e voi vedrete le loro convenzioni usate con varie sfumature in molti posti nel mondo dell'open source. Tecnicamente #-1# non indica sempre un veto formale anche in accordo con lo standard Apache, ma in modo informale ciò viene preso a significare un veto, o almeno una obiezione molto forte.

Come i voti, i veti possono applicarsi retroattivamente. Non sta bene per obiettare a un veto, sulla base del fatto che il cambiamento in questione è stato già inviato, o che l'iniziativa è stata presa (a meno che non sia qualcosa di irrevocabile, come l'emissione un comunicato stampa). D'altra parte un veto che arrivi con un ritardo di settimane o di mesi verosimilmente non è da prendere molto sul serio, nè dovrebbe esserlo.

Metter Giù Tutto Per Iscritto

A un certo punto il numero degli accordi e delle convenzioni che girano nel vostro progetto diventa talmente grande che avete bisogno di registrarli da qualche parte. Per dare una tale legittimazione documentale, chiarite che essa è basata sulle discussioni della mailing list e sugli accordi già effettivi. Quando componete il documento, fate riferimento alle discussioni rilevanti negli archivi della mailing list, e dove c'è un punto di cui non siete sicuri, chiedete. Il documento non deve contenere sorprese: esso non è la fonte di accordi, esso è solamente la loro descrizione. Certamente, se ha successo, la gente incomincerà a citarlo come una fonte di autorità in se stesso, ma ciò significa appunto che esso riflette la volontà generale del gruppo accuratamente.

Questo è il documento a cui si allude in sezione chiamata «Linee Guida per lo Sviluppatore» in Capitolo 2, *Partenza*. Naturalmente, quando il progetto è molto giovane, avrete da mettere giù linee guida senza il beneficio di una lunga storia del progetto da cui trarre ispirazione. Ma appena la comunità di sviluppo matura, potete adattare il linguaggio per rispecchiare le cose che via via vengono fuori.

Non cercate di essere completi. Nessun documento può contenere ogni cosa che la gente ha bisogno di sapere sulla partecipazione al progetto. Molte delle convenzioni che il progetto elabora rimangono per sempre non dette, mai menzionate esplicitamente, eppure accolte da tutti. Altre cose sono semplicemente troppo ovvie per essere menzionate, e solamente distrarrebbero dall'importante, ma non ovvio materiale. Per esempio non ha senso scrivere linee guida come #Siate puliti e rispettosi verso gli altri nella mailing list e non incominciate guerre di offese,# o #scrivete un codice chiaro, leggibile e libero da bugs#. Certamente queste cose sono desiderabili, ma poiché non c'è universo concepibile in cui esse potrebbero non essere desiderabili non vale la pena menzionarle. Se la gente è ruvida nella mailing list, o scrivere codice con bugs, essi non si fermeranno perché lo hanno detto le linee guida del progetto. C'è bisogno che queste situazioni siano affrontate quando nascono, non con ammonizioni generali ad essere buoni. D'altra parte se il progetto ha delle linee guida specifiche su *come* scrivere buon codice come le regole per documentare le API in un certo formato, allora queste linee guida devono essere messe giù nella maniera più completa possibile.

Una buona maniera per individuare cosa includervi, è basare il documento sulle domande che i nuovi arrivati fanno più spesso e sulle lamentele più frequenti degli sviluppatori. Questo non significa necessariamente che ciò dovrebbe finire nel foglio delle FAQ —probabilmente ciò ha bisogno di una struttura più narrativa di quanto le FAQ possano offrire; ma essa dovrebbe seguire lo stesso principio basato sul reale di dare un indirizzo ai problemi che realmente sorgono, piuttosto che quelli che voi anticipate possano sorgere.

Se il progetto è una benevola dittatura, o ha funzionari investiti di speciali poteri (presidente, poltrona, qualcos'altro), allora il documento è una buona opportunità di codificare le procedure di successione. A volte questo può essere semplice come nominare specifiche persone come sostituti nel caso che il BD lasci il improvvisamente il progetto per qualche ragione. Generalmente, se c'è un BD, solo il BD può andarsene nominando un successore. Se ci sono funzionari per elezione, allora la procedura di elezione e di nomina che è stata usata per sceglierli in primo luogo dovrebbe essere descritta nel documento.

Se non c'era nessuna procedura all'origine, allora ottenete il consenso sulla procedura nella mailing list *prima* di scriverla. La gente a volte può essere permalosa con le strutture gerarchiche, così il soggetto bisogna prenderlo con tatto.

Forse la cosa più importante è chiarire che i ruoli possono essere riconsiderati. Se le convenzioni descritte nel documento incominciano ad ostacolare il progetto, ricordate a tutti che si suppone che ci sia una viva riflessione delle intenzioni del gruppo, non un sorta di frustrazione o di blocco. Se qualcuno ha il vizio di chiedere inappropriatamente che le regole vengano riconsiderate ogni volta che le regole vanno a suo modo, voi non dovete discutere ciò con lui—a volte il silenzio è la miglior tattica. Se altre persone si aggiungono alla protesta, suoneranno insieme, e sarà ovvio che qualcosa bisogna cambiarla. Se nessuna altro si aggiunge, allora la persona non raccoglierà molto consenso e le cose resteranno così come sono.

Due buoni esempi di linee guida di un progetto sono il Subversion `hacking.html` file, a http://subversion.apache.org/docs/community-guide/, a e i documenti dell'amministrazione dell'Apache Software Foundation, a http://www.apache.org/foundation/how-it-works.html e http://www.apache.org/foundation/voting.html. La ASF in realtà è un insieme di progetti di software, legalmente organizzata come una organizzazione no profit, così i suoi documenti tendono a descrivere le procedure di organizzazione, più che le convenzioni di sviluppo. Esse sono anche letture di pregio, tuttavia, perché rappresentano l'esperienza accumulata di un gran numero di progetti open source.

Capitolo 5. I Soldi

Questo capitolo esamina come dare i finanziamenti a una ambiente di software libero. E' rivolto non solo agli sviluppatori che sono pagati per lavorare a progetti di software libero, ma anche ai loro managers, che hanno bisogno di capire le dinamiche sociali dell'ambiente di sviluppo. Nelle sezioni che seguono il destinatario (“tu”) si presume che sia sia lo sviluppatore pagato, sia chi lo dirige. Il consiglio sarà spesso lo stesso per ambedue; quando non lo è, il pubblico a cui si dirige sarà reso chiaro dal contesto.

Che le compagnie finanzino uno sviluppo di software libero non è un fenomeno nuovo. Una grande quantità di sviluppo è stato spesso sovvenzionato informalmente. Quando un amministratore di sistema scrive uno strumento di analisi del network per aiutare gli sviluppatori a fare il loro lavoro, allora lo posta on line, e raccoglie le correzioni dei bugs e i contributi alle funzionalità dagli altri amministratori, ciò che è avvenuto è che si è formato un consorzio non ufficiale. Il finanziamento del consorzio proviene dai salari dei sysadmins, e lo spazio per gli uffici e la banda per il network sono donati, sebbene in modo inconsapevole, dalle organizzazioni per le quali essi lavorano. Queste organizzazione traggono beneficio dall'investimento, ovviamente, anche non ne sono al corrente inizialmente.

La differenza oggi è che molti di questi sforzi stanno venendo formalizzati. Le compagnie sono divenute consapevoli dei benefici del software open source, e hanno incominciato a coinvolgersi di prima persona nello sviluppo. Anche gli sviluppatori hanno incominciato ad aspettarsi che i progetti veramente importanti attrarranno almeno donazioni, e possibilmente anche sponsors a lungo termine. Mentre la presenza dei soldi non ha cambiato le dinamiche base dello sviluppo di software libero, ha largamente cambiato la scala in cui ciò avviene, sia in termini di numero degli sviluppatori che in termini di tempo-per-sviluppatore. Ciò ha anche avuto effetto su quanti progetti sono organizzati, e su come le parti coinvolte interagiscono. I problemi non sono solamente quanti soldi si spendono o come viene valutato il ritorno dell'investimento. Essi riguardano anche l'amministrazione e il processo: come possono la struttura gerarchica di comando delle compagnie e le semi decentralizzate comunità di volontari dei progetti di software libero lavorare produttivamente insieme? Saranno sempre d'accordo su ciò che significa “produttivamente”?

Il sostegno finanziario è generalmente ben accetto dalle comunità open source. Esso può diminuire la vulnerabilità di un progetto alla Forza del Caso, la quale può spazzar via tanti progetti prima che decollino, quindi può rendere la gente più desiderosa di dare al software una chance—essi avvertono di star investendo il proprio tempo in qualcosa che sarà ancora in vita sei mesi da ora. Dopotutto la credibilità è contagiosa, a un certo punto. Quando, per esempio, L'IBM sostiene un progetto open source, la gente, in un certa misura sente che al progetto non sarà permesso di fallire, e la loro risultante buona volontà a impegnarsi devotamente ad esso può rendere ciò una profezia auto appagante.

In ogni caso il finanziamento porta anche una percezione di controllo. Se non gestiti con cura, i soldi possono dividere gli sviluppatori in gruppi ristretti e un gruppi non ristretti. Se i volontari non pagati si fanno l'idea che le decisioni della progettazione o l'aggiunta di funzionalità sono solo una prerogativa del miglior offerente, allora essi volteranno le spalle a un progetto che sembra più simile a una meritocrazia e meno simile a un lavoro non pagato a beneficio di qualcun altro. Essi non possono mai lagnarsi apertamente nella mailing list. Invece ci sarà sempre meno e meno rumore da parte di fonti esterne, quando i volontari smetteranno di cercare di essere presi seriamente. Il mormorio dell'attività di piccola scala continuerà, nella forma di rapporti di bugs e di piccole correzioni. Ma non ci sarà un largo contributo o una partecipazione esterna alle discussioni sul progetto. La gente sente ciò che ci si aspetta da loro e sta al di sopra o al di sotto di queste aspettative.

Anche se i soldi bisogna usarli con cura, ciò non significa che non possano acquistare influenza. Nella maggior parte dei casi certamente lo possono. Il trucco è che non possono acquistarla direttamente. In una diretta transazione commerciale, voi cambiate moneta per ciò che volete. Se avete il bisogno che una funzionalità venga aggiunta, sottoscrivete una contratto, pagate per esso, e ciò viene fatto. In un

progetto open source, non è così facile. Voi potete sottoscrivere un contratto con alcuni sviluppatori, ma essi avranno ingannato se stessi —e voi—se essi hanno garantito che il lavoro per cui avete pagato sarebbe accettato dalla comunità di sviluppo perché avete pagato per esso. Il lavoro può essere accettato solo per i suoi meriti e per come si inserisce nella visione della comunità in relazione al software. Potete avere qualche voce in quella visione, ma non sarete l'unica voce.

Così i soldi non può comprare l'influenza, ma possono comprare cose che *portano* all'influenza. L'esempio più ovvio sono i programmatori. Se i programmatori sono noleggiati, e restano intorno al progetto abbastanza per acquisire esperienza e credibilità nella comunità, allora possono influenzare il progetto con gli stessi mezzi degli altri membri. Essi avranno un voto, o, se ce ne sono molti di essi, avranno un blocco votante. Se esso sono rispettati nel progetto, avranno influenza oltre il loro stesso voto. Non c'è bisogno per gli sviluppatori pagati disquisire sulle loro ragioni, nemmeno. Dopotutto chiunque vuole che in cambiamento sia apportato al software, lo vuole per una ragione. Ed è giusto che il peso dato agli obiettivi della vostra compagnia sia determinato dallo stato delle sue rappresentanze nel progetto, non dalla grandezza della compagnia, dal suo budget, o dal piano di affari.

Tipi di Coinvolgimento

Ci sono molte ragioni per cui un progetto open source viene finanziato. Le voci di questa lista non sono mutuamente esclusive; spesso il ritorno finanziario di un progetto risulterà da molte, o anche tutti questi motivi:

Suddividere il Carico
: Separate organizzazioni con software imparentato si trovano a dover duplicare gli sforzi, sia per il fatto di scrivere codice ridondante in casa, sia per il fatto di comprare prodotti simili dai venditori proprietari. Quando si accorgono di quello che sta avvenendo, le organizzazioni possono unire le loro risorse e creare (o unire) un progetto tagliato sulle loro necessità. I vantaggi sono ovvi: i costi dello sviluppo si dividono, ma i benefici si accumulano. Anche se lo scenario sembra intuitivo per i nonprofit, esso può dare un senso di strategico anche per i concorrenti for profit.

  Esempi: http://www.openadapter.org/, http://www.koha.org/

Aumentare i Servizi
: Quando una compagnia mette in vendita i servizi dai quali dipende, o per i quali diviene più attrattiva, particolari programmi open source, è naturale l' interesse di quella compagnia ad assicurare che siano attivamente conservati.

  Esempio: CollabNet's [http://www.collab.net/] supporto di http://subversion.tigris.org/ (esclusioni di garanzia: cioè il lavoro della mia giornata ma è anche un perfetto esempio di questo modello).

Supportare la vendita di hardware
: Il valore dei computers e dei componenti è direttamente correlato al software disponibile per essi. I venditori di hardware, non solo i venditori dell'intera macchina, ma anche i costruttori di periferiche e di microchips hanno trovato che l'avere software libero da far girare sul loro hardware è importante per i clienti.

Scalzare un concorrente
: A volte le compagnie sostengono un progetto open source come mezzo per scalzare un prodotto concorrente, che a sua volta potrebbe o non potrebbe essere open source. Distruggere poco a poco una quota del mercato concorrente, usualmente non è la sola ragione per coinvolgersi in un progetto open source, ma può essere un fattore.

  Esempio: http://www.openoffice.org/ (no questa non è la sola ragione per cui esiste Openoffice, ma il software è almeno in parte una risposta a Microsoft Office).

Il marketing
: Avere la vostra compagnia associata a un progetto open source può essere un affare di buon marchio.

Doppia licenza
: *Doppia licenza* è la pratica di offrire software sotto una tradizionale licenza proprietaria a clienti che lo vogliono come parte di una applicazione proprietaria per se stessi, e simultaneamente sotto una licenza libera per coloro che vogliono usarlo sotto i termini della licenza open source (vedere sezione chiamata «Gli Schemi a Doppia Licenza» in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti*). Se la comunità di sviluppatori open source è attiva, il software gode di una larga area di debugging e di sviluppo, e la compagnia pure ricava un flusso di denaro per sostenere sviluppatori a tempo pieno.

  Due ben noti esempi sono MySQL [http://www.mysql.com/], costruttori di software per database dello stesso nome, e Sleepycat [http://www.sleepycat.com/], che offre distribuzione e supporto per. Non è una coincidenza che esse siano entrambe compagnie di database. Il database tende ad essere integrato nelle applicazioni, piuttosto che essere venduto direttamente agli utenti, cosicché è molto più adatto al modello di doppia licenza.

Le donazioni
: Un progetto largamente usato può ricevere a volte significativi contributi, da individui e organizzazioni, giusto per il fatto di avere un pulsante di donazioni on line, o per vendere merce col marchio come tazze di caffè, T-shirts, cuscinetti per il mouse, ecc... Una parola di attenzione: se il vostro progetto accetta donazioni, pianificate come verrà usato il danaro *prima* che arrivi e pubblicate il piano sul sito web. Le discussioni su come destinare il danaro tendono ad andare molto meno regolarmente si tengono prima che ci sia reale danaro da spendere; e, in ogni modo, se ci sono dei rilevanti disaccordi, è meglio scoprire che esso (il danaro) è fuori fintanto che è ancora accademico.

Un modello di fondazione per affari non è il solo fattore su come si mette in relazione alla comunità di sviluppatori. Interessa anche la relazione storica fra i due: la compagnia avviò il progetto, o si sta associando a uno sforzo esistente? In ambedue i casi, il fondatore deve guadagnarsi credibilità, ma, non in modo sorprendente, c'è tanto in più guadagno da fare nel secondo caso. L'organizzazione deve avere chiari obiettivi rispetto al progetto. La compagnia sta cercando di avere una posizione di leadership, o soltanto di essere una voce nella comunità, per guidare e non necessariamente governare le direzione del progetto? O vuole giusto avere a disposizione una coppia di persone raccomandate, capaci di correggere i bugs dei clienti e introdurre i cambiamenti nella distribuzione pubblica, senza affanni?

Tenete queste questioni in mente quando leggete le linee guida che seguono. Si intende che esse siano applicate a una sorta di coinvolgimento organizzativo in un progetto di software libero, ma ogni progetto è un ambiente umano, e quindi non ve ne sono due esattamente simili. In qualche grado, voi avrete da giocare ad orecchio, ma seguendo questi principi accrescerete la probabilità che le cose vadano per il verso che volete.

Pagate Per il Lungo Termine

Se state dirigendo programmatori in un progetto open source, teneteli abbastanza affinché essi acquisiscano esperienza tecnica e politica—un paio di anni come minimo. Certamente nessuno progetto nè open né closed source trae beneficio dall'uscita e dall'ingresso troppo frequente. Il bisogno di un nuovo arrivato di imparare le cordate dovrebbe essere un deterrente in ogni ambiente. Ma la penalizzazione è anche più forte nei progetti open source, perché i programmatori che escono portano con sé non solo la loro conoscenza del codice, ma anche il loro status nella comunità e le relazioni umane che hanno stabilito lì.

La credibilità che uno sviluppatore ha accumulato non può essere trasferita. Per fare il più ovvio esempio, uno sviluppatore che arriva non può ereditare l'accesso all'invio da uno che se ne va (vedere sezione chiamata «Il Danaro Non Può Comprare Ciò Che Amate» più avanti in questo capitolo), così se il nuovo sviluppatore non ha già l'accesso all'invio, avrà da inviare correzioni fin quando lo avrà. Ma l'accesso all'invio è solo la più misurabile manifestazione di perduta influenza. Uno sviluppatore di lungo termine conosce anche tutti i vecchi argomenti che sono stati triti e ritriti nelle liste di discussione. Un nuovo sviluppatore, non avendo memoria di quelle discussioni, può cercare di riportare a galla gli argomenti, portando a una perdita di credibilità per la vostra organizzazione. Gli altri potrebbero meravigliarsi “Non possono ricordarsi di ogni cosa? Un nuovo sviluppatore non avrà nemmeno un senso politico delle personalità del progetto, e non saranno capaci di influenzare gli orientamenti del progetto così rapidamente o così regolarmente come lo fa uno che vi è rimasto per lungo tempo.

Preparate i nuovi arrivati con un programma di ingaggio controllato. Il nuovo sviluppatore deve essere in contatto la comunità di pubblico sviluppo sin dal primo giorno, partendo con la correzione di bugs e le operazioni di ripulita, in modo che possano imparare il codice base e acquistare una reputazione nella comunità, e non accendere lunghe e complesse discussioni di progetto. In tutto questo tempo, uno o più sviluppatori dovrebbero essere a disposizione per domande e risposte, e dovrebbero leggere ogni post che i nuovi arrivati facciano nella mailing list degli sviluppatori, anche se in essa ci sono threads a cui gli sviluppatori non rivolgerebbero l'attenzione. Ciò aiuterebbe il gruppo a scorgere le potenziali rocce prima che i nuovi arrivati vi rimangano incagliati. Incoraggiamenti privati, dietro le scene e consigli possono anche aiutare molto, specialmente se il nuovo arrivato non è abituato ad una massiccia parallela revisione fra pari del suo codice.

Quando CollabNet ingaggia un nuovo sviluppatore per lavorare a Subversion, noi ci sediamo intorno a un tavolo e scegliamo qualche bug aperto per la nuova persona perché lui ci affondi i denti. Noi discuteremo le linee bozza delle soluzioni, e poi assegneremo almeno uno sviluppatore esperto per (pubblicamente) revisionare le correzioni che il nuovo arrivato posterà. Tipicamente non guarderemo la correzione prima che la lista degli sviluppatori la veda, sebbene lo potremmo se vi fosse qualche ragione. La cosa importante è che il nuovo sviluppatore passi attraverso il processo della pubblica revisione, imparando il codice base e simultaneamente e abituandosi a ricevere critiche da parte di perfetti sconosciuti. Ma noi cerchiamo di coordinare i tempi in modo che la nostra revisione venga immediatamente dopo che la patch è stata postata. Così la prima revisione che la lista vede è la nostra, cosa che può essere utile e impostare il tono delle altre revisioni. Ciò contribuisce anche all'idea che la nuova persona deve essere presa seriamente: se gli altri vedono che noi stiamo impiegando il tempo a fare dettagliate revisioni con esaurienti spiegazioni e riferimenti negli archivi dove è appropriato, apprezzeranno ciò come una forma di training che sta andando avanti, e ciò probabilmente significa investimento a lungo termine. Questo li renderà più ben disposti verso lo sviluppatore, almeno nella misura di spendere un tempo extra nel rispondere alle domande e nel revisionare le correzioni.

Apparite Come Molti, Non Come Uno Solo

I vostri sviluppatori dovrebbero adoperarsi per apparire nei forums pubblici del progetto come partecipanti individuali, piuttosto che come una presenza monolitica collettiva. Ciò non perché ci sia un connotato negativo inerente a una presenza monolitica collettiva (bene, forse c'è, ma ciò non è il motivo per cui esiste questo libro). Piuttosto è perché gli individui sono una sorta di identità per cui i progetti open source sono equipaggiati per fare affari. Un collaboratore individuale può avere discussioni, inviare correzioni, acquistare credibilità, votare, e così via. Una compagnia no.

Inoltre, con l'avere un modo decentralizzato, voi evitate di stimolare una opposizione centralizzata. Lasciate che i vostri sviluppatori non siano d'accordo fra di loro nelle mailing lists. Incoraggiateli a revisionare il codice l'uno dell'altro tanto spesso, quanto pubblicamente, quanto farebbero ciascuno dell'altro. Scoraggiateli dal votare sempre in blocco, perché se lo facessero, altri potrebbero percepire che ciò, giusto in linea generale, sarebbe un sforzo organizzato per tenerli frenati.

C'è una differenza fra l'essere realmente centralizzati e semplicemente adoperarsi di apparire tali. In certe circostanze l'avere che i vostri sviluppatori si comportino in concerto può essere molto utile, ed essi dovrebbero essere preparati a coordinarsi dietro le quinte quando necessario. Per esempio, nel fare una proposta, avere molte persone che siano d'accordo in anticipo, può essere utile ad essa nel percorso, dando l'impressione di un consenso crescente. Gli altri avranno l'impressione che la proposta abbia forza, e che se facessero obiezione, fermerebbero quella slancio. Così la gente obietterà solo se avrà una buona ragione per farlo. Non c'è niente di sbagliato nell'orchestrare un accordo come questo, fin quando le obiezioni sono prese seriamente. Le manifestazioni pubbliche di un accordo privato non sono meno genuine per essere state coordinate in anticipo, e non sono dannose finché non son usate pregiudizialmente per spegnere gli argomenti dell'opposizione. Il loro scopo è unicamente quello di ostacolare quel tipo di persone che amano obiettare giusto per stare nella scia; vedere sezione chiamata «Più semplice l'argomento, più lungo il dibattito» in Capitolo 6, *Comunicazione* per maggiori ragguagli su di essi.

Siate Aperti Verso Le Vostre Motivazioni

Siate aperti verso gli obiettivi della vostra organizzazione quanto potete senza compromettere i segreti del business. Se volete che il vostro progetto acquisisca una certa funzionalità perchè, per esempio, i vostri clienti la hanno chiesta a gran voce, ditelo apertamente sulla maliling list. Se i clienti vogliono restare anonimi, come a volte è il caso, allora chiedete loro se vogliono essere usati come esempi non nominati. Quanto più pubblicamente la comunità di sviluppo conoscerà sul perchè volete ciò che volete, tanto più essa sarà accomodante su qualunque cosa stiate proponendo.

Ciò va contro l'istinto—così facile ad acquisire così facile a scrollarsi di dosso—che la conoscenza è potere e quanto più gli altri conoscono dei vostri obiettivi, tanto più controllo hanno su di voi. Sostenendo pubblicamente la funzionalità (una correzione di un bug, o qualcos'altro) voi avete *già* gettato la carte sul tavolo. La sola questione ora è se avrete successo nel guidare la vostra comunità nel condividere il vostro obiettivo. Se voi semplicemente stabilite ciò che volete, ma non fornite esempi concreti sul perché, il vostro argomento è fiacco, e la gente comincerà a sospettare una agenda nascosta. Ma se voi fornite uno scenario di poche parole che mostri perché la nuova funzionalità è importante, allora potete avere un sensazionale effetto sul dibattito.

Per vedere perché la cosa sta così, considerate l'alternativa. Troppo frequentemente il dibattito sulle nuove funzionalità è lungo e noioso. Gli argomenti che le persone avanzano spesso si riducono a “Io personalmente voglio X” o il sempre popolare “Nei miei anni di esperienza come progettista di software X è estremamente importate per gli utenti / un inutile fronzolo che non piacerà a nessuno”. Prevedibilmente l'assenza di informazione nelle parole reali né accorciano né moderano tali dibattiti, ma invece permettono che essi vadano ala deriva lontano lontano da ogni ormeggio nella reale esperienza dell'utente. Senza una forza controbilanciante, il risultato finale molto verosimilmente non viene determinato da una persona che era la più chiara, o la più tenace, o la più anziana.

Come organizzazione con abbondanti dati disponibili sui clienti, avete la opportunità di fornire giusto questa forza bilanciante. Voi potete essere come una condotta per le informazione che potrebbero avere diversamente nessun mezzo per raggiungere la comunità di sviluppo. Il fatto che quella informazione può supportare i vostri desideri non costituisce nulla di imbarazzante. La maggior parte degli sviluppatori individualmente non ha una larga esperienza su come il codice che essi hanno scritto viene usato. Ogni sviluppatore usa il suo software nel suo modo caratteristico. Fin dove i modelli degli altri vanno, lui fa affidamento sull'intuizione e sulle ipotesi, e nel profondo del suo cuore, lui sa questo. Ma, fornendo dati credibili su un gran numero di utenti, voi date alla comunità di sviluppo pubblico qualcosa di simile all'ossigeno. Finché presenterete ciò nel modo giusto, essi gradiranno ciò entusiasticamente, e muoveranno le cose nella direzione che voi volete.

La chiave, certamente, è presentare la cosa ne modo giusto. Non lo fate mai insistendo che semplicemente per il fatto che avete a che fare con un gran numero di utenti e poiché essi hanno bisogno

di una data funzionalità (o così voi credete), allora la vostra soluzione dovrebbe essere implementata. Invece voi dovreste focalizzare i vostri post iniziali sul problema, piuttosto che su una particolare soluzione. Descrivete in gran dettaglio l'esperienza che i vostri clienti stanno facendo, offrite quanta più analisi vi è possibile, e quante soluzioni voi potete pensare. Quando la gente incomincia a far congetture sull'efficacia delle varie soluzioni, voi potete continuare ad utilizzare i vostri dati per sostenere o rifiutare quello che dicono. Voi potete avere una soluzione in mente per tutto il tempo, ma non sceglietela per speciali considerazioni all'inizio. Questo non è inganno, questo è il comportamento standard dell'onesto mediatore. Dopotutto il vostro vero obiettivo è risolvere il problema; una soluzione è solo un mezzo per quel fine. Se la soluzione che voi preferite è veramente la migliore, gli altri sviluppatori lo riconosceranno dal loro canto alla fine—e allora vi andranno dietro di loro spontanea volontà, che è molto meglio che se voi li minacciaste a implementarla. (C'è anche la possibilità che essi abbiano in mente un soluzione migliore)

Con questo non si vuole dire che non potete mai dichiararvi a favore di una specifica soluzione. Ma dovete avere la pazienza di vedere l'analisi che avete già fatto internamente ripetuta nella mailing list dello sviluppo pubblico. Non postate “Noi abbiamo discusso ogni aspetto di questo problema , ma non funziona per il motivo A, B e C. Una volta che pensate realmente al problema, l'unico modo per risolverlo è...” Il problema non è tanto che ciò suona arrogante tanto da dare l'impressione che avete *già* edicato alcune sconosciute (me, la gente presumerà, grandi) quantità di risorse analitiche al problema, a porte chiuse. Esso fa sembrare ciò come se comunque gli sforzi sono stati fatti, forse le decisioni sono state prese, che il pubblico non è informato, e che è una ricetta per il risentimento.

Naturalmente, *voi* sapete quanto sforzo avete internamente dedicato al problema, e quella consapevolezza è, in un cero modo, uno svantaggio. Ciò mette i vostri sviluppatori un uno spazio un pochino differente rispetto ad ogni altro nella mailing list, riducendo la loro abilità a vedere le cose dal punto di vista che non hanno ancora tanto pensato al problema. Prima potete indurre ogni altro a pensare alle cose negli stesi termini in cui lo fate voi, più piccolo sarà l'effetto di questa distanza. Questa logica non si applica solo a individuali situazioni tecniche, ma al più largo mandato di rendere i vostri obiettivi più chiari che potete. L'ignoto è sempre più destabilizzante del noto. E la gente capisce perché volete ciò che volete, essi si sentiranno a loro agio parlandovi anche quando sono in disaccordo. Se essi non possono capire ciò che vi fa fare tic, presupporranno il peggio, almeno alcune volte.

Non sarete capaci di pubblicare ogni cosa, e la gente non si aspetterà ciò. Tutte le organizzazioni hanno segreti, forse quelle con fini di profitto ne hanno di più, ma quelle noprofit ne hanno pure. Se dovete difendere un certo corso, ma non rivelate nulla sul perché, allora semplicemente offrite i migliori argomenti che potete impediti da quello svantaggio e accettate il fatto che non potete avere l'influenza che volete in quella discussione. Questo è uno dei compromessi che dovete fare per non avere la comunità di sviluppo sul vostro libro paga.

Il Danaro Non Può Comprare Ciò Che Amate

Se avete uno sviluppatore pagato nel progetto, allora stabilite delle linee guida su ciò che il denaro può comprare e ciò che non può comprare. Ciò non significa che dovete postare due volte al giorno nella mailing list ripetendo la vostra nobile e incorruttibile natura. Significa solo che dovreste essere in guardia nel caso doveste disinnescare le tensioni che *potrebbero* crearsi per i soldi. Non è il caso di di partire assumendo che le tensioni sono lì; dovete dimostrare una consapevolezza che esse hanno la potenzialità di nascere.

Un perfetto esempio di questo ci viene dal progetto Subversion. Subversion fu avviato nel 2000 da CollabNet [http://www.collab.net/], che era stata la principale finanziatrice sin dal suo inizio, pagando i salari di molti sviluppatori (esclusioni di garanzia: io sono uno di essi). Poco dopo che il progetto incominciò, noi ingaggiammo un altro sviluppatore, Mike Pilato, per congiungere gli sforzi. Ma allora la codifica era già partita. Anche se Subversion era ancora molto ai primi stadi, era già una comunità di sviluppo con una serie di regole base.

L'arrivo di Mike sollevò una interessante questione. Subversione aveva già una politica su come un nuovo sviluppatore consegue l'accesso all'invio. Primo, egli invia alla mailing list alcune correzioni. Dopo che un numero sufficiente di correzioni sono arrivate da parte di altri che hanno l'accesso all'invio, per vedere se il nuovo collaboratore sa quello che sta facendo, qualcuno propone che egli giusto invii direttamente (questa proposta è privata ed è descritta in sezione chiamata «Quelli Che Fanno gli Invii»). Dato per acquisito l'accordo di quelli che avevano l'accesso all'invio uno di essi invia una email al nuovo sviluppatore e gli propone l'accesso diretto all'invio al deposito del progetto.

CollabNet aveva ingaggiato Mike per lavorare specificatamente a Subversion. Fra quelli che già lo conoscevano, non c'era dubbio sulle sue capacità nello scrivere codice e sulla sua sollecitudine nel lavorare al progetto. Inoltre gli sviluppatori volontari avevano molto buone relazioni con gli impiegati di CollabNet, e molto probabilmente non avrebbero obiettato se noi avessimo dato l'acceso all'invio a Mike il giorno che fu ingaggiato. Ma noi sapevano che avremmo creato un precedente. Se noi avessimo concesso a Mike l'accesso all'invio per decreto avremmo detto che CollabNet aveva il diritto di ignorare le linee guida del progetto, semplicemente perché era il finanziatore principale. Mentre il danno di ciò non era necessariamente immediatamente evidente, avrebbe avuto il risultato che i non salariati si sarebbero sentiti privati del diritto di voto. Altre persone hanno da guadagnarsi il loro accesso all'invio —CollabNet giusto lo compra.

Così Mike accettò di di iniziare il suo impiego a CollabNet come qualsiasi altro volontario, senza l'accesso all'invio. Egli mandava correzioni alla mailing list, dove esse potevano essere revisionate, e lo erano, da chiunque. Noi anche dicemmo che stavamo facendo così deliberatamente nella mailing list, in modo che non si sarebbe potuta perdere la puntualizzazione. Dopo un paio di giorni di concreta attività di Mike, qualcuno (non ricordo se era un collaboratore di CollabNet, o no) lo propose per l'accesso all'invio, e lui fu accettato, come sapevamo che sarebbe stato.

Questo tipo di coerenza vi dà una credibilità che il denaro non può mai comprare. E la credibilità è una moneta da avere nelle discussioni tecniche: è una immunizzazione contro il fatto di avere i propri motivi messi in discussione in un secondo momento. A caldo sembrerà che la gente vinca la battaglia con argomenti non tecnici. Il finanziatore principale del progetto, a causa del suo profondo coinvolgimento e dell'ovvio interesse alle direzioni che i progetto prende, presenta un obiettivo più grande degli altri. Essendo scrupolosi nell'osservare le linee guida del progetto sin dalla partenza, il finanziatore si fa grande quanto gli altri.

(Vedere anche il blog di Denise Cooper a http://blogs.sun.com/roller/page/DaneseCooper/20040916 per una storia simile sull'accesso all'invio. Cooper era allora una “Diva Open Source”—ricordo che era il suo titolo ufficiale—e all'entrata del blog lei racconta come la comunità di sviluppo di Tomcat spinse Sun a mantenere i propri sviluppatori allo stesso standard di accesso all'invio degli sviluppatori non Sun.)

Il bisogno che i fondatori stiano alle stesse regole di chiunque altro significa anche che il modello di amministrazione della Benevola Dittatura (vedere sezione chiamata «I Dittatori Benevoli» in Capitolo 4, *L'Infrastruttura Sociale e Politica*) è leggermente più difficile da far cadere in presenza di un finanziamento, specialmente se il dittatore lavora per il finanziatore principale. Siccome una dittatura ha poche regole, è difficile per il finanziatore provare che essa sta venendo governata secondo gli standard della comunità, anche quando lo è. E' certamente non impossibile; essa richiede appunto un leader del progetto che sia capace di vedere le cose dal punto di vista degli sviluppatori esterni, allo stesso modo di quelli del finanziatore, e agisca in accordo con essi. Anche allora, è probabilmente una buona idea avere propositi non dittatoriali sedendo al vostro posto, pronti a essere messi in evidenza nel momento di di qualche indicazione di diffuso malcontento nella comunità.

La Contrattazione

Il lavoro a contratto necessita di essere trattato con cura nei progetti di software libero. Idealmente voi volete che un lavoro di imprenditore sia accettato dalla comunità di sviluppo e impacchettato

nella distribuzione pubblica. In teoria, non interesserebbe chi sia l'imprenditore, finché il suo lavoro è buono ed è fatto secondo le linee guida. Teoria e pratica possono a volte coincidere, anche: Un perfetto sconosciuto che si metta in evidenza con una buona patch, *sarà* generalmente capace di metterla nel software. Il problema è, è molto difficile produrre una buona patch per una crescita non banale o una nuova funzionalità quando si è veramente un perfetto sconosciuto; uno deve prima discutere quella con il resto del progetto. La durata della discussione non può essere predetta con precisione. Se il lavoratore a contratto è pagato ad ore, voi potete concludere pagando più di quanto aveste previsto; se lui è pagato con una somma secca, può concludere facendo più lavoro di quanto possa produrre.

Ci sono due modi per aggirare questo. Quello preferito è quello di fare una erudita congettura sulla lunghezza della durata del processo di discussione, basata su passate a esperienze, aggiungere qualche riempitivo per errori, e basare il contratto su quello. Ciò aiuta anche a suddividere il problema in due pezzi quanto più piccoli possibile, per aumentare la possibilità di predire ogni pezzo. L'altro modo è contrattare solamente per il rilascio di una patch, e trattare l'accettazione delle patches nel pubblico progetto, come una questione separata. Allora diventa molto più facile scrivere un contratto, ma siete inceppati con il il carico di mantenere una patch privata tanto a lungo quanto dipendete dal software, o almeno tanto a lungo quanto ci vuole a inserire la patch o l'equivalente funzionalità nella linea principale. Certamente, anche con il modo preferito, il contratto stesso non può esigere che la patch sia accettata nel codice, perché ciò comporterebbe vendere qualcosa che non è in vendita. (Cosa accadrebbe se il resto del progetto decidesse di non supportare quella funzionalità?). Comunque il contratto può richiedere un sforzo *bona fide* a far si che il cambiamento sia accettato dalla comunità, e che esso sia inviato al deposito se la comunità lo accetta. Per esempio, se il progetto aveva scritto standards riguardo ai cambiamenti al software, il contratto può far riferimento a quegli standards e specificare che il lavoro deve adattarsi ad essi. In pratica ciò si risolve nella maniera in cui uno spera.

La migliore tattica per contrattare con successo è ingaggiare uno degli sviluppatori del progetto preferibilmente uno con l'accesso all'invio come contraente. Ciò potrà sembrare una modo di comprare influenza, ebbene, lo è. Ma non è una forma corrotta come potrebbe sembrare. Una influenza di uno sviluppatore nel progetto e dovuta principalmente alla qualità del suo codice e alla sua interazione con gli altri sviluppatori. Il fatto che egli abbia il contratto per fare certe cose non eleva il suo stato in alcun modo, sebbene ciò possa far si che la gente lo osservi con più attenzione. La maggior parte degli sviluppatori non rischiano la loro posizione a lungo termine sostenendo una funzionalità fuori luogo o che non piace a molti. Infatti, ciò che conseguite, o dovreste conseguire quando assumete tale persona a contratto è il parere su quale sorta di cambiamento è verosimilmente accettato dalla comunità. Voi anche pervenite a un leggero cambiamento nelle priorità del progetto. Poiché l'elenco delle priorità è giusto una materia di chi ha tempo di lavorare a qualcosa, quando pagate per il tempo di qualcuno, voi fate sì che il suo lavoro salga un poco nella coda delle priorità. Questo è un ben compreso fatto di vita fra gli sviluppatori esperti open source, e almeno qualcuno di essi dedicherà attenzione al lavoro del lavoratore a contratto semplicemente perché sembra che ciò debba *essere fatto*, in modo tale che che essi si adoperano a che sia fatto bene. Forse essi non scriveranno nulla del codice, ma tuttavia discuteranno del progetto e della revisione del codice, ambedue delle quali cose possono essere molto utili. Per tutte queste ragioni, il lavoratore a contratto è dipinto al meglio dai ranghi di quelli già coinvolti nel progetto.

Ciò solleva immediatamente due questioni: i lavoratori a contratto devono essere sempre privati? E quando no lo sono, dovete preoccuparvi per il fatto che potreste creare delle tensioni nella comunità per il fatto che avete fatto contratti con alcuni e non con altri?

La cosa migliore è essere aperti sui contratti, quando potete. Diversamente il comportamento del lavoratore a contratto può sembrare strano agli altri nella comunità può darsi il suo dare improvvisamente e inspiegabilmente priorità a funzionalità per le quali non aveva mai avuto interesse in passato. Quando le persone gli chiedono perché le vuole ora, come può egli rispondere in modo convincente se non può parlare del fatto che è stato assunto per scriverle?

Allo stesso tempo nè voi né il lavoratore a contratto dovreste agire come se gli altri dovrebbero considerare il vostro aggiustamento come un buon affare. Troppo spesso io ho visto lavoratori a

contratto ballare il valzer nella mailig list dello sviluppo con l'atteggiamento che i loro posts dovrebbero essere presi più seriamente solamente perché pagati. Questo tipo di atteggiamento dà il segnale al resto del progetto che il lavoratore a contratto guarda al fatto del contratto come opposto al codice *risultato* del contratto—essere la cosa più importante. Ma dal punto di vista degli altri sviluppatori, solo il codice conta. Sempre, il fuoco dell'attenzione dovrebbe essere mantenuto sugli aspetti tecnici, non sui dettagli di chi ha pagato chi. Per esempio, uno degli sviluppatori nella comunità di Subversion tratta il contratto in una particolare graziosa maniera. Mentre discute i suoi cambiamenti in IRC egli vuol far menzione a parte (spesso in una privata osservazione, un *privmsg*, su IRC ad uno degli altri con accesso all'invio) che lui è stato pagato per il suo lavoro su questo particolare bug o funzionalità. Ma egli da anche tangibilmente l'impressione che avrebbe accettato di lavorare a quel cambiamento comunque, e che è felice che il denaro stia rendendogli possibile fare ciò. Egli può o non può rivelare la sua identità individuale, ma in ogni caso non si sofferma sul contratto. Le sue osservazioni su questo sono giusto un ornamento per una discussione diversamente tecnica su come fare qualcosa.

Questo esempio mostra un'altra ragione per cui è bene essere aperti sui contratti. Ci possono essere molte organizzazioni che sponsorizzano i contratti su un progetto open source, e se una conosce ciò che le altre stanno cercando di fare, esse possono essere in grado di mettere insieme le loro risorse. Nel caso sopra, il più grande finanziatore (CollabNet) non è coinvolto in ogni modo con questi contratti di lavoro a cottimo, ma la conoscenza che qualche altro sta sponsorizzando alcune correzioni di bugs permette a CollabNet di reindirizzare le sue risorse verso altri bugs, col risultato di una maggiore efficienza per il progetto nella sua interezza.

Si offenderanno alcuni sviluppatori perché altri sono pagati per lavorare al progetto? In generale, no, specialmente quando quelli che sono pagati sono stabilizzati, rispettati membri della comunità comunque. Nessuno si aspetta che il lavoro a contratto sia distribuito in modo uniforme fra tutti coloro che fanno gli invii. La gente capisce l'importanza di una relazione a lungo termine: le incertezze connesse col contratto, sono tali che una volta che hai trovato uno con cui poter può lavorare affidabilmente, sareste riluttanti a passare ad un'altra persona giusto a scopo di egualitarismo. Pensate a ciò così: la prima volta che ingaggiate, non ci saranno lamentele, perché chiaramente dovete scegliere *qualcuno*—non è una vostra mancanza il fatto che non potete prendere tutti. Più tardi, quando ingaggiate qualcuno per la seconda volta, questo è giusto il sentire comune: già lo conoscete, l'ultima volta con successo, così perché correre rischi non necessari. Così, è perfettamente naturale avere una o due persone cui rivolgersi nel progetto, invece di distribuire il lavoro uniformemente.

Revisione e Approvazione Dei Cambiamenti

La comunità è tuttavia importante per il successo del lavoro a contratto. Il suo coinvolgimento nel processo di progettazione e revisione per cambiamenti su misura non può essere un pentimento. Deve essere considerato parte del lavoro e pienamente compreso dal lavoratore a contratto. Non pensate all'esame accurato della comunità come a un ostacolo da superare pensate ad esso come a un libero tavolo di progetto e a un dipartimento di QA. E' un beneficio essere inseguiti come in una caccia, non solamente sopportatati.

Studio analitico: il protocollo di autenticazione di password CSV

Nel 1995 io ero una metà della partenership che fornì il supporto e la crescita del CSV (il Concurrent Versions System; vedere http://www.cvshome.org/). Il mio partner Jim ed io eravamo informalmente i sostenitori di CSV a quel punto; ma non avevamo mai pensato con attenzione a come dovevamo metterci in relazione alla comunità di sviluppo di CSV in maggioranza costituita da sviluppatori volontari. Noi avevamo giusto accettato che che essi mandassero le patches, e noi le avevamo applicate, e questo era praticamente come funzionava.

A quei tempi, un CSV in rete poteva essere realizzato soltanto su un remoto programma di login come `rsh`. L'uso la stessa password per l'accesso al CSV e al login era un ovvio rischio di sicurezza, e

diverse organizzazione furono rimandate nel tempo per questo. Una banca importante di investimenti ci ingaggiò per aggiungervi un ulteriore meccanismo di autenticazione, così che esse potessero usare il CSV in rete con sicurezza per il loro uffici.

Jim e io accettammo il contratto e ci sedemmo attorno a un tavolo per progettare il nuovo sistema di autenticazione. Ciò a cui arrivammo era molto semplice (gli Stati Uniti avevano esportato controlli su un codice crittografico all'epoca, cosìcchè il cliente capì che noi non potevamo implementare una efficace autenticazione), ma mentre non avevamo esperienza di progettazione di simili protocolli, tuttavia facemmo poche gaffes che sarebbero state ovvie per un esperto. Se ci fossimo preso il tempo per metter giù una proposta, e mostrarla agli altri sviluppatori per la revisione, avremmo intercettato questi errori in anticipo. Ma noi non facemmo mai così, perché non ci serviva pensare alla mailing list di sviluppo come una risorsa da usare. Noi sapevamo che la gente avrebbe probabilmente accettato qualunque cosa avessimo inviato e poiché, e—e poiché noi non cooscevamo ciò che non sapevamo—non ci infastidimmo a fare il lavoro in un modo visibile, per esempio, postando patches frequentemente, facendo piccoli, digeribili invii a una sezione speciale, ecc.. Il risultante protocollo di autenticazione non fu molto buono, certamente, una volta che diventò stabile, era difficile da migliorare, a causa di questioni di compatibilità.

La radice del problema non era una mancanza di esperienza; noi avremmo potuto facilmente aver imparato ciò che era necessario. Il problema era il nostro atteggiamento nei confronti della comunità di sviluppo dei volontari. Noi guardavamo all'accettazione dei cambiamenti come a una barriera da saltare, piuttosto che a un processo col quale la qualità dei cambiamenti poteva esser migliorata. Poichè confidavamo che ogni cosa che avessimo fatto sarebbe stata accettata (come lo era), facemmo uno sforzo piccolo per coinvolgere gli altri.

Certamente quando state scegliendo un imprenditore, volete qualcuno con le giuste capacità tecniche ed esperienza per il lavoro. Ma è anche importante scegliere qualcuno con una traccia dei precedenti comportamenti e realizzazioni di una costruttiva interazione con gli altri sviluppatori nella comunità. In questo modo voi state prendendo più di una singola persona; voi state prendendo un agente che sarà capace di disegnare una rete di esperienze in modo da essere sicuri che il lavoro sia fatto in modo solido e mantenibile.

Finanziare Attività di Non Programmazione

La programmazione è solo un parte di ciò che entra in un progetto open source. Dal punto di vista dei volontari del progetto è la parte più visibile e affascinante. Questo sfortunatamente significa che altre attività, come la documentazione, le prove formali, ecc.., possono talvolta essere ignorate, almeno in confronto alla quantità di attenzione che spesso riceve il software proprietario. Le compagnie sono spesso capaci di inventare questo, dedicando una parte della loro struttura interna di sviluppo di software a progetti open source.

La chiave per fare ciò con successo è trasferire tra i processi interni delle compagnie e quelli delle comunità di sviluppo pubblico. Tale trasferimento non è facile: spesso le due non sono una copia identica, e le differenze possono solo essere superate con l'intervento umano. Per esempio, la compagnia può usare un tracciatore di bugs differente da quello del progetto pubblico. Anche se esse usano un software di tracciamento identico, i dati immagazzinati in essi saranno molto differenti, perché un tracciamento di bugs richiesto da una compagnia è molto differente da quello di un comunità pubblica di software. Un pezzo di informazione che si mette a fare il tracciatore può aver bisogno di essere riflesso nell'altro, con porzioni riservate rimosse, o, in altra direzione, aggiunte.

Le sezioni che seguono riguardano la costituzione e il mantenimento dei ponti per superare le differenze. Il risultato finale dovrebbe essere quello che il progetto open source funziona meglio, la comunità riconosce l'investimento di risorse della compagnia, e anche non si avverte che la compagnia sta dirigendo in modo improprio le cose verso i sui obiettivi.

La Garanzia Della Qualità (cioè, Eseguire Prove Professionali)

Nello sviluppo di software proprietario, è normale avere gente unicamente destinata alla garanzia della qualità: ricerca dei bug, tests di prestazioni e scalabilità, controllo dell'interfaccia e della documentazione. Come regola, queste attività non sono inseguite tanto vigorosamente dalla comunità di volontari nel progetto di software libero. In parte perchè è difficile trovare lavoro volontario per un lavoro non affascinante come il testing, in parte perché la gente tende a dare per scontato che l'avere una vasta comunità di utilizzatori dà al progetto una buona copertura di testing, e, nel caso del testing delle prestazioni e della scalabilità, in parte perché i volontari spesso non hanno l'accesso alle risorse hardware, in qualche modo.

La convinzione che avere molti utilizzatori è equivalente ad avere molti che testano non è completamente senza fondamento. Certamente non ha senso assegnare persone che testano funzionalità base in una ambiente comune: i bugs saranno rapidamente trovati dagli utilizzatori nel naturale corso delle cose. Ma poiché gli utilizzatori stanno giusto cercando di di finire il lavoro, inconsciamente non partono con l'intenzione di esplorare casi sconosciuti di funzionamento ai limiti nelle funzionalità dei programmi, e sono propensi a lasciare certi tipi di bugs non trovati. Inoltre, quando scoprono un bug con un facile stratagemma, spesso implementano in silenzio lo stratagemma senza darsi noia di riportare il bug. Ancora più insidiosamente, il modo d'uso dei vostri clienti (le persone che guidano il *vostro* interesse nel software) può differire in in modo statisticamente significativo dal modo d'uso dell'Utilizzatore Medio Della Strada.

Un gruppo professionale do testing può scoprire questo tipo di bugs, e può farlo facilmente sia con il software libero sia con il software proprietario. La sfida è di riportare al pubblico il risultati del gruppo di testing in una forma utile. I gruppi di testing in azienda usualmente hanno un loro modo di riportare questi risultati, impiegando un linguaggio specifico della compagnia o una conoscenza di particolari clienti e del loro gruppo di dati. Tali rapporti sarebbero inadatti per il tracciatore di bug pubblico, sia a causa della loro forma e a causa della confidenzialità. Anche se il software tracciatore di bugs interno della vostra compagnia fosse lo stesso di quello usato nei progetti pubblici, l'amministrazione poterebbe aver bisogno di commenti specifici della compagnia e di cambiamento dei meta dati (per esempio sollevare una priorità interna dei problemi o programmare la sua risoluzione per un particolare cliente). Usualmente tali note sono confidenziali—talvolta non sono nemmeno mostrate al cliente. Ma anche quando esse non sono confidenziali, esse non riguardano il progetto pubblico, e quindi il pubblico non dovrebbe essere distratto da essi.

Tuttavia il cuore stesso del rapporto dei bugs *è*importante per il pubblico. Infatti un rapporto del vostro dipartimento di testing è più prezioso di un rapporto dagli utilizzatori in libertà, poiché il dipartimento di testing indaga su cose su cui altri non indagherebbero. Dato che è improbabile che voi otteniate quel rapporto di bugs da altra fonte, volete di sicuro preservarlo e renderlo disponibile per il progetto pubblico.

Per fare questo o un dipartimento QA può archiviare i problemi nel tracciatore di problemi pubblico, se lo trova comodo, o un intermediario (usualmente uno degli sviluppatori) può “trasportare” i rapporti del dipartimento interno di testing in nuovi problemi nel tracciatore pubblico. Trasporto significa semplicemente descrivere i bugs in un modo tale che non faccia riferimento all'informazione specifica del cliente (il sistema della ripetizione può usare dati del cliente, assumendo che egli lo approvi, certamente).

E' alquanto preferibile che il dipartimento QA archivi i problemi direttamente nel tracciatore pubblico. Ciò dà al pubblico una stima del coinvolgimento nel progetto della vostra compagnia: un utile riporto dei bugs conferisce alla vostra compagnia credibilità giusto come lo farebbe un contributo tecnico. Ciò anche dà agli sviluppatori una linea di comunicazione col gruppo di testing. Per esempio, se il gruppo interno di QA sta monitorando il tracciatore pubblico di problemi, uno sviluppatore può inviare una

correzione per un bug di scalabilità (per il quale lo sviluppatore può non avere le risorse per testarlo da se), e quindi aggiungere una nota al problema chiedendo al QA di vedere se la correzione ha avuto l'effetto desiderato. Aspettatevi un po' di resistenza da parte di qualche sviluppatore; i programmatori hanno la tendenza a guardare al QA come, nel migliore dei casi, al diavolo. Il gruppo QA può rimediare a questo trovando bugs significativi e mettendo in archivio rapporti comprensibili; d'altra parte se i loro rapporti non sono almeno buoni quanto quelli provenienti dalla comunità degli utilizzatori regolari, allora è inutile averli in relazione direttamente con il team di sviluppo.

In un modo o nell'altro, una volta che esiste un pubblico problema il problema originale dovrebbe far riferimento al problema pubblico per il contenuto tecnico. L'organizzazione e gli sviluppatori pagati possono continuare ad annotare i problemi interni con commenti specifici della compagnia quanto necessario, ma usare il problema pubblico per una informazione che dovrebbe essere disponibile per tutti.

Dovreste entrare in questo processo aspettandovi spese extra. Mantenere due problemi per un bug è, naturalmente, un lavoro maggiore che mantenerne uno. Il beneficio è che molti codificatori vedranno il rapporto e saranno capaci di contribuire a una soluzione.

La Consulenza Legale e la Difesa

Le compagnie per profitto o noprofit sono quasi le uniche entità che pongono l'attenzione sui complessi aspetti legali del software libero. Gli sviluppatori individuali spesso capiscono le sottigliezze delle varie licenze open source ma non hanno il tempo o le risorse per seguire in dettaglio la legge sul copyright, sul marchi o sul brevetto. Se la vostra compagnia ha un dipartimento legale, può aiutare il progetto nel curare l'aspetto legale del codice, e aiutare gli sviluppatori a capire i potenziali problemi legali dei brevetti e del marchio. La forma che questo aiuto può prendere è discussa in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti*. La cosa principale è essere sicuri che le comunicazioni fra il dipartimento legale e la comunità degli sviluppatori, se avviene punto, avvenga con il reciproco riconoscimento dei molto diversi universi da cui le parti provengono. Occasionalmente, questi due gruppi parlano uno dopo l'altro, ogni parte dando per scontato una comprensione dello specifico campo che l'altra non ha. Una buona strategia è avere un intermediario (usualmente, uno sviluppatore, oppure un legale con esperienza tecnica) che stia nel mezzo e medi finché sia necessario.

La Documentazione e l'Usabilità

La documentazione e l'usabilità sono ambedue i punti delicati nei progetti open source, sebbene io penso, almeno nel caso della documentazione, che la differenza fra il software libero e quello proprietario sia frequentemente esagerata. Nondimeno è nei fatti vero che molti software open source difettano di una documentazione di prima classe e di ricerca di usabilità.

Se la vostra organizzazione vuole contribuire a riempire questi vuoti, probabilmente la cosa migliore che può fare è assumere le persone che *non* sono sviluppatori regolari nel progetto, ma che saranno capaci di interagire produttivamente con gli sviluppatori. Non assumere sviluppatori regolari è un bene per due ragioni: uno, in quel modo non portate via tempo dal progetto; due, quelli vicini al software sono usualmente le persone sbagliate per scrivere la documentazione o studiare l'usabilità, perché non hanno la il pensiero di vedere il software dal punto di vista di un estraneo.

Comunque sarà necessario per chiunque lavori a questi problemi comunicare con gli sviluppatori. Trovate persone che siano abbastanza tecniche per parlare col team dei codificatori, ma non così esperti nel software da non potersi ancora immedesimare nei normali utilizzatori.

Un utilizzatore di medio livello è la persona giusta per scrivere una buona documentazione. Infatti, dopo che la prima edizione di questo libro fu pubblicata, ricevetti la seguente email da uno sviluppatore open source chiamato Dirk Reiners:

```
Un commento sui Soldi:: La documentazione e l'Usabilità: quando avevamo
qualche soldo da spendere e concludemmo che una guida per quelli che
cominciavano era la parte più critica  assumemmo un utilizzatore di medio
livello per scriverla. Egli era andato per induzione al sistema abbastanza
recentemente per ricordare i problemi, ma era passato attraverso di essi
per sapere come descriverli. Ciò gli permise di scrivere qualcosa che
serviva solo per le correzioni minori da parte di sviluppatori di base per
le cose che non aveva capito bene, ma che tuttavia trattava le 'ovvie' cose
che gli sviluppatori avrebbero tralasciato.

Il suo caso era persino migliore, come se fosse stato suo compito introdurre
un sacco di altra gente (studenti) nel sistema, in modo che egli unì
l'esperienza di tanta gente, che è qualcosa che fu una felice avvenimento
e che è probabilmente difficile da raggiungere nella maggior parte dei casi.
```

Procurare l'Hosting/Banda

Per un progetto che non stia usando un hosting confezionato gratuito (vedere sezione chiamata «Canned Hosting» in Capitolo 3, *L'Infrastruttura Tecnica*), procurare un server e una connessione a un netwoork e in modo molto rilevante, un sistema di aiuto all'amministrazione può essere di significativa assistenza. Anche se questo è tutto quello che la vostra organizzazione fa per il progetto, può essere moderatamente un modo efficace per ottenere una buona atmosfera di pubbliche relazioni, sebbene ciò non porterà ad una influenza sulla direzione del progetto.

a che fare con la vostra compagnia, anche se voi non contribuite per nulla allo sviluppo. Il problema è, gli sviluppatori sono al corrente di questa tendenza associativa eccessiva, e possono non essere contenti di avere il loro progetto sul vostro dominio a meno che voi non immettiate più risorse che non la sola banda. Dopotutto ci sono un sacco di posti che danno hosting di questi tempi. La comunità può eventualmente essere del parere che la relativa cattiva assegnazione del riconoscimento non equivale alla convenienza ottenuta con l'hosting e trasferire il progetto altrove. Così se volete fornire l'hosting, fatelo—ma o pianificate di essere coinvolti di più presto o state attenti a quanto coinvolgimento reclamate.

Il Marketing

Sebbene la maggior parte degli sviluppatori non gradirebbero ammetterlo, il marketing funziona. Una buona campagna di marketing *può* creare un ronzio intorno a un prodotto open source, anche al punto che avveduti codificatori si trovano che essi stessi ad avere pensieri vagamente positivi sul software anche per ragioni sulle quali non possono assolutamente mettervi le dita. Non spetta a me dissertare la dinamica della corsa agli armamenti del marketing in generale. Ogni compagnia coinvolta in software libero si troverà alla fine a considerare come mettere in commercio se stessa, il software e le sue relazioni col software: Il consiglio di sotto riguarda come evitare trabocchetti in tale impegno; vedere anche sezione chiamata «La Pubblicità» in Capitolo 6, *Comunicazione*.

Ricordate Che Siete Osservati

Nell'intento di mantenere dalla vostra parte la comunità degli sviluppatori volontari è*molto* importante non dire ciò non sia in modo dimostrabile vero. Verificate tutte le affermazioni con cura, prima di farle e date al pubblico i mezzi per verificare le vostre affermazioni da se. La verifica indipendente dei fatti è una parte importante dell'open source e si applica a molto più che al solo codice.

Naturalmente nessuno consiglierebbe alle compagnie di fare affermazioni non verificabili comunque. Ma con le attività open source, c'è una insolita gran quantità di gente con l'esperienza per verificare

le affermazioni le persone che trovano conveniente anche avere un accesso a internet a larga banda e i giusti contatti sociali per pubblicizzare le sue conclusioni in un modo da danneggiare, dovrebbero sceglierlo loro. Quando la Global Megacorp Chemical Industries inquina un corso d'acqua, questo è verificabile, ma solo da parte di scienziati esperti, lanciando alla gente di grattarsi la testa e di chiedersi cosa pensare. Invece il vostro comportamento nel mondo dell'open source non è solo visibile e registrato; è anche possibile per molta gente verificarlo indipendentemente, pervenire alle proprie conclusioni e propagare quelle conclusioni per via orale. Queste reti di comunicazioni sono già in piedi; esse sono l'essenza di come l'open source opera, e possono essere usate per trasmettere ogni sorta di informazione. La confutazione è usualmente difficile, se non impossibile, specialmente quando ciò che la gente sta dicendo è la verità.

Per esempio è giusto riferire alla vostra organizzazione di aver “fondato il progetto X” se realmente lo avete fatto. Ma non fate riferimento a voi stessi come i “costruttori di X” se la maggior parte del codice è stato scritto da estranei. Al contrario, non proclamate di aver profondamente coinvolto una comunità di sviluppatori volontari se chiunque può vedere nel vostro deposito e constatare che ci sono pochi o nessun cambiamento al codice provenienti dal di fuori della vostra organizzazione.

Non molto tempo fa, vidi un annuncio da parte di una ben nota compagnia, che affermava che la compagnia stava rilasciando un importante confezione di un software sotto una licenza open source. Quando l'annuncio iniziale uscì, diedi un'occhiata al deposito del controllo della ora pubblica versione e vidi che conteneva solo tre revisioni. In altre parole quelli avevano fatto una importazione iniziale del codice sorgente, ma difficilmente qualcosa era avvenuta da allora. Cosa che in se stessa non era —quelli avevano fatto solo un annuncio dopotutto. Non c'era motivo di aspettarsi una grande attività di sviluppo immediatamente.

Qualche tempo più tardi, quelli fecero un altro annuncio. Questo è quello che diceva, con il nome e il numero di versione sostituiti da pseudonimi:

> *Abbiamo il piacere di annunciare che in seguito a rigorosi tests da parte della Singer Community, Singer 5 per Linux e Windows sono pronti per scopi di produzione.*

Curioso di sapere cosa la comunità avesse scoperto “nei rigorosi tests” ritornai al deposito a vedere la storia dei loro cambiamenti recenti. Il progetto era ancora alla revisione 3. Apparentemente non avevano trovato una *sola* correzione di bug di pregio prima della release. Pensando che il risultato dei tests della comunità potessero essere stati registrati altrove, esaminai il tracciatore di bugs. Li c'erano esattamente aperti 6 problemi, 4 dei quali erano stati aperti ormai per diversi mesi.

Questo è incredibile, certamente. Quando i collaudatori controllano su un grande e complesso pezzo di software per un certo tempo, troveranno bugs. Anche se le correzioni di questi bugs non lo trasformano nella prossima release, uno tuttavia si aspetterebbe qualche attività di controllo della versione come risultato del processo di prova, o almeno qualche nuovo problema. Eppure a tutta apparenza, niente era avvenuto tra l'annuncio della licenza open source e le prima release open source.

Il punto non è che la compagnia stava mentendo sull'attività di prova della comunità. Non ho idea se lo stesse facendo o no. Ma essi erano ignari di quanto *sembrava* verosimile che essi stavano mentendo. Siccome né il controllo di versione né il deposito di controllo né il tracciatore di problemi davano indicazione che le asserite prove erano avvenute non avrebbero dovuto fare l'affermazione in primo luogo, o avrebbero dovuto fornire un link a qualche tangibile risultato di quelle prove (“Abbiamo trovato 287 bugs; cliccare qui per i dettagli”). La seconda cosa avrebbe permesso a chiunque di capire il livello dell'attività della comunità molto velocemente. Così com'era mi ci vollero pochi minuti per determinare che qualunque cosa fosse questa comunità di prova, non aveva lasciato tracce in nessuno dei posti usuali. Quello non fu un grande sforzo, e sono sicuro che non sono il solo che si prese il disturbo.

La trasparenza e la possibilità di verifica sono anche una parte importante della credibilità, certo. Vedere sezione chiamata «Riconoscimenti» in Capitolo 8, *Gestire i Volontari* per maggiori informazioni su questo.

Non Colpite Il Prodotto Open Source Concorrente

Astenetevi dal dare cattive opinioni sul software open source concorrente. E' perfettamente giusto fornire *fatti* —negativi cioè affermazioni facilmente confermabili del tipo spesso visto in buoni quadri comparativi. Ma caratterizzazioni di natura meno rigorosa è meglio evitarle, per due ragioni. Primo, esse possono portare a guerre di ingiurie che distolgono da discussioni produttive. Secondo, e più importante, alcuni dei vostri sviluppatori volontari nel *vostro* progetto possono anche allontanarsi per lavorare al progetto concorrente. Ciò è più probabile di quando potrebbe sembrare in un primo momento: i progetti sono già nello stesso dominio (che è il motivo per cuoi sono in competizione), e sviluppatori con esperienza in quel dominio possono dare contributi dovunque la loro esperienza è applicabile. Anche quando non c'è una diretta sovrapposizione di sviluppatori è probabile che gli sviluppatori del vostro progetto abbiano familiarizzato con gli sviluppatori del progetto affine. La loro capacità di mantenere rapporti costruttivi personali potrebbe essere intralciata da messaggi negativi di marketing.

Primo, esse possono portare a guerre di ingiurie che distolgono da discussioni produttive. Secondo, e più importante, alcuni dei vostri sviluppatori volontari nel vostro progetto possono anche allontanarsi per lavorare al progetto concorrente. Ciò è più probabile di quando potrebbe sembrare in un primo momento: i progetti sono già nello stesso dominio (che è il motivo per cuoi sono in competizione), e sviluppatori con esperienza in quel dominio possono dare contributi dovunque la loro esperienza è applicabile. Anche quando non c'è una diretta sovrapposizione di sviluppatori è probabile che gli sviluppatori del vostro progetto abbiano familiarizzato con gli sviluppatori del progetto affine. La loro capacità di mantenere rapporti costruttivi personali potrebbe essere intralciata da messaggi negativi di marketing. Colpire prodotti closed source concorrenti sembra essere largamente accettato nel mondo open source, specialmente quando questi prodotti sono creati dalla Microsoft. Personalmente io deploro questa tendenza (sebbene d'altra parte non ci sia nulla di sbagliato nella chiara comparazione dei fatti), non solo perché è rozzo, ma anche perché è dannoso per un progetto partire credendo che ciò sia una propria pubblicità e quindi ignorare i modi in cui la competizione può essere veramente migliore. In generale guardatevi dagli effetti che le leggi del marketing possono avere sulla vostra comunità di sviluppo. La gente può venire così eccitata nell'essere sostenuta dai dollari del marketing da perdere la propria obiettività sulle vere solidità e debolezze del suo software. E' normale, e anche previsto, per una compagnia di sviluppatori esibire un certo distacco nei confronti delle leggi del marketing, anche in forum pubblici. Chiaramente essi non dovrebbero venir fuori e contraddire il messaggio di marketing direttamente (a meno che esso non sia veramente sbagliato, tuttavia uno spera che quel tipo di cosa dovrebbe essere scoperta prima). Ma possono prendersi gioco di questo, di volta in volta, come modo per riportare il resto della comunità di sviluppo sulla terra.

Capitolo 6. Comunicazione

L'abilità di scrivere chiaramente è forse la più importante capacità che uno possa avere in un ambiente open source. Sulla lunga distanza, importa più del talento nella programmazione. Un bravo programmatore con pessime capacità di comunicazione fare una sola cosa per volta, e comunque avere dei problemi a convincere altri a prestargli attenzione. Ma un pessimo programmatore con buone capacità comunicative può coordinare e persuadere molte persone a fare molte cose diverse, quindi avere un effetto significativo sulla direzione e sulla forza del progetto.

Non sembra esserci molta correlazione, in nessuna direzione, tra l'abilità di scrivere buon codice e l'abilità di comunicare con i propri simili. C'è una qualche correlazione tra la buona programmazione e una buona descrizione dei problemi tecnici, ma descrivere problemi tecnici è solo una piccola parte della comunicazione in un progetto. Molto più importante è l'abilità di identificarsi con la audience di qualuno, di vedere i propri post e commenti come altri li vedono, e di fare in modo che altri vedano i loro post con simile oggettività . Altrettanto importante è notare quando un certo mezzo di comunicazione non sta più lavorando bene, magari perchè non scala al crescere del numero di utenti, e prendere tempo per farci qualcosa.

Tutto questo è in teoria ovvio—cosa lo rende difficile in pratica è che gli ambienti di software libero sono practice is that free software development environments are incoerentemente diversi sia nelle audience che nei meccanismi di comunicazione. Un certo pensiero deve esserre espresso in un messaggio sulla mailing list, come annotazione nel bug tracker o come commento nel codice? Quando rispondere ad una domanda in forum pubblico, quanta conoscenza si può assumere esistere dal lato di chi legge, dato che "chi legge" non sempre è solo chi ha posto la domanda all'inizio, ma tutti quelli che potrebbero vedere la risposta? Come possono gli sviluppatori stare in contatto costruttivo con gli utenti, senza essere sommersi dalle richieste di funzionalità , scarne segnalazioni di bug e chiacchiera generale? Come dire quando un canale ha raggiunto i limiti della sua capacità , e cosa farci?

Le soluzioni a questi problemi sono solitamente parziali, perchè ogni singola soluzione è alla fine resa obsoleta dalla crescita del progetto o dai cambiamenti nella struttura del progetto. Sono anche spesso*ad hoc*, perchè sono improvvisato risposte a situazioni dinamiche. Tutti i partecipanti possono diventare consci di quando e come la comunicazione può crollare, ed essere coinvolti nelle soluzioni. Aiutare la gente a fare questo è una grossa parte della gestione di un progetto open source. Le sezioni che seguono trattano sia di come portare avanti la propria comunicazione. e di come rendere una priorità la cura dei meccanismi di comunicazione per tutti nel progetto.[1]

Sei quello che scrivi

Considerate questo: la sola cosa che ognuna sa di voi su Internet viene da quello che scrivete, o cosa altri scrivono di voi. Potrete essere brillanti, attenti e carismatici di persona—ma se le vostre email sono confusionarie e non strutturate, la gente assumerà che siate così. O magari siete davvero confusionari e disordinati di persona, ma nessuno ha bisogno di saperlo, se i vostri messaggi sono lucidi e informativi.

Dedicare un po' di cura alla vostra scrittura vi ripagherà enormemente. Lo smanettone di lunga data di free software Jim Blandy racconta la seguente storiella;

> Nel 1993, stavo lavorando per la Free Software Foundation, e stavamo facendo il beta testing della versione 19 di GNU Emacs. Facevamo un rilascio della beta all'incirca ogni settimana, e la gente la provava e ci mandava le segnalazioni di bug. C'era questo

[1] C'è stata un po' di interessante ricerca accademica su questo argomento; per esempio vedi *Group Awareness in Distributed Software Development* di Gutwin, Penner, e Schneider. Questo articolo è stato online per un po', poi non disponibile, quindi online di nuovo a http://www.st.cs.uni-sb.de/edu/empirical-se/2006/PDFs/gutwin04.pdf. Quindi provate lì prima, ma siate pronti ad usare un motore di ricerca se si è di nuovo spostato.

> tizio ch nessuno di noi aveva mai incontrato di persona ma che faceva un gran lavoro: le sue segnalazioni di bug erano sempre chiare e ci portavano dritti al problema, e quando ci forniva un fix lui stesso, era quasi sempre corretta. Era incredibile.
>
> Prima che la FSF possa usare codice scritto da qualcun altro, dobbiamo fargli fare alcune pratiche legali per assegnare i diritti di copyright per quel codice alla FSF. Solo prendere codice da perfetti sconosciuti e buttarlo lì è la ricetta per un disastro legale.
>
> Ho mandato una email al tizio con le pratiche, dicendo, "Qui ci sono alcune pratiche di cui abbiamo bisogno, qui c'è cosa significa, firmi questa, fai firmare al tuo datore di lavoro quest'altra, e poi possiamo iniziare a integrare i tuoi fix. Grazie mille."
>
> Mi risponde dicendo "Non ho un datore di lavoro."
>
> Allora gli dico, "OK, va bene, fallo firmare dalla tua università e mandacelo indietro."
>
> Dopo un po' mi risponde di nuovo e dice, "Bè, veramente... ho tredici anni e vivo con i miei genitori."

Dato che il ragazzino non scriveva come un tredicenne, nessuno sapeva che lo fosse. In seguito ci sono alcuni modi per far fare una buona impressione anche alla vostra scrittura.

Struttura e Formattazione

Non cadete nella trappola di scrivere tutto come se fosse un messaggio di testo per il telefono cellulare. Scrivete frasi complete, mettendo la maiuscola alla prima parola di ogni frase, e usate interruzione di paragrafo dove necessario. Ciò è fondamentale nelle email e in altri scritti strutturati. In IRC o forum similarmente effimeri, va generalmente bene lasciare perdere le maiuscole usare abbreviazioni di espressioni comuni eccetera. Soltanto non portate queste abitudini in forum più formali, persistenti. Email, documentazione, segnalazioni di bug, e altre forme di scrittura pensate per persistere dovrebbero essere scritte usando la grammatica e la sintassi standard, e avere una struttura narrativa coerente Questo non è perchè c'è qualcosa di interiormente buono nel seguire regole arbitrarie, ma piuttosto queste regole *non* sono arbitrarie: sono evolute nella forme attuali perchè rendono il testo più leggibile, e dovreste seguirle per questa ragione. La leggibilità è desiderabile non solo perchè significa che più persone capiranno cosa scrivete, ma perchè vi fa apparire come il tipo di persona che usa del tempo per comunicare chiaramente: cioà qualcuno a cui valga la pena dare attenzione.

Soprattutto per le email, gli sviluppatori esperti di open source hanno stabilito alcune convenzioni:

Mandare solo email di puro testo, no HTML, RichText, o altri formati che potrebbero essere opachi ad alcuni lettori di email di solo testo. Impostate le vostre righe per essere lunghe circa 72 colonne. Non andate oltre le 80 colonne, che sono diventate lo standard *de facto* della lunghezza di terminale (cioè alcuni usano terminali più larghi, ma nessuno ne usa di più stretti). Facendo le vostre righe un po' *meno* di 80 colonne, lasciate spazio per alcuni livelli di caratteri di citazione aggiunti nelle risposte di altri senza costringere alla riformattazione del vostro testo.

Usate vere interruzioni di linea. Alcuni programmi di mail fanno una specie di falsa delimitazione di riga, quindi quando state scrivendo una mail, il display mostra delle interruzioni di linea che in realtà non ci sono. Quando la email viene spedita, potrebbe non avere le interruzioni di linea voi pensiate che abbia, e si disporrà in maniera orrenda sugli schermi di un po' di gente. Se il vostro programma può usare false interruzioni di linea, cercate una qualche impostazione che possiate attivare per fare in modo di mostrare le vere interruzioni di linea quando scrivete.

Quando si includono output video, stralci di codice, o altro testo preformattato, delimitatelo chiaramente, così che anche un occhio pigro possa facilmente vedere i confini tra le vostre parole e

il materiale che state evidenziando. (Non mi sarei mai aspettato di scrivere un consiglio come questo quando iniziai questo libro, ma più avanti su un gran numero di mailing list open source, ho visto gente mischiare testo da diverse fonti senza rendere chiaro cosa fosse cosa. L'effetto è molto frustrante. Rende i loro post decisamente più difficili da capire, e sinceramente fa vedere queste persone come un po' disordinate.)

Quando citate una email di qualcun altro, inserite le vostre risposte dove è più appropriato, in molti posti diversi se necessario, e tagliate via le parti della mail che non usate. Se state scrivendo un breve commento che riguarda all'intero messaggio, va bene *anticipare* il commento (cioè mettere la vostra risposta al di sopra del testo citato della email); altrimenti, dovreste prima citare la porzione rilevante del testo originale, seguita dalla vostra risposta.

Scegliete attentamente l'oggetto delle vostre email. E' la riga più importante della vostra email, perchè permette ad ogni altra persona nel progetto di decidere se leggerla o meno. I moderni software di lettura email organizzano gruppi di messaggi correlati in thread, che possono essere definiti non solo da un oggetto comune, ma da vari altri header (che a volte non sono mostrati) Ne consegue che se un thread inizia ad andare verso un nuovo argomento, potete—e dovreste—aggiustare di conseguenza l'oggetto quando rispondete. L'integrità del thread sarà preservata, grazie a quegli altri header, ma il nuovo oggetto aiuterà la gente che cerca un'idea del thread a sapere che il soggetto è cambiato. Similarmente, se davvero volete iniziare un nuovo argomento, fatelo mandando una nuova email, e non rispondendo a email esistenti e cambiandone l'oggetto. Altrimenti, la vostra email sarebbe ancora raggruppata con quelle dello stesso thread a cui state rispondendo, e quindi confonderebbe la gente nel pensare che sia su qualcosa che non è. Di nuovo, la pena non sarebbe solo lo spreco del loro tempo, ma la piccola falla nella vostra credibilità come qualcuno spigliato nell'uso dei mezzi di comunicazione.

Contenuto

Email ben formattate attraggono i lettori, ma il contenuto li mantiene. Nessun insieme di regole fisse può garantire un buon contenuto, certo, ma ci sono alcuni principi che lo rendono possibile.

Rendete le cose facili ai vostri lettori. Ci sono tonnellate di informazioni in giro in ogni progetto open source attivo, e non ci si può aspettare che i lettori siano familiari con la maggior parte dell'informazione —infatti, non ci si può aspettare che sappiano come diventarne familiari. Ovunque possibile, i vostri messaggi dovrebbero fornire informazione nella forma più appropriata per i lettori. Se dovete usare due minuti in più per cercare una URL di un particolare thread negli archivi della mailing list per risparmiare ai lettori di farlo, ne vale la pena. Se dovete spendere 5 o 10 minuti riassumendo le conclusioni fino a questo punto di un thread complesso, così da dare alla gente un contesto in cui capire il messaggio,allora fate così. Pensatela in questo modo: più un progetto ha successo, più alto sarà il rapporto lettori per scrittore in ogni dato forum. Se ogni messaggio che pubblicate è visto da *n* persone, allora quando *n* cresce, la convenienza di fare uno sforzo extra per risparmiare tempo a questa gente sale con lui. E quando la gente vi vede imporre a voi stessi questo standard, lavorerà per fare altrettanto nelle loro comunicazioni. Il risultato è, idealmente, un aumento dell'efficienza globale del progetto: quando c'è una scelta tra che *n* persone facciano uno sforzo e una persona farlo, il progetto preferisce la seconda ipotesi.

Non perdetevi in iperboli. Esagerare nei messaggi in linea è una tipica corsa alle armi. Per esempio, una persona che segnala un bug potrebbe preoccuparsi che gli sviluppatori non gli presteranno sufficente attenzione, quindi lo descriverà come un problema serio e bloccante che sta impedendo a lui (e a tutti i suoi amici/colleghi/cugini) di usare il software in maniera produttiva, quando è soltanto una piccola noia. Ma l'esagerazione non è limitata agli utenti—i programmatori a volte fanno lo stesso durante dibattiti tecnici, in particolare quando il disaccordo è su di una questione di gusto piuttosto che di correttezza:

> "Fare in questo modo renderebbe il codice totalmente illegibile. Sarebbe un incubo mantenerlo, rispetto alla proposta di J. Random..."

Lo stesso sentimento in realtà diventa *più forte* quando espresso in maniera meno netta:

> "Funziona, ma non è ideale in termini di leggibilità e mantenibilità , io penso. La proposta di J.Random evita questi problemi perchè..."

Non sarete in grado di liberarvi completamente dalle iperboli, e in generale non è necessario. Rispetto ad altre forme di cattiva comunicazione, l'iperbole non è globalmente dannosa—danneggia principalmente chi la fa. I destinatari possono compensarla, soltanto il mittente perde un po' più credibilità ogni volta. Quindi, per l'amore della vostra stessa influenza nel progetto, provate a stare nel lato della moderazione. In questo modo, quando voi *avete bisogno* di fare un'affermazione forte, la gente vi prenderà seriamente.

Controllate due volte. Per ogni messaggio più lungo di un paragrafo di media grandezza, rileggetelo dall'inizio alla fine prima di mandarlo ma dopo che pensate di averlo finito una prima volta. Questo è un consiglio noto a chiunque abbia seguito lezioni di composizione, ma è soprattutto importante nelle discussioni online. Dato che il processo di comporre online tende ad essere altamente discontinuo (durante la scrittura di un messaggio, potreste aver bisogno di andare indietro e controllare altre email, visitare alcune pagine web, usare un comando per catturare il suo output di debug eccetera) è incredibilmente facile perdere il vostro senso di posizione nella narrazione. I messaggi che sono stati composti in maniera discontinua e non controllati prima di essere inviati sono spesso riconoscibili come tali, soprattutto dallo smarrimento (o così si dovrebbe sperare) dei loro autori. Prendetevi del tempo per rivedere cosa mandate. Più i vostri messaggi stanno assieme strutturalmente, più saranno letti.

Tono

Dopo aver scritto migliaia di messaggi, probabilmente noterete il vostro stile diventare conciso. Questa sembra essere la norma nella maggior parte dei forum tecnici, e non c'è nulla di sbagliato di per sè. Un grado di concisione che sarebbe inaccettabile nelle normali interazioni sociali è semplicemente la normalità per chi ha a che fare con il free software. Qui c'è una risposta che una volta presi da una mailing list su qualche CMS (content management system) ben in evidenza:

```
Puoi spiegare un po' di più esattamente qual'è il tuo problema,eccetera?

Anche:

Che versione di Slash stai usando? Non l'ho capito dal tuo precedente messaggio.

Esattamente, come hai fatto il build del codice apache/mod_perl?

Hai provato la patch di Apache 2.0 di cui si è parlato su slashcode.com?

  Shane
```

Ora, *questo* è conciso! Nessun saluto, nessuna firma a parte il nome, e il messaggio stesso è solo un serie di domande poste nel modo più compatto possibile. La sua unica frase dichiarativa era una implicita critica al mio messaggio originale. Eppure, fui felice di vedere la email di Shane, e non presi la sua concisione come segno di nient'altro se non essere una persona occupata. Il mero fatto che stava chiedendo domande, invece di ignorare il mio messaggio, significava che voleva spendere del tempo sul mio problema.

Tutti i lettori reagiranno positivamente al suo stile? Non necessariamente; dipende dalla persona e dal contesto. Per esempio, se qualcuno ha appena scritto riconoscendo di aver fatto un errore (magari aveva segnalato un bug), e sapete dalla vostra esperienza passata che questa persona tende ad essere un po'

insicura, allora mentre potreste scrivere una risposta compatta, dovreste fare in modo di completarlo con un qualche tipo di presa di coscienza dei suoi sentimenti. Il grosso della vostra risposta può essere una precisa, ingegneristica analisi della situazione, concisa quanto volete. Ma alla fine, scrivete qualcosa che indichi che la vostra concisione non deve essere presa per freddezza. Per esempio, se avete appena dato una gran quantità di consigli esattamente su come la persona deve riparare il bug, poi concludete con "Buona fortuna, <il vostro nome>" per indicare che gli augurate del bene e che non è pazzo. Uno smiley strategicamente piazzato o altro genere di emoticon può a volte essere anche abbastanza per rassicurare un interlocutore.

Può sembrare strano focalizzarsi tanto sui sentimenti dei partecipanti quanto sulla superficie di cosa dicono, ma per dirla semplicemente, i sentimenti condizionano la produttività . I sentimenti sono importanti anche per altre ragioni, ma anche limitandoci a livelli puramente utilitaristici, possiamo notare che persone infelici scrivono software peggiore, e meno. Data la natura ristretta della maggior parte dei media elettronici, comunque, non ci sarà spesso alcuna indicazione di come una persona si sente. Dovrete farvi una educata idea basata su a) come la maggior parte della gente si sentirebbe in quella situazione, e b) cosa sapete di questa particolare persona da passate interazioni. Alcune persone preferiscono un atteggiamento diretto, e trattare semplicemente con chiunque come fossero faccia a faccia, con l'idea che se un partecipante non tira fuori che si sente in un particolare modo, allora uno non ha modo di trattare con lui come dovrebbe. Non seguo questo approccio, per alcune ragioni. Uno, la gente non si comporta così nella vita reale, quindi perchè dovrebbero online? Due, dato che la maggior parte delle interazioni avviene in forum pubblici, la gente tende ad essere ancora più riservata nell'esprimere emozioni che se fosse in privato. Per essere più precisi, spesso vogliono esprimere emozioni dirette agli altri, come gratitudine o rabbia, ma non emozioni dirette verso se stessi, come insicurezza o orgoglio. Comunque, la maggior parte degli umani lavora meglio quando sanno che gli altri sono al corrente del loro stato di pensiero. Prestando attenzione a piccoli indizi, potete solitamente indovinare giusto la maggior parte delle volte, e motivare la gente a rimanere coinvolta ad un livello maggiore di quello che altrimenti farebbero.

Certo non voglio dire che il vostro ruolo sia di essere un terapista di gruppo, aiutanto costantemente tutti a rimanere in contatto con i loro sentimenti. Ma facendo molta attenzione ai percorsi sul lungo periodo del comportamento umano, inizierete ad avere un'idea di loro come individui anche se non li avete mai incontrati faccia a faccia. Ed essendo sensibile al tono del vostro scrivere, potete avere una sorprendente influenza su come gli altri si sentono, per il bene finale del progetto.

Riconoscere la maleducazione

Una delle caratteristiche distintive della cultura open source è la sua nozione di cosa costituisce maleducazione e cosa no. Mentre le convenzioni descritte in seguito non sono peculiari dello sviluppo di software libero, nè del software in generale— dovrebbero essere familiari a chiunque lavori in matematica, scienze dure, o discipline ingegneristiche—il software libero, con i suoi confini porosi e il costante afflusso di nuove leve, è un ambienete dove la gente non abituata a queste convenzioni si contri con loro.

Cominciamo con le cose che *non* sono maleducate:

Il criticismo tecnico, anche quando diretto e non filtrato, non è maleducato. Invece, può essere una forma di adulazione: la critica sta dicendo, per implicazione, che vale la pensa prendere seriamente in considerazione il suo bersaglio, e vale la pena spenderci un po' di tempo. Vale a dire, più sarebbe stato facile ignorare il post di qualcuno, più diventa un complimento prendere del tempo per criticarlo (ovviamente a meno che la critica diventi un attacco *ad hominem* o qualche altra farma di palese maleducazione).

Anche domande grezze e spoglie, come quelle di Shane a me nella mail prima citata, non sono maleducate. Domanede che in altri contesti sembrano fredde, retoriche o persino ironiche, sono

spesso intese come serie, e non hanno nessun obbiettivo nascosto tranne ottenere informazioni il più velocemente possibile. La famosa domanda del supporto tecnico "Il tuo computer è attaccato alla corrente?" è un classico esempio di questo. La persona di supporto davvero ha bisogno di sapere se il tuo computer è attaccato alla corrente, e dopo i primi giorni di lavoro, si è stancato di premettere alla domanda qualche educato preambolo ("Chiedo scusa, vorrei soltanto farle alcune semplici domande per escludere alcune possibilità. Alcune di queste sono molto elementari,ma mi aiuti..."). A questo punto, non gli interessano più i preamboli, chiede direttamente: è attaccato o no? Domande simili sono fatte di continuo nelle mailing list di software libero. L'intento non è insultare il destinatario, ma per escludere velocemente le spiegazioni più ovvie (e magari le più comuni). I destinatari che capiscono questo e rispondono di conseguenza, guadagnano punti nell'ottenere una visione ampia senza chiedere pressantemente. Ma i destinatari che reagiscono male non devono essere neanche esclusi. E' solo uno scontro di culture, non è colpa di nessuno. Spiegate amabilmente che le vostre domande (o critiche) non avevano significati nascosti; erano solo tesi ad ottenere (o trasmettere) informazioni nel modo più efficiente possibile, nient'altro.

Cos'è allora maleducato?

Per lo stesso principio per cui un dettagliato criticismo tecnico è una forma di adulazione, non fornire critica di qualità può essere un insulto. Non intendo semplicemente ignorare il lavoro di qualcuno, sia esso una proposta, un cambiamento al codice, la segnalazione di un nuovo problema o altro. A meno che non abbiate esplicitamente promesso in anticipo una risposta dettagliata, va solitamente bene semplicemente non rispondere per niente. La gente assumerà che non avevate tempo di dire nulla. Ma se *rispondete*, non risparmiate; prendete il tempo per analizzare davvero le cose, fornire esempi concreti dove appropriato, spulciare negli archivi per cercare post correlati nel passato eccetera. O se non avete tempo di imbarcarvi in questo tipo di sforzo, ma tuttavia avete bisogno di scrivere qualche tipo di risposta veloce, allora scrivetelo apertamente nel messaggio ("Penso che ci sia un problema registrato per questo, ma sfortunatamente non ho avuto il tempo di cercarlo, mi dispiace"). La cosa principale è riconoscere l'esistenza della norma culturale, o assecondandola o riconoscendo apertamente di non aver avuto tempo. In entrambi i modi, la norma è rispettata. Ma non rispettarla, allo stesso tempo non spiegando *perchè* non l'avete fatto, è come dire che la discussione (e coloro che vi partecipano) non valeva il vostro tempo. E' meglio mostrare che il vostro tempo è prezioso essendo chiari che essendo pigri.

Ci sono certamente molte altre forme di maleducazione, ma la maggior parte di queste non sono peculiari del software libero, e il senso comune è una sufficiente guida per evitarle. Vedete anche sezione chiamata «Stroncate sul Nascere la Scortesia» in Capitolo 2, *Partenza*, se non l'avete ancora fatto.

Facce

C'è una regione nel cervello umano che è dedicata in maniera specifica al riconoscimento delle facce. E' informalmente nota come 'area fusiforme della faccia', e le sue capacità sono nella maggior parte innate, non imparate. Ne consegue che riconoscere gli individui è talmente una capacità cruciale per la sopravvivenza che abbiamo sviluppato hardware specializzato per farlo.

La collaborazione basata su Internet è quindi psicologicamente strana, perchè implica una stretta collaborazione tra esseri umani che non riescono praticamente mai ad identificarsi l'un l'altro con i metodi più naturali ed intuitivi: il riconoscimento facciale innanzitutto, ma anche il suono della voce, la postura eccetera. Per compensare a questo, provate ad usare un *nome immagine* consistente ovunque. Potrebbe essere la parte iniziale del vostro indirizzo email (la parte prima del simbolo @), il vostro username IRC, il nome che usate nei commit, lo username del tracciamento dei problemi, eccetera. Questo nome è la vostra "faccia" online : una breve stringa identificativa che provvede ad alcuni degli stessi usi della vostra vera faccia, anche se sfortunatamente non stimola lo stesso hardware integrato nel cervello.

Il nome immagine dovrebbe essere qualche permutazione intuitiva del vostro nome reale (il mio, per esempio, è "kfogel"). In alcune situazioni sarà comunque accompagnato dal vostro nome completo, per esempio nelle testate delle email:

```
From: "Karl Fogel" <kfogel@whateverdomain.com>
```

In realtà, ci sono due cose in questo esemio. Come menzionato prima, il nome immagine rimanda al nome reale in modo intuitivo. Inoltre, il nome reale è *reale*. cioè non è qualche appellativo costruito come:

```
From: "Fantastico Hacker" <fantasticohacker@undominio.com>
```

C'è un famosto fumetto di Paul Steiner, del 5 luglio 1993 uscito sul *The New Yorker*, che mostra un cane loggato ad un computer, guardare in basso e dire ad un altro in modo cospirativo: "In Internet, nessuno sa che sei un cane". Questo tipo di pensiero risiede probabilmente dietro a molte delle identità da fighi, auto-esaltanti che la gente si dà in rete—come se chiamarsi "Fantastico Hacker" farà davvero credere alla gente di *esserlo*. Ma il fatto rimane: anche se nessuno sa che sei un cane, sei ancora un cane. Una fantastica identità online non impressiona mai i lettori. Invece, li fa pensare che sei più forma che sostanza, o semplicemente che sei insicuro. Usate il vostro vero nome per tutte le interazioni, o se per qualche ragione avete bisogno di anonimato, allora costruite un nome che sembri come un nome perfettamente reale, ed usatelo consistentemente.

Oltre a tenere il vostro nome immagine consistente, ci sono alcune cose che potete fare per renderlo più attraente. Se avete un titolo ufficiale (per esempio "dottore", "professore", "direttore"), non ostentatelo, nè menzionatelo tranne quando è direttamente rilevante nella conversazione. Il mondo hacker in generale, e la cultura del software libero in particolare, tende a vedere l'esposizione del titolo come esclusiva e segno di insicurezza. Va bene se il vostro titolo appare come parte del blocco di firma standard alla fine di ogni email, solo non usatelo mai come mezzo per rinforzare la vostra posizione in una discussione—il tentativo garantisce fuoco di risposta. Volete la gente che rispetta voi, non il titolo.

Parlando dei blocchi di firma: manteneteli brevi e gradevoli, o meglio ancora, inesistenti. Evitate ingombranti disclaimer legali alla fine di ogni email, specialmente quando esprimono sentimenti incompatibili con la partecipazione ad un progetto di software libero. Per esempio, il seguente classico appare alla fine di ogni messaggio che un particolare utente manda su una pubblica mailing list di cui faccio parte:

```
OSSERVAZIONE IMPORTANTE

Se avete ricevuto questa email per errore o volete leggere il nostro disclaimer
per le email e la politica di monitoraggio, per favore fate riferimento sotto o co
il mittente.

Questa comunicazione proviene da Deloitte & Touche LLP.  Deloitte &
Touche LLP è una società a responsabilità limitata registrata in Inghilterra e Gal
il numero registrato OC303675. Una lista dei nomi dei membri è disponibile
per ispezioni a Stonecutter Court, 1 Stonecutter
Street, London EC4A 4TR, United Kingdom, la sede principale dell'attività ed uffic
Deloitte & Touche LLP è autorizzato e regolato dalla Financial Services Authority.

Questa comunicazione e tutti gli allegati contengono
informazioni che sono confidenziali e possono essere legalmente protette.
Sono per l'uso esclusivo dei destinatari. Se non siete il destinatario, per favore
```

```
notate che ogni forma di comunicazione, pubblicazione, copia o uso di questa comun
o delle informazioni contenute o degli allegati è strettamente proibita e può esse
Se avete ricevuto questa comunicazione per errore, per favore rimandatela con "ric
come oggetto a to IT.SECURITY.UK@deloitte.co.uk e poi cancellate la email e distru

Non si può garantire che le comunicazioni email siano sicure e senza errori,
dato che le informazioni possono essere intercettate, corrotte, stralciate, perse,
distrutte, in ritardo o incomplete, o contenenti virus. Non ci assumiamo
la responsabilità di nessuno di questi fatti o delle loro conseguenze. Chiunque co
con noi via email accetta il rischio di farlo.

Quando inviate ai nostri clienti, tutti le opinioni o i consigli contenuti in ques
email ed ogni allegato sono soggetti ai termini e alle condizioni espresse nella l
di ingaggio del cliente in uso a Deloitte & Touche LLP.

Opinioni, conclusioni e altre informazioni in questa email e tutti gli allegati ch
riguardano gli affari ufficiali della ditta non sono da lei forniti nè supportati.
```

Per qualcuno che si mostra appena per chiedere qualcosa ogni tanto, questo enorme disclaimer sembra un po' strano ma probabilmente non fa male. Comunque, se questa persona volesse partecipare attivamente al progetto, questa sbrodolata legale inizierebbe ad avere effetti più insidiosi. Manderebbe almeno due segnali potenzialmente distruttivi: primo, che questa persona non ha pieno controllo dei suoi strumenti—è intrappolato in qualche programma di posta aziendale che appioppa un noioso messaggio alla fine di ogni email, e non ha avuto modo di evitarlo— e, secondo, che ha poco o nessun supporto organizzativo per le sue attività di software libero. Si che l'organizzazione chiaramente non gli ha impedito di lasciare messaggi sulle mailing list, ma ha fatto sembrare i suoi messaggi nettamente ostili, come se il rischio di far uscire informazioni confidenziali debba smorzare tutte le altre priorità.

Se lavorate in un'azienda che insiste nell'aggiungere tali blocchi di firma a tutte le email in uscita, allora cercate di avere un account email gratis come, per esempio, gmail.google.com, www.hotmail.com, o www.yahoo.com, e usare questo come indirizzo per il progetto.

Evitare le Trappole Comuni

Non mandare messaggi senza motivo

Una trappola comune nella partecipazione a progetti in rete è pensare che dobbiate rispondere a tutto. Non dovete. Prima di tutto, solitamente ci saranno più thread che vanno avanti di quelli a cui potete star dietro, almeno dopo che il progetto ha passato i primi suoi mesi. Secondo, anche nei thread a cui avete deciso di partecipare, la maggior parte delle cose che la gente dice non avrà bisogno di risposta. I forum di sviluppo in particolare tendono ad essere dominati da tre tipi di messaggi:

1. Messaggi che propongono qualcosa di non banale

2. Messaggi che esprimono supporto od opposizione a qualcosa che qualcun altro ha detto

3. Messaggi di ricapitolazione

Nessuno di questi richiede *inerentemente* di risposta, in particolare se potete essere sicuri, basandovi sull'esperienza accumulata nei thread, che qualcun altro probabilmente dirà comunque cosa avreste detto. (Se siete preoccupati di essere presi in un ciclo di attesa-attesa perchè anche tutti gli altri stanno usando la stessa tattica, non siatelo; c'è praticamente sempre *qualcuno* là fuori che si sentirà di saltare nel mucchio.) Una risposta dovrebbe essere motivata da un proposito definito. Innanzitutto chiedetevi: sapete cosa volete raggiungere? E poi: non sarà raggiunto a meno che voi diciate qualcosa?

Due buone ragioni per aggiungere la vostra voce ad un thread sono a) quando vedete un difetto in una proposta e sospettate di essere l'unico a vederlo, e b) quando vedete che sta succedendo qualche equivoco tra altri, e sapete che potete appianarlo con un messaggio chiarificatore. Va solitamente bene fare un messaggio per ringraziare qualcuno per aver fatto qualcosa, o per dire "Anche io!", affinchè un lettore possa capire facilmente che tale messaggio non ha bisogno di nessuna risposta o ulteriore azione, e quindi lo sforzo mentale richiesto dal messaggio finisce nettamente quando il lettore arriva all'ultima riga delle email. Ma anche allora, pensateci due volte prima di dire qualcosa; è sempre meglio lasciare la gente desiderare che voi postiate di più invece che di meno. (Vedete la seconda metà di Appendice C, *Perchè dovrebbe importarmi di che colore sia la rastrelliera?* per ulteriori pensieri su come comportarsi su mailing list trafficate.)

Thread Produttivi vs Thread Improduttivi

Su una mailing list trafficata, avete due imperativi. Uno, ovviamente, è capire di cosa avete bisogno di seguire e cosa potete ignorare. L'altro è di comportarvi in modo da evitare di *causare* rumore: non solo volete che i vostri messaggi abbiano un alto tasso segnale/rumore, volete anche che siano quel tipo di messaggio che stimola *altra* gente a scrivere con un simile tasso segnale/rumore o non scrivere per niente.

Per vedere come fare ciò, considerate il contesto in cui avviene. Quali sono alcuni dei segnali di un thread improduttivo?

- Argomenti che hanno già iniziato ad essere ripetuti, dato che chi ha scritto pensa che nessuno li abbia sentiti la prima volta.
- Aumento dei livelli di iperbole e coinvolgimento dato che i limiti si fanno sempre più stretti.
- Una preponderanza di commenti di persone che fanno poco o niente, mentre le persone che tendono a fare le cose sono in silenzio.
- Molte idee discusse senza che una chiara proposta sia stata fatta. (Certo, ogni idea interessante nasce da una visione imprecisa; la domanda importante è in quale direzione si va da lì. Il thread sembra voler cambiare la visione in qualcosa di più concreto, o stanno nascendo sotto-visioni, visioni laterali e dispute ontologiche?)

Solo perchè un thread non è produttivo non basta per stabilire che sia una perdita di tempo. Potrebbe trattare un argomento importante, nel qual caso il fatto che non si stia risolvendo rende tutto più problematico.

Guidare un thread verso l'utilità senza essere pressanti è un'arte. Non funzionerà semplicemente consigliando alla gente di smettere di perdere il loro tempo, o chiedendo loro di non scrivere a meno di avere qualcosa di costruttivo da dire. Potreste, certo, pensare queste cose privatamente, ma se lo dite ad alta voce allora sarete offensivi. Invece, dovete suggerire le condizioni per ulteriori progressi—dare alla gente una strada, un sentiero da seguire che porta ai risultati che volete, pur senza sembrare di stare dettando la strada. La distinzione è principalmente di tono. Per esempio, questo non va bene:

> *Questa discussione non sta andando da nessuna parte. Possiamo per favore abbandonare questo argomento finchè qualcuno ha una patch per implementare una di queste proposte? Non c'è ragione per continuare a girarci attorno dicendo le stesse cose. Il codice parla più forte delle parole, gente.*

Mentre questo è buono:

> *Molte proposte sono passate in questo thread, ma nessuna ha avuto tutti i dettagli definiti, almeno non abbastanza per un voto si/no. Comunque non stiamo dicendo nulla di nuovo ora; stiamo solo ripetendo cosa è stato detto prima. Quindi la cosa*

> *migliore sarebbe probabilmente che i prossimi messaggi contengano o una completa specifica delle funzionalità proposte, o una patch. Quindi almeno avremmo una azione definita da compiere (cioè avere consenso sulla specifica, o applicare e testare la patch).*

Confrontate il secondo approccio con il primo. Il secondo modo non traccia una linea tra voi e gli altri, nè li accusa di procedere in una discussione a spirale. Parlo di "noi", che è importante sia che abbiate o meno partecipato veramente nel thread in precedenza, perchè ricorda a tutti che anche quelli che sono stati in silenzio fino ad ora possono ancora contribuire al risultato del thread. Descrive perchè il thread non sta andando da nessuna parte, ma lo fa senza peggiorazioni o giudizi—spassionatamente preciso solo alcuni fatti. Più importante, offre un corso positivo di azioni, così che invece di sentirsi come se la discussione sia stata troncata (una restrizione verso cui potrebbero solo tentare di ribellarsi), la gente si sentirà come se fosse stata loro offerto un modo di portare la conversazione ad un livello più costruttivo. Questo è uno standard che la gente vorrà naturalmente raggiungere.

Non vorrete sempre portare un thread al prossimo livello di costruttività—a volte vorrete solo farlo finire. Il proposito del vostro messaggio allora è fare uno o l'altro. Se potete dire dal modo in cui il thread è andato fino ad allora che nessuno da veramente *facendo* i passi che suggerite, allora il vostro messaggio effettivamente chiude il thread senza sembrare di farlo. Di certo non c'è nessun modo a prova di idiota per chiudere un thread, e anche se ci fosse, non vorreste usarlo. Ma chiedere ai partecipanti o di mostrare progressi visibili o di smettere di scrivere è perfettamente difendibile, se fatto diplomaticamente. Siate comunque attenti nel fermare prematuramente i thread. Un po' di chiacchiere possono essere produttive, a seconda dell'argomento, e chiedere che sia risolto troppo velocemente soffocherà il processo creativo, così come vi farà sembrare impazienti.

Non aspettatevi che un thread si stoppi all'istante. Ci saranno comunque ancora alcuni messaggi dopo al vostro, sia perchè le mail si saranno incrociate nell'instradamento, o perchè la gente vuole avere l'ultima parola. Questo non è nulla di cui preoccuparsi, e non avete bisogno di scrivere di nuovo. Lasciate la gente calmarsi, o non calmarsi, a seconda dei casi. Non potete avere completo controllo; dall'altro lato, potete aspettarvi di avere statisticamente un effetto significativo su molti thread.

Più semplice l'argomento, più lungo il dibattito

Anche se la discussione può deviare su ogni argomento, la probabilità di deviazione sale quando la difficoltà tecnica di un argomento diminuisce. Dopo tutto, maggiore è la difficoltà tecnica, meno partecipanti potranno veramente seguire cosa sta succedendo. Quelli che possono essere gli sviluppatori più esperti, che hanno già preso parte in queste discussioni migliaia di volte in precedenza, e sanno che tipo di comportamento può portarli ad ottenere quel consenso con cui ognuno può andare avanti.

Quindi, il consenso è più difficile da ottenere nelle questioni tecniche che sono semplici da capire ed è facile farsi un'opinione, ed in argomenti "leggeri" come l'organizzazione, la pubblicità, i fondi, eccetera. La gente può partecipare a queste discussioni sempre, poichè non sono necessarie qualifiche per farlo, nessun modo chiaro per decidere (anche dopo) se una decisione sia stata giusta o sbagliata, e perchè aspettare più a lungo degli altri partecipanti alla discussione è a volte una tattica vincente.

Il principio che la quantità di discussione è inversamente proporzionale alla complessità dell'argomento ha circolato per molto tempo, ed è noto informalmente come l'*Effetto Bikeshed*. Segue la spiegazione di Poul-Henning Kamp, da un messaggio ora famoso fatto agli sviluppatori BSD:

> E' una lunga storia, o meglio è una vecchia storia, ma in realtà è abbastanza breve.
> C. Northcote Parkinson scrisse un libro nei primi anni 60, chiamato "Parkinson's Law" ("Legge di Parkinson"), che contiene molti aspetti delle dinamiche della gestione.
>
> [...]

> Nello specifico esempio che coinvolge la rastrelliera delle biciclette, l'altro componente vitale è una centrale atomica, penso che questo illustri l'età del libro.
>
> Parkinson mostra come puoi andare nell'ufficio del direttore e ottenere l'approvazione per costruire una centrale atomica da milioni o persino miliardi di dollari, ma se volete costruire una rastrelliera per le biciclette sarete bloccati in discussioni senza fine.
>
> Parkinson spiega che questo accade perchè una centrale atomica è così vasta, così costosa e così complicata che la gente non può percepirla, e piuttosto che provarci, ricadono nell'assunzione che qualcun altro abbia controllato tutti i dettagli prima di andare così avanti. Richard P. Feynmann da alcuni esempi interessanti e molto pertinenti, riguardanti Los Alamos nei suoi libri.
>
> Dall'altro lato una rastrelliera per bici. Chiunque può costruirne una in un fine settimana, e ancora avere il tempo di guardare la partita in TV. Quindi non importa quanto ben preparato, quanto ragionevole con la vostra proposta, qualcuno coglierà la possibilità di mostrare che sta facendo il suo lavoro, che sta prestando attenzione, che è*qui*.
>
> In Danimarca lo chiamiamo "lasciare l'impronta". Riguarda l'orgoglio personale e il prestigio, si tratta di essere in grado di indicare da qualche parte e dire "Qui! *io* l'ho fatto." E' un importante tratto nei politici, ma presente in molta gente se viene data l'occasione. Pensate ai passi nel cemento fresco.

(Vale anche la pena leggere il suo messaggio completo. Vedete Appendice C, *Perchè dovrebbe importarmi di che colore sia la rastrelliera?*; o anche http://bikeshed.com.)

Chiunque abbia mai preso regolarmente parte in qualche gruppo di decision riconoscerà di cosa Kamp sta parlando. Comunque, è solitamente impossibile persuadere *tutti* di evitare di disegnare rastrelliere. La cosa migliore che possiate fare è precisare che il fenomeno esiste, quando vedete che sta succedendo, e persuadere gli sviluppatori anziani—le persone i cui messaggi hanno maggior peso—di posare i loro pennelli presto, così almeno loro non contribuiscono al rumore. Dipingere rastrelliere non scomparirà mai del tutto, ma potete renderlo più breve e meno frequente diffondendo la coscienza del fenomeno nella cultura del progetto.

Evitare le Guerre Sante

Una *Guerra Santa* è una disputa, spesso ma non sempre riguardo ad un problema relativamente secondario, in cui la gente si sente abbastanza appassionata da continuare a discutere in ogni caso nella speranza che la loro parte prevalga. Le guerre sante non sono come dipingere rastrelliere. La gente che dipinge rastrelliere è solitamente rapida nel saltare su con un'opinione (perchè loro possono), ma non se ne sentiranno necessariamente convinti, infatti potranno a volte esprimere altre opinioni incompatibili, per mostrare che capiscono tutti i versi del problema. In una guerra santa, d'altro canto, capire le altre posizioni è un segno di debolezza. In una guerra santa, tutti sanno che c'è Una Risposta Giusta; solo non sono d'accordo su quale sia.

Una volta che una guerra santa è iniziata, in genere non può essere risolta accontentando tutti. Non fa bene puntualizzare, nella mischia di una guerra santa, che una guerra santa è in corso. Tutti lo sanno già. Purtroppo, un tratto comune delle guerre sante è il disaccordo sulla domanda *se* la disputa è risolvibile continuando la discussione. Visto da fuori, è chiaro che nessuno schieramento sta cambiando le idee dell'altro. Visto da dentro, l'altro schieramento è ottuso e non sta pensando in modo chiaro, ma ci potrebbero arrivare se pressati abbastanza. Ora, *non* sto dicendo che non ci sia mai uno schieramento giusto in una guerra santa. A volte c'è—nelle guerre sante a cui ho partecipato, è sempre stato il mio ovviamente. Ma non importa, perchè non c'è un algoritmo per dimostrare in modo convincente che uno schieramento o l'altro abbia ragione.

Un modo comune ma non soddisfacente con cui la gente prova a risolvere le guerre sante è dire "Abbiamo già speso più tempo ed energia discutendo ciò di quanto ne valga la pena! Possiamo per favore semplicemente lasciare stare?" Ci sono due problemi in questo. Primo, questo tempo e questa energia sono già stati spesi e non potranno mai essere recuperati— l'unica domanda è quanto *altro* sforzo rimane? Se certa gente pensa che solo ancora un poco di discussione porterà il problema alla fine, allora ha ancora senso (dal loro punto di vista) continuare.

L'altro problema nel chiedere di lasciare perdere il problema è che questo è spesso equivalente a permettere ad uno schieramento, lo status quo, di dichiarare la vittoria per mancanza di azioni. E in alcuni casi, lo status quo è comunque noto per essere inaccettabile: tutti sono d'accordo che qualche decisione deve essere presa, qualche azione intrapresa. Lasciare il soggetto sarebbe peggio per tutti di quanto lo sarebbe per qualcuno lasciare perdere. Ma dato che il dilemma si applica ugualmente a tutti, è comunque possibile finire a discutere per sempre su cosa fare.

So how should you handle holy wars?

Allora come dovreste trattare le guerre sante?

La prima risposta è fate in modo che non succedano. Non è una cosa così senza speranza come sembra:

Potete anticipare alcune guerre sante standard: tendono a venire fuori sui linguaggi di programmazione, licenze (vedi sezione chiamata «La GPL e la compatibilità di Licenza» in Capitolo 9, *Licenze, Diritti d'Autore e Brevetti*), blocco dei reply-to (vedi sezione chiamata «Il grande dibattito sul 'Rispondi A'» in Capitolo 3, *L'Infrastruttura Tecnica*), e alcuni altri argomenti. Ogni progetto solitamente ha una sua guerra santa o due, con cui gli sviluppatori di lunga data diventeranno presto familiari. Le tecniche per fermare le guerre sante, o almeno limitarne i danni, sono sempre le stesse ovunque. Anche se siete convinti che il vostro schieramento abbia ragione, cercate di trovare *qualche* modo di esprimere simpatia e comprensione per gli argomenti che l'altro schieramento propone. Spesso il problema in una guerra santa è che poichè ogni schieramento ha costruito le proprie mura le più alte possibile, e reso chiero che ogni altra opinione è pura follia, l'atto di arrendersi o cambiare la propria idea diventa psicologicamente intollerabile: sarebbe l'ammissione non solo di aver sbagliato, ma di essere stati *certi* e comunque aver sbagliato. Il modo in cui potete rendere questa ammissione accettabile per l'altro schieramento è di esprimere voi stessi qualche dubbio—precisamente mostrando che capite gli argomenti che stanno facendo e trovarli almeno interessanti, se non alla fine convincenti. Fate un gesto che dia spazio per un gesto reciproco, e solitamente la situazione migliorerà. Non sarà più facile nè difficile raggiungere il risultato tecnico che volevate, ma almeno potete evitare inutili danni morali al morale del progetto.

Quando una guerra santa non può essere evitata, decidete presto quanto ci tenete, e poi vogliate pubblicamente lasciarla perdere. Quando fate così. potete dire che vi tirate indietro perchè non ne vale la pena, ma non esprimete amarezza e *non* usate l'occasione per un ultimo colpo agli argomenti dello schieramento opposto. Lasciare perdere è efficace solo quando fatto con grazia.

Le guerre sante sui linguaggi di programmazione sono un po' un caso speciale, perchè spesso sono altamente tecnici, e comunque molte persone si sentono qualificate a prenderne parte, e la posta è molto alta, dato che il risultato può determinare in quale linguaggio una buona parte del codice del progetto sarà scritta. La soluzione migliore è scegliere presto il linguaggio, con l'appoggio degli influenti sviluppatori iniziali, e poi difenderlo per il fatto che è quello per cui siete a vostro agio ad usarlo, *non* sul fatto che è meglio di qualche altro linguaggio che avrebbe invece potuto essere usato. Non lasciate mai degenerare la conversazione in un confronto accademico sui linguaggi di programmazione (che sembra accadere in particolare spesso quando qualcuno tira fuori Perl, per qualche ragione); è l'argomento mortale in cui dovete semplicemente rifiutarvi di farvi trascinare.

Per una maggiore conoscenza delle guerre sante, vedi http://catb.org/~esr/jargon/html/H/holy-wars.html, e l'articolo di Danny Cohen che rese popolare il termine, http://www.ietf.org/rfc/ien/ien137.txt.

L'Effetto "Minoranza Rumorosa"

In ogni discussione su mailing list, è facile per una piccola minoranza dare l'impressione che ci sia un grande problema di dissenso, inondando la mailing list con numerose lunghe email. E' un po' come una guerriglia, tranne il fatto che l'illusione di dissenso diffuso è persino più potente, perchè è divisa in un arbitrario numero di messaggi discreti e la maggior parte della gente non si preoccuperà di tenere traccia di chi ha detto cosa, quando. Avranno la vaga impressione che l'argomento è molto controverso, aspetteranno che la confusione finisca.

Il modo migliore per contrastare questo effetto è puntualizzare chiaramente e fornire prove a supporto di quanto piccolo sia il vero numero dei dissidenti, rispetto a quelli che sono d'accordo. Per incrementare la disparità, potreste voler chiedere privatamente alla gente che è stata quasi sempre zitta, ma che sospettate che sarebbe d'accordo con la maggioranza. Non dite nulla che suggerisca che i dissidenti volessero deliberatamente provare ad accrescere l'impressione che stavano dando. Ci sono possibilità che non lo facessero, e se anche lo avessero fatto, non c'è nessun vantaggio strategico nel puntualizzarlo. Tutto ciò di cui avete bisogno è mostrare i veri numeri in un confronto faccia a faccia, e la gente capirà che la loro percezione della situazione non corrisponde alla realtà.

Questo consiglio non vale solo per problemi con chiare posizioni pro e contro. Vale in ogni discussione dove c'è confusione, ma non è chiaro che la maggior parte della gente consideri il problema in discussione un vero problema. Dopo un po', se siete d'accordo che il problema non vale l'azione, potete vedere che ha fallito nell'ottenere seguito (anche se ha generato molte email), potete semplicemente osservare pubblicamente che non c'è seguito. Se l'effetto "minoranza rumorosa" ha lavorato, il vostro messaggio sembrerà come un respiro di aria fresca. L'impressione della maggior parte della gente della disussione fino a quel momento sarà stata in qualche modo confusa: "Huh, di sicuro sembra che ci sia qualche grosso problema qui, perchè ci sono molti messaggi, ma non riesco a vedere succedere nessun progresso". Spiegando come la forma della discussione la faccia apparire più turbolenta di quando sia davvero, gli date retrospettivamente una nuova forma, in cui la gente può rivedere la propria comprensione di cosa ne veniva fuori.

Gente Difficile

Non è più facile avere a che fare con gente difficile nei forum elettronici di quanto lo sia di persona. Per "difficile" non intendo "maleducata". La gente maleducata è fastidiosa, ma non necessariamente difficile. In questo libro si è già discusso di come trattarli: commentare la maleducazione la prima volta, e da allora in poi, o ignorarli o trattarli come chiunque altro. Se continuano ad essere maleducati, si renderanno di solito così impopolari da non avere influenza su altri nel progetto, quindi sono un problema che si circoscrive da sè.

I casi veramente difficile sono persone che non sono apertamente maleducate, ma che manipolano o abusano dei processi del progetto in un modo che finisce col costare tempo ed energia di altra gente, pur non portando alcun beneficio al progetto. Questa gente spesso cerca punti limite nelle procedure del progetto, per darsi più influenza di quella che altrimenti avrebbero. Questo è molto più insidioso della mera maleducazione, perchè nè il comportamento nè il danno che causa è evidente all'osservatore casuale. Un classico esempio è il guerrigliero, in cui qualcuno (sempre sembrando il più ragionevole possibile) continua a sostenere che il problema in discussione non è pronto per una soluzione, e propone molte possibili soluzioni, o nuovi punti di vista su vecchie soluzioni, quando cosa sta davvero succedendo è che capisce che un consenso o uno scontro sta per formarsi, e non gli piace dove il problema è andato a finire. Un altro esempio è quando c'è un dibattito che non convergerà ad un consenso, ma il gruppo cerca almeno di chiarificare i punti di disaccordo e produrre un riassunto per chiunque si aggiunga da quel momento in poi. L'ostruzionista, che sa che il riassunto potrebbe portare ad un risultato che non gli piace, spesso proverà a ritardare il sommario, complicando sempre di più le domande di cosa dovrebbe esserci, o obbiettando a consigli ragionevoli o introducendo nuovi e inaspettati punti.

Gestire la Gente Difficile

Per contrastare tale comportamento, aiuta capire la mentalità di coloro che lo adottano. Solitamente la gente non lo fa di proposito. Nessuno si sveglia al mattino e dice a se stesso: "Oggi cinicamente manipolerò i form procedurali per essere un irritante ostruzionista." Piuttosto, tali azioni sono spesso precedute da una sensazione semi-paranoica di essere tagliato fuori dalle interazioni e decisioni del gruppo. La persona sente che non sarà presa in considerazione seriamente, o (nei casi più gravi) che c'è quasi una cospirazione contro di lui—che gli altri membri del gruppo hanno deciso di formare un club esclusivo, di cui lui non è membro. Questo allora giustifica, nella sua mente, il prendere le regole alla lettera e procedere in una manipolazione formale delle procedure del progetto, per farsi prendere*seriamente* in considerazione da tutti gli altri. In casi estremi, la persona può persino pensare che sta combattendo una battaglia solitaria per salvare il progetto da se stesso.

E' la natura di questo tipo di attacco dall'interno che non tutti lo noteranno nello stesso momemento, e certa gente potrebbe non vederlo del tutto a meno che presentato con forte evidenza. Questo significa che neutralizzarlo potrebbe essere un bel po' di lavoro. Non è abbastanza persuadere voi stessi che sta succedendo; dovete trovare abbastanza prove anche per persuadere gli altri, e poi dovete far conoscere queste prove in modo intelligente.

Dato che è così tanto lavoro combattere, è spesso meglio tollerarlo giusto un po'. Pensatelo come una malattia da parassiti, ma leggera: se non è troppo debilitante, il progetto può permettersi di rimanere infetto, e le medicine avrebbero dolorosi effetti collaterali. Comunque, se tollerarla diventa troppo dannoso, allora è il momento di agire. Iniziate a prendere appunti sulle modalità che vedete. Fate in modo di includere riferimenti agli archivi pubblici—questa è una delle ragioni per cui il progetto registra le cose, così potete anche usarle. Una volta che avete costruito un buon caso, iniziate ad avere conversazioni private con altri partecipanti al progetto. Non dite loro cosa avete osservato, piuttosto, chiedete prima a loro cosa hanno osservato. Questa potrebbe essere la vostra ultima possibilità di avere un riscontro non filtrato di come gli altri vedono il comportamento di chi crea problemi; una volta che iniziate a parlarne apertamente, l'opinione diventerà polarizzata e nessuno sarà in grado di ricordare cosa avesse pensato in precedenza riguardo al problema.

Se le discussioni private indicano che almeno anche qualcun altro vede il problema, allora è il momento di fare qualcosa. Questo è quando dovete diventare *veramente* cauti, perchè è molto facile per questo tipo di gente cercare di far sembrare come se li steste criticando ingiustamente. Qualunque cosa facciate, non accusateli mai di abusare in modo malizioso delle procedure del progetto, di essere paranoici, o, in generale, di tutte le altre cose che sospettate siano probabilmente vere. La vostra strategia deve essere di sembrare sia più ragionevole e più concentrato con la salute generale del progetto, con l'obiettivo di o riformare il comportamento della persona, o farlo andare via in maniera definitiva. A seconda degli altri sviluppatori, e della vostra relazione con loro, potrebbe essere vantaggioso prima cercare alleati privatamente. O potrebbe non esserlo; potrebbe solo creare malumori dietro le quinte, se la gente pensa che stiate intraprendendo una impropria campagna silenziosa.

Ricordate che anche se l'altra persona potrebbe essere uno che si comporta in maniera distruttiva, *voi* sarete quelli che appaiono distruttivi se fate una pubblica accusa da cui non potete tornare indietro. Siate sicuri di avere molti esempi per dimostrare quello che state dicendo, e ditelo il più gentilmente possibile pur essendo diretti. Magari non persuaderete la persona in questione, ma va bene fino a quando persuadete tutti gli altri.

Caso di Studio

Ricordo solo una situazione, in più di 10 anni di lavoro nel software libero, dove le cose si fecero così cattiva che dovemmo chiedere tutti insieme a qualcuno di smettere di scrivere. Come spesso accade, non era maleducato, e sinceramente voleva solo essere d'aiuto. Solo non sapeva quando scrivere e quando non scrivere. Le nostre mailing list sono aperte al pubblico, e lui stava scrivendo così spesso, e

chiedendo domande su così tanti argomenti diversi, che stava diventando un problema di rumore per la comunità. Avevamo già provato a chiedergli gentilmente di fare un po' più di ricerca di risposte prima di scrivere, ma non aveva avuto effetto.

La strategia che alla fine funzionò è un perfetto esempio di come costruire un caso robusto su dati neutrali e in quantità. Uno dei nostri sviluppatori fece un po' di scavi negli archivi, e poi mandò il seguente messaggio privatamente a pochi sviluppatori. L'imputato (il terzo nome nella lista sotto, mostrato qui come "J. Random") aveva una storia molto breve nel progetto, e non aveva contribuito codice nè documentazione. E comunque era il terzo più attivo produttore di messaggi sulla mailing list:

```
From: "Brian W. Fitzpatrick" <fitz@collab.net>
To: [... lista dei destinatari omessa per riservatezza ...]
Subject: Il Lavandino dell'energia di Subversion
Date: Wed, 12 Nov 2003 23:37:47 -0600

Negli ultimi 25 giorni, i sei maggiori produttori di messaggi sulla mailing list s
[sviluppatori|utenti] sono stati:

    294  kfogel@collab.net
    236  "C. Michael Pilato" <cmpilato@collab.net>
    220  "J. Random" <jrandom@problematic-poster.com>
    176  Branko #ibej <brane@xbc.nu>
    130  Philip Martin <philip@codematters.co.uk>
    126  Ben Collins-Sussman <sussman@collab.net>

Vorrei dire che cinque di queste persone stanno contribuendo a Subversion, che
raggiungerà 1.0 nel prossimo futuro.

Vorrei anche dire che una di queste persone sta consumando in maniera consistente
e l'energia degli altri 5, per non dire della mailing list intera,
quindi (magari non intenzionalmente) rallentando lo sviluppo di Subversion. Non ho
fatto un'analisi di tutti i thread, ma facendo il vgrep delle mie mail di Subversi
che ogni mail di persona ha ricevuto risposta almeno una volta da almeno 2 delle a
della lista sopra.

Penso che qui sia necessario qualche tipo di intervento radicale, anche se faremo
via la persona sopracitata. Carinerie e gentilezze si sono già dimostrate senza ef

dev@subversion è una mailing list per facilitare lo sviluppo di un sistema
di controllo di versione, non una sessione di terapia di gruppo.

-Fitz, che prova a guadare attraverso tre giorni di email svn che ha lasciato accu
```

Anche se potrebbe non sembrare così a prima vista, il comportamento di J.Random era un classico esempio di abuso delle procedure di progetto. Non stava facendo nulla di ovvio come provare a sabotare un voto, ma stava abusando della politica della mailing list di affidarsi sulla auto moderazione dei suoi membri. Abbiamo lasciato al giudizio di ogni individuo quando scrivere messaggi e su quali argomenti. Quindi, non avevamo procedure di ricorso per gestire qualcuno che o non aveva, o non usava, tale giudizio. Non c'era alcuna regola a cui riferirsi e dire che il tizio la stava violando, eppur tutti sapevano che il suo frequente scrivere messaggi stava diventando un problema serio.

La strategia di Fitz era, in retrospettiva, da maestro. Ha trovato un esempio dannatamente fondato, ma poi l'ha distribuito discretamente, mandandolo prima alle poche person il cui supporto sarebbe stato la chiave per ogni azione drastica. Si sono trovati d'accordo che qualche tipo di azione era necessaria,

e alla fine abbiamo chiamato J. Random al telefono, descritto il problema a lui direttamente, e gli abbiamo chiesto semplicemente di smetterla di scrivere messaggi. Lui non ha mai veramente capito il perchè; se fosse stato in grado di capire, probabilmente avrebbe fin dall'inizio usato un criterio adeguato. Ma accettò di smettere di scrivere, e la mailing list tornò ad essere utilizzabile. Parte del motivo per cui questa strategia ha funzionato, magari, è stata l'implicita minaccia che avremmo potuto iniziare a limitare i suoi messaggi usando il software di moderazione solitamente usato per prevenire lo spam (vedi sezione chiamata «Prevenire lo spam» in Capitolo 3, *L'Infrastruttura Tecnica*). Ma la ragione per cui eravamo in grado di avere questa opzione di riserva è stata il fatto che Fitz ha in primo luogo trovato il necessario supporto nelle persone importanti.

Gestire la Crescita

Il prezzo della crescita è pesante nel mondo dell'open source. Come il vostro software diventa più popolare, il numero di persone che compaiono alla ricerca di informazioni cresce in modo sensazionale, mentre il numero delle persone in grado di dare le informazioni cresce molto meno lentamente. Inoltre, anche se il rapporto fosse uniformemente bilanciato, ci sarebbe tuttavia un problema fondamentale di scalabilità col modo in cui la maggior parte dei progetti open source gestiscono le comunicazioni. Considerate le mailing list, per esempio. La maggior parte dei progetti hanno una mailing list per le domande degli utilizzatori generici—a volte il nome delle mailing list è “utilizzatori”, “discutere”, “aiuto”, o qualcos'altro. Qualunque sia il nome, il proposito della mailing list è sempre lo stesso: fornire un posto dove la gente possa ricevere risposta alle sue domande, dove gli altri osservano e (presumibilmente) assorbono conoscenze dall'osservazione di questi scambi.

Queste mailing list funzionano molto bene fino a poche migliaia di utilizzatori e/o fino a un paio di centinaia di post al giorno. Ma circa dopo di ciò il sistema incomincia a collassare, perché ogni iscritto vede ogni post. Se il numero dei post alla mailing list incomincia a superare quello che ogni singolo lettore può elaborare in un giorno, la mailing list diventa un carico per i suoi membri. Immaginate, per esempio, che Microsoft abbia una tale mailing list per Windows XP. Windows XP ha centinaia di milioni di utenti; se anche un decimo dell'1% ha domande in un periodo di ventiquattrore, allora questa ipotetica mailing list riceverebbe centinaia di migliaia di post al giorno! Una tale mailing list non potrebbe mai esistere, perché nessuno vorrebbe rimanere iscritto ad essa. Questo problema non è limitato alle mailing list; la stessa logica si applica ai canali IRC, ai forum di discussioni online, indubbiamente ad ogni sistema in cui un gruppo ascolta domande dagli individui. Le implicazione sono sinistre: l'usuale modello open source di supporto in parallelo di massa non si adegua ai livelli necessari per la dominazione del mondo.

Non ci sarà nessuna esplosione quando il forum raggiungerà il punto di rottura. Ci sarà solo un silenzioso effetto di reazione negativa: la gente si cancella l'iscrizione dalle mailing lists, o lascia il canale IRC, o in ogni caso smette di infastidire facendo domande, perché possono vedere che non saranno ascoltati in tutto il rumore. Nella misura in cui sempre più la gente fa questa scelta altamente razionale, l'attività dei forum sembra restare in un livello manovrabile. Ma esso rimane in un livello manovrabile precisamente perché la gente razionale (o almeno con esperienza) ha incominciato a guardare altrove per le informazioni—mentre la gente inesperta rimane dentro e posta continuamente. In altre parole, l'effetto a senso unico di continuare a usare modelli di comunicazioni non ampliabili quando il progetto cresce è quello che la qualità delle domande e delle risposte tende a scendere, il che fa sembrare che i nuovi utilizzatori sono più muti di quanto erano soliti essere, mentre nei fatti probabilmente non lo sono. E' solo che il rapporto beneficio/costo dell'usare questi forums ad alta popolazione scende, così naturalmente quelli con esperienza incominciano per primi a guardare altrove per le risposte. Adattare il meccanismo delle comunicazioni in modo che faccia fronte alla crescita del progetto quindi comporta due strategie:

1. Riconoscere quando parti particolari di un forum *non* stanno soffrendo la crescita illimitata, anche se il forum nella sua interezza la sta soffrendo, e separare quelle parti per creare dei nuove forum specializzati (cioè non permettete che il buono sia trascinato in basso dal cattivo).

2. Assicurarsi che ci siano fonti di informazione automatizzate disponibili, e che siano mantenute organizzate, aggiornate, e facili da trovare.

La strategia (1) non è difficile di solito. La maggior parte dei progetti partono con un solo forum principale: una mailing list di discussioni generali, nella quale possono essere discussi idee di funzionalità, questioni di progettazione, e problemi di codice. Chiunque sia coinvolto nel progetto è nella mailing list. Dopo un po', di solito diventa chiaro che la mailing list si è evoluta in varie sotto mailing list distinte basate sull'argomento. Per esempio, alcune discussioni sono chiaramente sulla progettazione e sullo sviluppo; altre sono domande degli utilizzatori della varietà “Come faccio X”; può darsi che ci sia una terza famiglia di argomenti centrati sull'elaborazione dei report di bug e su richieste di accrescimento; e così via. Un dato individuo, certo, potrebbe partecipare a molti differenti tipi di discussioni, ma la cosa importante è che non ci sia una grande quantità di sovrapposizioni fra i tipi stessi. Essi potrebbero essere suddivisi in mailing list separate senza causare una pericolosa balcanizzazione, perché le discussioni raramente attraversano i limiti dell'argomento.

In realtà fare queste divisioni è un processo in due tempi. Voi create la nuova mailing list (o il canale IRC, o qualunque altra cosa sia) e poi spendete quanto tempo sia necessario nel rimproverare e nel ricordare alla gente *di usare* appropriatamente i nuovi forum. Il secondo passo può durare settimane, ma alla fine la gente si farà l'idea. Voi dovete semplicemente considerare importante dirlo a chi invia che il post è stato inviato alla destinazione sbagliata, e farlo in modo visibile, in modo che gli altri siano incoraggiati ad essere di aiuto nell'instradamento. E' anche utile avere una pagina web che fornisca una guida a tutte le mailing list disponibili; la vostra risposta può far riferimento a quella pagina web e, come premio, il destinatario può imparare qualcosa sul cercare nelle linee guida prima di postare.

La strategia (2) è un processo continuo che dura il tempo di vita del progetto e coinvolge molti partecipanti. Certo è in parte questione di avere la documentazione aggiornata (vedere sezione chiamata «La documentazione» in Capitolo 2, *Partenza*) e assicurarsi di indirizzare la gente lì. Ma è anche molto più di questo; le sezioni che seguono discuteranno questa strategia in dettaglio.

Uso Ben Visibile degli Archivi

Tipicamente tutte le comunicazioni in un progetto open source (eccetto talvolta le conversazioni IRC) vengono archiviate. Gli archivi sono pubblici e vi si possono fare ricerche, e hanno una stabilità informativa: cioè, una volta che un dato pezzo di informazione è registrato in un particolare indirizzo, rimane in quell'indirizzo per sempre.

Usate questi archivi quanto più è possibile, e quanto più in modo visibile possibile. Anche quando sapete la risposta spontanea a qualche domanda, se pensate che c'è un riferimento nell'archivio che contiene la risposta, spendete tempo a riportarla alla luce e fornitela. Ogni volta che fate ciò in modo visibile, qualche persona imparerà che ci sono gli archivi lì, e che cercare in essi può produrre risposte. Anche, riferendovi agli archivi invece di riscrivere il consiglio, rafforzate una norma sociale contro la duplicazione delle informazioni. Perché avere la stessa risposta in due posti differenti? Quando il numero di posti in cui essa può essere trovata è tenuto al minimo, le persone che le hanno trovate prima è molto probabile che ricordino cosa cercare per trovale di nuovo. Riferimenti ben collocati, possono anche contribuire alla qualità dei risultati della ricerca in generale, perchè essi rafforzano il ranking della risorsa obiettivo nei motori di ricerca di Internet.

Ci sono volte in cui la duplicazione delle informazioni ha senso, comunque. Per esempio, supponete che ci sia già una risposta negli archivi, non vostra, che dice:

```
Pare che in vostri indici di Scanley si siano imbrogliati. Per ripararli fate ques

1. Spegnete il server di Scanley.
```

```
2. Fate girare il programma 'sbroglia' che si carica con Scanley.
3. Avviate il server.
```

Allora, mesi dopo, vedete un altro post che indica che gli indici di qualcuno si sono imbrogliati. Cercate negli archivi e vien fuori la vecchia risposta di sopra, ma vi rendete conto che sono mancanti alcuni passi (forse per errore, o perché il software è cambiato da quando quel post fu scritto). Il modo classico di gestire ciò è postare un nuovo, più completo set di istruzioni, e rendere obsoleto il vecchio post menzionandolo:

```
Pare che in vostri indici di Scanley si siano imbrogliati. Vedemmo questo problema

1. Spegnete il server di Scanley.
2. Diventate l'utilizzatore col quale il server di Scanley gira.
3. Come tale utilizzatore fate girare il programma 'sbroglia' per gli indici.
4. Fate girare Scanley a mano per vedere se gli indici funzionano ora.
5. Riavviate il server.
```

(In un mondo ideale, sarebbe possibile attaccare una nota al vecchio post, che dica che c'è una informazione più fresca e che punti al nuovo post. Comunque non so di un nuovo software per l'archiviazione che offra una funzionalità “obsoleto per”, forse perchè sarebbe leggermente complessa da implementarsi in un modo che non violi l'integrità dell'archivio in quanto registrazione parola per parola. Questa è un'altra ragione perché creare pagine dedicate con risposte alle domande comuni è una buona idea).

Negli archivi probabilmente si ricerca di più per risposte a domande tecniche, ma la loro importanza per i progetto va ben al di là di questo. Se le linee guida formali di un progetto sono la sua legge statutaria, gli archivi sono la sua legge comune: una registrazione di tutte le decisioni prese e di come sono arrivate là. In ogni discussione ricorrente è quasi obbligatorio oggigiorno partire con una ricerca nell'archivio. Questo ti permette di incominciare una discussione con un sommario dello stato corrente delle cose, di anticipare obiezioni, di preparare i rifiuti, e possibilmente di scoprire degli angoli a cui non avevate pensato. Anche, gli altri partecipanti si *aspetteranno* che abbiate fatto una ricerca nell'archivio. Anche se le discussioni precedenti non andarono da nessuna parte, voi dovreste includere dei puntatori ad esse quando balza si di nuovo l'argomento (così la gente può guardare da sé) che non a) andarono da nessuna parte e probabilmente diranno ora ciò che non è stato detto prima b) che voi avete fatto i vostri compiti, e quindi state probabilmente dicendo cose che non sono state dette prima.

Trattate Tutte le Risorse Come un Archivio

Tutti i precedenti consigli si applicano a più che ai soli archivi delle mailing lists. L'avere particolari pezzi di informazione in stabili indirizzi che si possono trovare convenientemente dovrebbe essere un principio organizzativo di tutte le informazioni del progetto. Lasciatemi portare le FAQ del progetto come caso di studio.

Come la gente usa le FAQ?

1. Essi vogliono cercare in esse per parola e frase.

2. Essi vogliono guardarle per assorbire informazioni senza necessariamente guardare per risposte a domande specifiche.

3. Essi si aspettano dei motori di ricerca come Google per conoscere il contenuto delle FAQ, così le ricerche possono dar luogo a nuove voci nelle FAQ.

4. Essi vogliono essere in grado condurre altre persone direttamente a specifiche voci nelle FAQ.

5. Essi vogliono essere capaci di aggiungere materiale nuovo alle FAQ, ma notate che questo avviene molto meno spesso di quanto siano cercate le risposte—Le FAQ sono molto spesso più lette che scritte.

Il punto 1 implica che le FAQ dovrebbero essere disponibili in una sorta di formato testuale. I punti 2 e 3 implicano che le FAQ dovrebbero essere disponibili in una pagina Html, con il punto 2 che indica addizionalmente che l'Html dovrebbe essere creato per la leggibilità (cioè, vorrete qualche controllo sul loro aspetto e percezione), e dovrebbero avere una tavola dei contenuti. Il punto 4 significa che ad ogni nuova voce delle FAQ dovrebbe essere assegnata una *ancora con nome*, un tag che permette alla gente di raggiungere una particolare locazione nella pagina. Il punto 5 significa che i file sorgenti delle FAQ dovrebbero essere disponibili in un modo adatto (vedere sezione chiamata «Tenere tutto sotto controllo di versione» in Capitolo 3, *L'Infrastruttura Tecnica*), in un formato che sia facile da editare.

Ancore con nome e attributi ID

Ci sono due modi per ottenere che il browser salti ad un determinata locazione: le ancore con nome e gli attributi id.

Una *ancora con nome* è appunto un normale elemento Html (`<a>...</a>`), ma con un attributo “name”:

```
<un name=”miaetichetta”>...</a>
```

Versioni più recenti di Html supportano un attributo *id generico*, che può essere attaccato ad un elemento Html, non solo `<a>`. Per esempio:

```
<p id=”miaetichetta”>...</p>
```

Ambedue gli attributi id e le ancore con nome sono usati nello stesso modo. Uno aggiunge un segno hash e l'etichetta a un URL per far si che il browser salti dritto a quel punto nella pagina:

```
http://myproject.example.com/faq.html#miaetichetta
```

Virtualmente tutti i browsers supportano le ancore con nome; la maggior parte dei browsersa moderni supportano l'attributo id. Per farlo funzionare con sicurezza io raccomanderei di usare sia le ancore con nome da sole *sia* gli attributi id insieme (con la stessa etichetta per ambedue in una data coppia, certo). Le ancore con nome non possono essere a chiusura automatica, anche se non c'è testo dentro l'elemento, voi lo potete scrivere in una forma di due tipi:

```
<a name="mia etichetta"></a>
```

...sebbene normalmente ci sarebbe lì qualche testo, come il titolo di una sezione.

Se usate l'ancora con nome o l'attributo id, o ambedue, ricordate che l'etichetta non sarà visibile a qualcuno che osserva verso quella locazione senza usare l'etichetta. Ma una tale persona potrebbe voler scoprire l'etichetta di una locazione particolare, in modo che possa inviare per email l'URL di una risposta delle FAQ a un amico, per esempio. Per aiutarlo a fare questo aggiungete un *attributo title* allo stesso elemento(i) a cui aggiungeste il “nome” e/o l'attributo id, per esempio:

```
<a name=”miaetichetta” title=”miaetichetta”>...</a>
```

Quando il puntatore del mouse viene mantenuto sopra il testo all'interno dell'elemento con l'attributo id, la maggior parte dei browsers esporranno un piccolo box di popup che mostra il titolo, Io di solito includo il segno hash, a ricordare all'utilizzatore che ciò è quello che metterebbe alla fine dell'URL per saltare dritto a q uesta locazione la prossima volta.

Formattare così le FAQ è solo un esempio di come rendere presentabile una risorsa. La stesse qualità la ricerca diretta, la disponibilità nei principali motori di ricerca di Internet, la facilità di consultazione,

la stabilità referenziale, a e (dove applicabile) l'editabilità—si applicano si applicano ad altre pagine web, all'albero del codice sorgente, al tracciatore di bug, ecc.. Appunto avviene che la maggior parte delle mailing list che archiviavano il software molto tempo fa si resero conto dell'importanza di queste qualità, che il motivo per cui le mailing list tendono ad avere queste funzionalità alla nascita, mentre altri formati possono richiedere qualche sforzo extra da parte di quelli che hanno la manutenzione (Capitolo 8, *Gestire i Volontari* discute come suddividere il carico della manutenzione fra molti volontari).

La Tradizione della Codifica

Nella misura in cui un progetto acquista anzianità e complessità, la quantità di dati che ogni partecipante che arriva deve assorbire cresce. Coloro che sono stati col progetto per lungo tempo saranno in grado di imparare, e inventare, le convenzioni del progetto nella misura in cui andarono avanti. Essi spesso non saranno consapevolmente al corrente di quale enorme corpo di tradizione hanno accumulato, e possono essere sorpresi di fronte a quanti passi falsi sembrano fare i nuovi arrivati. Certo, il fatto non è che i nuovi arrivati siano di qualche qualità inferiore di prima; è che essi sono di fronte a un più grande carico di acculturazione rispetto ai nuovi arrivati del passato.

L'anzianità che il progetto accumula è tanta nel comunicare e nel preservare le informazioni quanta essi ne hanno negli standard del codice e altre minuterie tecniche. Noi abbiamo dato un'occhiata a tutti e due i tipi di standard in sezione chiamata «La documentazione sviluppatore» in Capitolo 2, *Partenza* e sezione chiamata «Metter Giù Tutto Per Iscritto» in Capitolo 4, *L'Infrastruttura Sociale e Politica* rispettivamente e gli esempi sono dati lì. Ciò di cui tratta questa sezione è come mantenere le linee guida aggiornate nella misura in cui il progetto si evolve, specialmente le linee guida su come sono trattate le comunicazioni, perché queste sono le uniche che cambiano al massimo grado nella misura in cui il progetto cresce in grandezza e complessità.

Primo, prestate attenzione ai motivi per i quali la gente si confonde. Se vedete la stessa situazione presentarsi ancora e ancora, specialmente con i nuovi partecipanti. La possibilità che c'è è è una linea guida che necessita di di essere documentata e non lo è. Secondo, non vi stancate di dire la stessa cosa ancora e ancora, e non suonate come se siete stanchi di dirle. Voi e i veterani di altri progetti dovete ripeterlo a voi stessi; questo è un effetto inevitabile dell'arrivo di nuovi arrivati.

Ogni pagina web, ogni messaggio della mailing list, ed ogni canale IRC dovrebbe essere considerato uno spazio per i consigli, non per pubblicità, eccetto che per pubblicità delle risorse del vostro progetto. Ciò che mettete in quello spazio dipende dalla demografia di quelli che lo leggono. Un canale IRC per le domande degli utilizzatori, è, per esempio, adatto a portare gente che che non ha mai interagito col progetto prima—spesso qualcuno che ha appena installato il software e ha una domanda a cui vorrebbe venga risposto immediatamente (dopotutto, se potesse aspettare, l'avrebbe mandata alla mailing list, che probabilmente meno meno del suo tempo totale, sebbene ci vorrebbe di più perché ne venga indietrouna risposta). La gente di solito non fa un investimento permanente nel canale IRC; essi si presentano, fanno la loro domanda, e vanno via.

Quindi l'argomento del canale dovrebbe essere diretto a persone che cercano risposte tecniche sul software *adesso*, piuttosto che a, diciamo, a persone che potrebbero essere coinvolte nel progetto nel lungo termine e per i quali le linee guida di interazione della comunità potrebbero essere più appropriate. Qui è come un canale occupato gestisce ciò (comparate questo col precedente esempio in sezione chiamata «IRC / Sistemi di Chat in tempo reale» in Capitolo 3, *L'Infrastruttura Tecnica*):

```
State parlando su #linuxhelp

L'argomento per #linuxhelp è Prego LEGGERE
http://www.catb.org/~esr/faqs/smart-questions.html &&
http://www.tldp.org/docs.html#howto PRIMA di fare domande | Le regole del canale s
```

```
http://kerneltrap.org/node/view/799 prima di chiedere di aggiornare al kernel 2.6.
aggiornare a 2.6.8.1 o 2.4.27 | in certa misura disastro hash: http://tinyurl.com/
| reiser4 fuori
```

Con le mailing list lo “spazio pubblicitario” è una piccola è un piccolo spazio in basso attaccato ad ogni messaggio. La maggior parte dei progetti vi mettono lì le istruzioni iscrizione/deiscrizione, e magari un puntatore alla homepage oppure alla pagina delle FAQ. Voi potreste pensare che chi si è iscritto alla mailing list saprebbe dove trovare queste cose ed essi probabilmente lo sanno—ma molta più gente di quelli che si sono iscritti vedono questi messaggi della mailing list. Un post archiviato può essere linkato da molti posti, alcuni post diventano cosi largamente noti che alla fine hanno molti più lettori al di fuori della mailing list che dentro.

La formattazione può fare molta differenza. Per esempio, nel progetto di Subversion, noi stavamo avendo un limitato esito favorevole nell'usare il filtro dei bug descritto in sezione chiamata «Pre-Filtraggio del Bug Tracker» in Capitolo 3, *L'Infrastruttura Tecnica* . Molti rapporti di bug fasulli stavano venendo archiviati da gente inesperta, e ogni volta che ciò avveniva, l'archiviatore doveva essere educato esattamente nello stesso modo di 500 persone prima di lui. Un giorno, dopo che uno dei nostri sviluppatori era finalmente arrivato alla fine della sua cordata e aveva inveito contro qualche utilizzatore deficitario che non leggeva le linee guida del tracciatore di problemi abbastanza con cura, un altro sviluppatore decise che questo comportamento era andato avanti a lungo abbastanza. Egli suggerì che noi formattassimo la pagina frontale del tracciatore di bug in modo che la parte più importante, la ingiunzione a discutere il bug sulla mailing list o sul canale IRC prima di archiviarlo, si distinguesse per l'enormità, lettere in grassetto rosso, su uno sfondo giallo brillante, centrato in prominenza sopra ogni altra cosa nella pagina. Noi facemmo così (potete vederne i risultati a http://subversion.tigris.org/project_issues.html), e il risultato fu un salto notevole nella velocità dei problemi fasulli archiviati. Ancora li prendiamo, certo, noi li prenderemo sempre ma la velocità è diminuita considerevolmente, anche se il numero degli utilizzatori cresce. La conclusione è non solo che il database contiene meno spazzatura, ma che quelli che rispondono alla archiviazione dei problemi rimangono di umore migliore e sono più propensi a restare amichevoli quando rispondono ad una archiviazione di una delle adesso rare archiviazioni fasulle. Questo migliora sia l'immagine del progetto, sia la salute mentale dei suoi volontari.

La lezione per noi fu che scrivere solamente le linee guida non era abbastanza. Noi dovevamo metterle dove sarebbero state viste da coloro che più di tutti avevano bisogno di esse, e formattarle in modo tale che il loro stato come materiale di introduzione sarebbe stato immediatamente chiaro alle persone non familiari col progetto.

Le pagine statiche non sono il solo luogo per far pubblicità ai clienti del progetto. E' anche richiesta una certa quantità di politica interattiva (nel senso di “ricordare amichevolmente” non nel senso di ammanettare e mettere in prigione). Tutta la revisione paritaria, anche le revisioni degli invii descritte in sezione chiamata «Praticare una Visibile Revisione del Codice» in Capitolo 2, *Partenza*, dovrebbe includere la revisione della conformità o non conformità della gente alle norme del progetto, specialmente riguardo alle convenzioni delle comunicazioni.

Un altro esempio dal progetto di Subversion: noi stabilimmo che “r12908" significasse "revisione 12908” nel deposito del controllo di versione. Il prefisso minuscolo “r” è facile da battere, e poiché è la metà dell'altezza delle cifre, esso rende un blocco di testo facilmente riconoscibile quando combinato con le cifre. Certo, quando una email di invio arriva con un messaggio di log come questo:

```
------------------------------------------------------------------------
r12908 | qsimon | 2005-02-02 14:15:06 -0600 (Mer, 02 Feb 2005) | 4 righe

Patch dal collaboratore J. Casuale <jrcontrib@gmail.com>
```

```
* trunk/contrib/client-side/psvn/psvn.el:
  Corretti alcuni  errori di stampa dalla revisione 12828.
------------------------------------------------------------------------
```

...parte della revisione di questo invio è per dire “Strada facendo prego usate 'r12828', non 'revisione 12828' quando vi riferite al cambiamento passato”. Questa non è pedanteria; è altrettanto importante per l'analisi sintattica automatica quanto per i lettori umani.

Seguendo il principio generale che ci dovrebbero essere dei metodi di riferimento canonico e che questi metodi di riferimento dovrebbero essere usati coerentemente ovunque, il progetto in effetti esporta certi standards. Questi standards mettono la gente in grado di scrivere strumenti che presentino le comunicazioni del progetto in modi più usabili—per esempio un revisione formattata come "r12828" potrebbe essere trasformata in un link vivo al sistema di osservazione del deposito. Ciò sarebbe piuttosto difficile se la revisione fosse scritta "revisione 12828", sia perché quella forma potrebbe essere divisa da una interruzione di linea, sia perché è meno distinta (la parola “revisione” apparirà spesso da sola, e il gruppo dei numeri apparirà spesso da solo, mentre la combinazione "r12828" può significare solo un numeri di revisione. Simili preoccupazioni si applicano ai numeri di problema, voci di FAQ (suggerimento: usate un URL con un'ancora con nome, come descritto in Ancore con nome e attributi ID), ecc.

Anche per le entità dove non c'è una ovvia breve, forma canonica, la gente tuttavia dovrebbe essere incoraggiata a fornire pezzi chiave di informazione coerentemente. Per esempio, quando ci si riferisce ad un messaggio della mailing list, non date solo il mittente e i soggetto; date anche l'URL dell'archivio *e* e la testata Message ID. L'ultimo permette alla gente che ha la sua copia della mailing list (le gente a volte tiene copie offline, per esempio da usare su un laptop in viaggio) per identificare senza ambiguità i messaggio giusto anche se non ha l'accesso agli archivi. Il mittente e il soggetto non sarebbero sufficienti, per che la stessa persona potrebbe fare parecchi post nello stesso trhead, anche nello stesso giorno.

Più il progetto cresce, più importante diventa questo tipo di coerenza. Coerenza significa che qualunque persona guardi, essa vede seguiti gli stessi comportamenti, così essi sanno seguire i comportamenti stessi. Questo, successivamente, riduce il numero di domande che essi hanno bisogno di di fare. Il carico di avere milioni di lettori non è più grande di quello di averne uno; i problemi di scalabilità cominciano a sorgere quando un a certe percentuale di di quei lettori fa domande. Nella misura in cui il progetto cresce, esso deve ridurre quella percentuale aumentando la densità e l'accessibilità dell'informazione, in modo che la persona in grado di trovare ciò di cui ha bisogno senza dover chiedere.

Nessuna Conversazione nel Tracciatore di Bug

In un progetto che sta facendo un uso attivo del tracciatore di bug, c'è sempre un pericolo che il tracciatore si trasformi esso stesso in un forum di discussione, anche se la mailing list sarebbe in realtà migliore. Di solito si incomincia abbastanza innocentemente: qualcuno annota un problema con una, diciamo, soluzione proposta, e fa seguire un altra annotazione che indica problemi. La prima persona risponde, ancora aggiungendosi al problema...e va così.

Il problema con questo è, primo, che il tracciatore di bug è un luogo piuttosto ingombrante per tenervi una discussione, e secondo, che altre persone possono non farvi attenzione—dopotutto essi si aspettano che che la discussione dello sviluppo avvengano sulla mailing list dello sviluppo, così è lì che guardano per essa. Essi possono non essere per nulla iscritti alla mailing list dei cambiamenti dei problemi, e anche se lo sono, possono non seguirla molto da vicino.

Ma esattamente dove nel processo è andata sbagliata qualcosa? Fu quando la persona d'origine aggiunse la sua soluzione al problema—dovrebbe aver postato nella mailing list invece? O fu quando la seconda persona rispose nel problema, invece che sulla mailing list?

Non esiste una risposta giusta, ma c'è un principio generale: se state appunto aggiungendo dati a un problema, allora fatelo nel tracciatore, ma se state incominciando una *conversazione*, allora fatelo nella mailing list. Potete non essere sempre in grado di dire quale è il caso, ma usate appunto il miglior giudizio. Per esempio, quando state aggiungendo una patch con una soluzione potenzialmente controversa, potreste essere in grado di anticipare che la gente sta per avere una domanda su di essa. Così anche se normalmente aggiungereste la patch al problema (ipotizzando che non volete o non potete fare l'invio del cambiamento direttamente), in questo caso potreste invece scegliere di postarla alla mailing list. In ogni caso, alla fine lì verrà il momento nello scambio in cui una parte o l'altra può dire che è sul punto di passare dalla sola aggiunta di dati a una reale conversazione—nell'esempio che incominciò questa sezione, che sarebbe la seconda persona che risponde, colui che si rendeva conto che c'erano problemi con la patch, poté predire che stava per seguire una conversazione reale, e che quindi avrebbe dovuto essere tenuta sul mezzo appropriato.

Per usare una analogia matematica, se sembra che l'informazione sarà rapidamente convergente, allora mettetela direttamente nel tracciatore di bug; se sembra che sarà divergente allora una mailing list o un canale IRC può essere un posto migliore.

Ciò non significa che non ci dovrebbe mai essere scambio nel tracciatore di bug. Chiedere maggiori dettagli per la ricetta di riproduzione da chi ha fatto il report all'origine tende ad essere un processo altamente convergente, per esempio. E' improbabile che la risposta della persona sollevi nuovi problemi; è semplicemente fornire maggiori dettagli sull'informazione già archiviata. Non c'è bisogno di distrarre la mailing list con quel procedimento. Abbiate cura con ogni mezzo di ciò con una serie di commenti nel tracciatore. Allo stesso modo, se siete abbastanza sicuri che il bug è stato riportato male (cioè, non è un bug), allora potete semplicemente dirlo così bene nel problema. Anche indicare un problema minore con una soluzione proposta è bene, nell'ipotesi che il problema non sia un pezzo di una rappresentazione che suscita applausi per la risoluzione completa.

D'altra parte, se state sollevando dei problemi filosofici sulla portata del bug o sull'appropriato comportamento del software, potete essere abbastanza sicuri che gli altri sviluppatori vorranno essere coinvolti. Sembra che la discussione diverga per un momento prima di convergere, così tenetela nella mailing list.

Linkate sempre all'argomento della mailing list dal problema, quando scegliete di postare alla mailing list. E anche importante per qualcuno che sta seguendo il problema essere capace di raggiungere la discussione. La persona che inizia il thread può trovare ciò laborioso, ma l'open source è fondamentalmente una responsabilità di chi scrive. E' molto più importante rendere facili le cose per le decine di centinaia di persone che possono leggere il bug, che per le tre o cinque persone che scrivono intorno ad esso.

E' bene trarre importanti conclusioni o sommari dalla discussione sulla mailing list e incollarle nel problema, se ciò renderà le cose convenienti per i lettori. Deve iniziare una discussione sulla mailing list un comune idioma, mettete un link al thread nel problema, e poi quando la discussione finisce, incollate il sommario finale nel problema (insieme con un link al messaggio contenente il sommario), così chi osserva il problema possa vedere quale conclusione sia stata raggiunta senza dover cliccare da qualche altra parte. Notate che di solito il problema della duplicazione dei dati da ”due capi” non esiste qui, perché ambedue gli archivi e i commenti al problema di solito sono statici, dati che non è possibile cambiare in nessun modo.

La Pubblicità

Nel software libero c'è una discreta regolare continuità tra le discussioni puramente interne e le regole delle pubbliche relazioni. Ciò avviene in parte perchè il pubblico di destinazione è mal definito. Dato che la maggioranza o tutti i post sono pubblicamente accessibili, il progetto non ha il controllo pieno sull'impressione che ne ha il mondo. Qualcuno,— diciamo, un editor slashdot.org [http://slashdot.org/]

—può attrarre l'attenzione dei lettori verso un post che nessuno si sarebbe mai aspettato che sarebbe stato visto dall'esterno del progetto. Questo è un fatto concreto con la quale convivono tutti i progetti open source, ma in pratica, il rischio è piccolo. In generale gli annunci che più il progetto vuole che siano pubblicizzati sono quelli che saranno più pubblicizzati, nell'ipotesi che usiate i meccanismi giusti per indicare la rilevanza di una notizia al mondo esterno.

Per gli annunci principali tendono ad esserci quattro o cinque principali canali di distribuzione, sui quali gli annunci dovrebbero essere fatti quanto più simultaneamente possibile:

1. La pagina principale del vostro progetto è vista probabilmente da più gente che qualsiasi altra parte del progetto. Se avete annunci veramente importanti mettete lì una breve inserzione. La breve inserzione dovrebbe essere un piccolo specchietto che linki al comunicato stampa (vedere sotto) per maggiori informazioni.

2. Allo stesso tempo, voi dovreste avere una area “Notizie” o ”Comunicati stampa” sul sito, dove un annuncio possa essere scritto nei dettagli. Parte del proposito di un comunicato stampa e quella di fornire un solo canonico “oggetto annuncio”a cui altri siti possano linkare, in modo da assicurarsi che esso sia strutturato di conseguenza: sia come pagina web per le release, sia come nuova entrata nel blog, sia come altro tipo di entità che possa essere linkata pur essendo tuttavia tenuta distinta da altri comunicati stampa nella stessa area.

3. Se il vostro progetto ha un feed RSS, assicuratevi che l'annuncio vada anche lì. Ciò può avvenire automaticamente quando create il comunicato stampa, a seconda di come le cose sono messe sul vostro sito. (*RSS* è un meccanismo per distribuire sommari di meta dati ricchi agli “iscritti”, cioè gente che ha indicato un interesse nel ricevere questi sommari. Vedere per maggiori informazioni sugli RSS. Se l'annuncio riguarda una nuova release del software, allora aggiornate la voce del vostro progetto su (vedere http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html per maggiori informazioni sugli RSS.)

4. Se l'annuncio riguarda una nuova release del software, allora aggiornate la voce del vostro progetto su http://freshmeat.net/ (vedere su come creare la voce in primo luogo). Ogni volta che aggiornate una voce di Freshmet, quella voce va sulla change list per il giorno. La change list non è aggiornata solo sullo stesso Freshmet, ma sui vari portali (incluso http://slashdot.org) che sono osservati ansiosamente da orde di gente. Freshmet offre gli stessi dati via feed RSS, così la gente che non è iscritta al suo feed RSS del vostro progetto può ancora vedere l'annuncio attraverso quelli di Freshmet.

5. Mandate una email alla mailing list degli annunci del progetto. Il nome di questa mailing list dovrebbe essere veramente “annuncia”, ciòè, `annuncia@yourprojectdomain.org`, perché questa è una convenzione piuttosto standard ora e lo statuto della mailing list dovrebbe render chiaro che è a traffico molto lento riservata agli annunci principali del progetto. La maggior parte di questi annunci saranno sulle release del software, ma occasionalmente su altri eventi, come una iniziativa di raccolta fondi, la scoperta di una vulnerabilità nella sicurezza (vedere sezione chiamata «Annunciare le Vulnerabilità della Sicurezza») più avanti in questo capitolo, o un cambiamento nel progetto può essere postato anche lì. Poiché essa è a basso traffico e usata solo per cose importanti, la mailing list `annuncia` ha tipicamente la più alta quantità di iscritti di ogni mailing list nel progetto (certo, ciò significa che voi non dovete abusare con essa— riflettete prima di postare). Per evitare che gente a caso faccia annunci, o peggio, spam di passaggio, la mailing list `annuncia` deve sempre essere moderata.

Cercate di fare gli annunci in tutti i posti in modo simultaneo quanto più è possibile. La gente potrebbe confondersi vedendo un annuncio sulla mailing list ma poi non vedendolo nella pagina principale del sito del progetto o nell'area dei comunicati stampa. Se ricevete i vari cambiamenti (emails, scrittura delle pagine web, ecc..) in un fila di attesa e le mandate tutte in un riga potete mantenere molto piccola la finestra di incoerenza.

Per un evento meno importante, potete eliminare una o tutte le uscite di cui sopra. L'evento sarà ugualmente notato dal mondo di fuori in proporzione alla susa importanza. Per esempio, se una nuova release del software è un evento importante, il fissare solamente la data della nuova release, mentre tuttavia in qualche modo fa notizia, non è quasi così impostate quanto la release stessa. Il fissare una data ha il valore di una email alle mailing list giornaliere (non alla maling list annuncia) e di un aggiornamento della linea del tempo del progetto e della pagina web dello stato, ma niente di più.

Comunque potreste vedere quella data apparire nella discussione da qualche altra parte in Internet, ovunque ci sia gente interessata al progetto. Persone che stanno in disparte sulle vostre mailing list, solo per ascoltare e mai dire qualcosa, non stanno necessariamente zitte altrove. L'orale dà una distribuzione molto ampia; dovreste contare su essa, e costruire anche annunci minori in modo da incoraggiare una trasmissione informale accurata. Nello specifico, post che vi aspettate siano quotati dovrebbero avere una parte finalizzata ad essere quotata, giusto come se steste scrivendo un comunicato stampa. Per esempio:

> *Giusto un aggiornamento nel progresso: state progettando di rilasciare la versione 2.0 di Scanley a metà Agosto 2005. Potete sempre controllare http://www.scanley.org/status.html per aggiornamenti. La principale funzionalità sarà la ricerca con le espressioni regolari.*
>
> *Le altre nuove funzionalità includono: ... Ci saranno anche varie correzioni di bug, incluso: ...*

Il primo paragrafo è breve, dà i due più importanti pezzi di informazione (la data del rilascio e la principale nuova funzionalità), e un URL da visitare per ulteriori notizie. Se quel paragrafo è la sola notizia che attraversa lo schermo di qualcuno, state ancora facendo molto bene. Il resto della email potrebbe andar perso senza aver effetto sulla sostanza del contenuto. Certo, a volte le persone vorranno linkare all'intera email comunque, ma appunto come spesso, essi ne citeranno solo una piccola parte. Dato che l'ultima ipotesi è una possibilità, potete anche renderla facile per loro, e nel patteggiare avere qualche influenza su ciò che viene citato.

Annunciare le Vulnerabilità della Sicurezza

Gestire le vulnerabilità della sicurezza è differente dal gestire ogni altro tipo di report di bug. Nel software libero, fare le cose apertamente e con trasparenza è normale quasi come un credo religioso. Ogni passo della gestione standard dei bug è visibile a tutti quelli che hanno la cura di di guardare: l'arrivo del report iniziale, la conseguente discussione, e l'eventuale correzione.

I bug della sicurezza sono differenti. Essi possono compromettere i dati degli utenti, e magari l'intero computer dell'utente. Per discutere apertamente un tale problema si dovrebbe avvisare della sua esistenza il mondo intero—incluse tutte le parti che potrebbero fare un uso maligno del bug. Anche solo facendo l'invio della correzione in effetti dà l'annuncio dell'esistenza del bug (ci sono persone potenziali che sferrano gli attacchi che guardano i log degli invii dei progetti pubblici, sistematicamente alla ricerca di cambiamenti che indicano problemi di sicurezza nel codice di pre cambiamento). Molti progetti open source hanno fissato lo stesso gruppo di passi per gestire questo conflitto fra l'essere aperti e la segretezza, basati su queste linee guida:

1. Non parlate del bug pubblicamente finché non sia disponibile un correzione; quindi fornite la correzione all'esatto stesso momento in cui annunciate il bug.

2. Arrivate con quella correzione quando più velocemente potete—specialmente se qualcuno al di fuori del progetto riportò il bug, perché allora voi sapete che c'è almeno una persona al di fuori del progetto che è in grado di sfruttare la vulnerabilità.

In pratica questi principi portano a una serie di passi molto standardizzati che sono descritti nella sezione sotto.

Ricevere il report

Ovviamente il progetto ha bisogno di ricevere i bug nella sicurezza da ognuno. Ma gli indirizzi dei rapporti dei bug non ne hanno bisogno, perché anche essi sono visti da chiunque. Quindi, abbiate un mailing list separata per ricevere i rapporti dei bug nella sicurezza. Questa mailing list non deve avere archivi leggibili pubblicamente, e i loro iscritti devono essere strettamente controllati—solo sviluppatori di lungo periodo fidati possono stare sulla mailing list. Se avete bisogno di una definizione formale di “fidati”, dovete usare “chiunque abbia avuto l'accesso all'invio da due anni o più”, o qualcosa di simile, per evitare favoritismi. Questo è il gruppo che deve gestire i bug nella sicurezza.

Idealmente, la mailing list sulla sicurezza non dovrebbe essere protetta da spam o moderata, perché non potete volere che un importante report sia filtrato o ritardato giusto perché è avvenuto che nessun moderatore fosse online quel weekend. Se usate programmi di protezione da spm automatici, cercate di configurarli con settaggi di alta tolleranza; è meglio consentire pochi spam che perdere un report. Affinché la mailing list sia efficiente dovete pubblicizzare il suo indirizzo, certo; ma dato che non sarà moderata o, al massimo, leggermente protetta da spam, non cercate mai di postare il suo indirizzo senza una qualche sorta di trasformazione di nascondimento, come descritto in sezione chiamata «Nascondere gli indirizzi presenti negli archivi» in Capitolo 3, *L'Infrastruttura Tecnica* . Fortunatamente per nascondere dell'indirizzo non c'è bisogno che l'indirizzo sia illeggibile; vedere http://subversion.tigris.org/security.html, e prendete visione del sorgente della pagina Html, per un esempio.

Sviluppate la correzione silenziosamente

E così cosa fa la mailing list quando riceve un report? Il primo compito è quello di valutare la serietà e la urgenza del problema:

1. Quanto seria è la vulnerabilità? Permette a chi fa l'attacco di prendere la direzione del computer si qualcuno che usa ilo vostro software? O si perdono semplicemente informazioni sulla grandezza di qualcuno dei suoi file?

2. Quanto facile è sfruttare la vulnerabilità? Può un attacco essere prestabilito, o richiede una conoscenza profonda, o un calcolo studiato, e fortuna?

3. *Chi* fece il report del problema a voi? Lar risposta a questa domanda non cambiate la natura della vulnerabilità, certo, ma vi dà un'idea di quante altre persone potrebbero sapere di essa. Se il report viene da una degli sviluppatori del progetto, voi potete respirare un pò più facilmente (ma solo un poco) perché potete confidare sul fatto che egli non ha parlato a nessuno di esso. D'altro canto, se il report viene con una email da `anonimo14@globalhackerz.net`, allora sarebbe meglio che che voi agiate quanto più velocemente possibile. La persona vi fece un favore informandovi del problema, ma non avete idea di quante persone sono state informate da lui, o di quanto abbia aspettato prima di sfruttare la vulnerabilità sulle installazione caricate.

Notate che la differenza di cui stiamo parlando qui è fra *urgente* e *estremamente urgente*. Anche quando il report proviene da una fonte nota e amica, ci potrebbe essere altra gente sulla rete che scoprì il bug da tempo e che non lo ha giusto riportata. La sola occasione in cui le cose non sono urgenti è quando il bug in modo innato non compromette la sicurezza in modo serio.

L'esempio "`anonimo14@globalhackerz.net`" non è faceto, tra parentesi. Voi potete ricevere realmente dei rapporti di bug da persone dall'identità nascosta, che con le loro parole e il loro comportamento, non chiariscono del tutto se sono dalla vostra parte o no. Non ha importanza: se hanno fatto rapporto sul buco nella sicurezza a voi essi riterranno di avervi fatto un favore, e voi potete rispondere a modo. Ringraziateli per il report, dategli una data nella o prima della quale progettate di rilasciare una correzione pubblica, e teneteli nel giro. A volte essi possono dare *a voi* una data—cioè, una minaccia intrinseca di pubblicizzare il bug in una certa data, siate pronti o no. Questo può essere

avvertito come un minaccioso gioco di potere, ma è più probabilmente una azione preventiva risultante dalla passata delusione con produttori di software indifferenti che non presero i report di sicurezza abbastanza seriamente. D'altra parte, voi non potete permettervi di irritare questa persona. Dopotutto, se ilo bug è serio, egli ha conoscenze che potrebbero causare grossi problemi ai vostri utenti. Trattate bene queste persone che fanno i report, e sperate che trattino bene voi.

Un'altra persona che fa frequenti report di sicurezza è il professionista della sicurezza, uno che controlla il codice per campare e si mantiene con le ultime notizie sulle vulnerabilità della sicurezza. Queste persone hanno di solito esperienza su tutti e due i lati della staccionata—essi hanno ricevuto e mandato report, magari più della maggioranza degli sviluppatori nel vostro progetto. Essi anche di solito danno anche una scadenza sulla correzione di una vulnerabilità, prima che diventi pubblica. Questa scadenza può essere in un certo modo negoziabile ma questo tocca a chi manda il report; le scadenze sono diventate riconosciute fra i professionisti della sicurezza in qualche grado come l'unica via affidabile per ottenere che le organizzazioni affrontino i problemi di sicurezza tempestivamente. Così non trattate le scadenze da maleducati; è una tradizione che gode di buona reputazione, e ci sono buone ragioni per questo.

Una volta che ne conoscete la serietà e l'urgenza, potete partire col lavoro della correzione. A volte c'è un compromesso fra il fare una correzione elegantemente e il farla velocemente; questo è il perché dovete accordarvi sull'urgenza prima di partire. Mantenete la discussione ristretta ai membri della mailing list, più chi fece il report originariamente (se lui vuole essere coinvolto), a qualche sviluppatore che è necessario tirare dentro per ragioni tecniche.

Non inviate la correzione al deposito. Mantenetela in forma di patch fino alla data di andare in pubblico. Nel caso doveste inviarla, anche con un log innocente a vedersi, qualcuno potrebbe notarla e capire il cambiamento. Voi non sapete mai chi sta guardando nel deposito, e perché sarebbe interessato. Cessare le email di invio non aiuterebbe; prima di tutto l'interruzione nella sequenza dell'invio delle email potrebbe sembrare il se stessa sospetta, e comunque, i dati sarebbero tuttavia nel deposito. Appunto fate tutto lo sviluppo in una patch e tenete la patch in qualche posto privato, magari un deposito privato separato conosciuto solo alle persone al corrente del bug. (Se usate un sistema di controllo della versione decentrato come Arch o SVK, potete fare il lavoro sotto il pieno controllo della versione, e giusto tenete quel deposito inaccessibile agli esterni.)

I numeri CAN/CVE

Potete aver visto un *numero CAN* o un *numero CVE* associati con i problemi di sicurezza. Questi numeri di solito appaiono come "CAN-2004-0397" o "CVE-2002-0092", per esempio.

Ambedue i tipi di numeri rappresentano lo stesso tipo di entità: una voce nella lista di “Vulnerabilità comuni ed Esposizioni” curata in http://cve.mitre.org/. Il proposito della lista è quello di fornire nomi standardizzati per tutti i problemi conosciuti di sicurezza, in modo che chiunque deve usare un unico nome canonico quando ne discute uno e un posto centralizzato dove andare per trovare maggiori informazioni. La sola differenza tra il numero “CAN” e “CVE” è che il primo rappresentata una voce candidata, non ancora approvata per l'inserimento nella lista della dal Consiglio editoriale della CVE e il secondo rappresenta una voce approvata. Comunque tutti e due i tipi di voce sono visibili al pubblico, e il numero di una voce non cambia quando è approvato—il prefisso “CAN” è solo sostituito da “CVE”

Una voce CAN/CVE non contiene di per se stessa una descrizione completa del bug non dice come proteggersi da essa. Invece, contiene un breve sommario, e un elenco di riferimenti a risorse esterne (come archivi di mailing lista) dove la gente possa andare per prende maggiori informazioni. Il vero proposito di è quello di fornire uno spazio ben organizzato in cui in cui ogni vulnerabilità possa avere un nome e una chiara rotta verso dati ulteriori. Vedere http://cve.mitre.org/cgi-bin/cvename.cgi?name=2002-0092 per un esempio di voce. Notate che i riferimenti possono essere molto succinti, con le sorgenti che appaiono abbreviazioni criptate. Una chiave per queste abbreviazione si trova a http://cve.mitre.org/cve/refs/refkey.html.

Se la vostra vulnerabilità soddisfa i criteri CVE, potete voler acquisire ad esso un numero CAN. Il procedimento per fare ciò e deliberatamente impedito: fondamentalmente voi dovete conoscere qualcuno, o conoscere qualcuno che conosce qualcuno. Ciò non è folle come potrebbe suonare. Affinché il Consiglio Editoriale del CVE eviti di essere travolto con candidature spurie o scritte in modo deficitario, prende candidature solo da fonti già note o fidate. Per ottenere che la vostra vulnerabilità sia elencata, quindi, avete bisogno di un percorso di conoscenze dal vostro progetto al Consiglio Editoriale del CVE. Chiedete fra i vostri sviluppatori; uno di essi o probabilmente conosce qualcun altro che ha percorso il procedimento CAN prima, o qualcuno che lo ha, ecc.. Il vantaggio di fare ciò in questo modo è anche quello che in qualche punto lungo la catena, qualcuno può sapere abbastanza da dire a) che essa non conterebbe come vulnerabilità o come esposizione in accordo con i criteri del MITRE, per cui non è il caso di inviarla o b) la vulnerabilità *ha* già un numero CAN o CVE. La seconda cosa può presentarsi se il bug è stato appena pubblicato su un'altra mailing list consultiva di sicurezza, per esempio su http://www.cert.org/ o sulla mailing list BugTraq a http://www.securityfocus.com/. (Se ciò è avvenuto senza che il vostro progetto ne abbia sentito parlare, allora vi dovreste preoccupare di cos'altro potrebbe andare avanti di cui voi non avete conoscenza).

Se ottenete un numero CAN/CVE, voi volete ottenerlo nei primi stadi dell'investigazione sul bug, in modo che tutte le ulteriori comunicazioni possano far riferimento a quel numero. Le voci CAN vengono impedite fino alla data della pubblicazione; la voce consisterà in un simbolo vuoto (così voi non perdete il nome), ma ciò non rivelerà nessuna informazione sulla vulnerabilità fino alla data in cui voi annunciate il bug e la correzione.

Maggiori informazioni sul processo CAN/CVE possono essere trovate a http://cve.mitre.org/about/candidates.html, e una esposizione particolarmente chiara dell'uso da parte di un progetto open source dei numeri CAN/CVE si trova a http://www.debian.org/security/cve-compatibility.

Pre notifica

Una volta che il team delle risposte sulla sicurezza (cioè quegli sviluppatori che stanno sulla mailing list della sicurezza, o che sono stati tirati dentro l'affare della sicurezza con un particolare rapporto) ha una correzione pronta, dovete decidere come distribuirla.

Se inviate semplicemente la correzione al deposito, o diversamente la annunciate al mondo, voi in effetti costringete chiunque usi il vostro software ad aggiornare immediatamente o rischiare di essere bucato. Talvolta è appropriato, quindi, fare una pre notifica per certi utenti importanti. Ciò è particolarmente vero per il software client/server, dove ci possono essere ben noti server che sono degli allettanti obiettivi per chi fa gli attacchi. Gli amministratori di questi server apprezzerebbero il fatto di avere qualche giorno in più o due per fare l'aggiornamento, in modo da essere protetti nel frattempo che il metodo di attacco diventa di pubblica conoscenza.

Pre notifica significa semplicemente inviare email a quegli amministratori prima della data dell'uscita, per dire loro della vulnerabilità e come correggerla. Dovreste mandare la pre notifica solo a persone che voi confidate siano discrete con le informazioni. Cioè, la qualifica per ricevere la pre notifica è duplice: il destinatario deve far girare un grosso, importante server l'essere in pericolo sarebbe una questione seria, *e* e il destinatario dovrebbe essere noto per non essere uno che chiacchiera sul problema di sicurezza prima della data dell'uscita.

Mandate le email di prenotifica individualmente (una alla volta) ad ogni destinatario. *Non* mandate la lista intera dei destinatari subito, perché essi vedrebbero i nomi gli uni degli altri—nel senso che voi stareste avvisando ogni destinatario del fatto che ogni *altro* destinatario può avere una buca nella sicurezza nel suo server. Mandando a tutti loro l'email via CC non visibile (BCC) non è una buona soluzione nemmeno, perché alcuni admin proteggono la loro casella di posta con filtri antispam che o bloccano o riducono la priorità delle email BCC, dal momento che così tanto spam è inviato via BCC di questi tempi.

Qui c'è un esempio di email di pre notifica:

```
DA: Qui il Vostro Nome
To: admin@large-famous-server.com
Risposta a: Qui in Vosto Nome (non l'indirizzo della mailing list sulla sicurezza)
Oggetto: Notifica confidenziale della vulnerabilità di Scanley.


Questa email è una una notifica confidenziale di una allarme nel server di Scanley

Prego *non inoltrare* nessuna parte di questa email ad alcuno. Non c'è annuncio fi

State ricevendo questa email perché (noi pensiamo) che voi facciate girare un serv

Riferimenti:
===========

   CAN-2004-1771: Sovraccarico dello stack di Scanley nelle query

Vulnerabilità:
==============

Il server può essere messo nelle condizioni di eseguire comandi arbitrari se il lo

Serietà:
=========

   Molto seria. Può comportare l'arbitraria esecuzioni di codice sul server.

Soluzioni:
============

   Il mettere l'opzione del 'processo del linguaggio naturale' a 'off' in scanley.

Patch:
======

   La patch di sotto si applica a Scanley 3.0, 3.1, e 3.2.

Una nuova release (la Scanley 3.2.1) verrà creata il o appena prima del 19 Maggio,

[...la patch va qui...]
```

Se avete un numero CAN, includetelo nella pre notifica (come mostrato sopra) anche le informazioni a sono ancora bloccate e quindi la pagina del MITRE non mostra nulla. L'inclusione del numero CAN mette il destinatario in grado di sapere con certezza che il bug del quale è stato pre notificato è lo stesso di cui ha sentito parlare attraverso i canali pubblici, così egli non ha a preoccuparsi se è necessaria qualche azione ulteriore o no, che è precisamente lo scopo dei numeri CAN/CVE.

Distribuite la correzione pubblicamente

L'ultimo passo nella gestione di un bug di sicurezza è quello di di distribuire la pubblicità della correzione. In un unico, comprensivo annuncio, dovreste descrivere il problema, dare il numero CAN/

CVE, se esiste, descrivere come risolvere il problema, e come correggerlo permanentemente. Di solito “fix” significa aggiornare alla nuova versione del software, sebbene, a volte, significa applicare una patch, particolarmente se il software è fatto girare normalmente in una forma sorgente in qualche modo. Se create una nuova release, essa dovrebbe differire dalla release esistente esattamente per la patch sulla sicurezza. In questo modo gli admin prudenti possono aggiornare senza preoccuparsi su quale altra cosa essa stia avendo effetto; essi non dovrebbero nemmeno preoccuparsi dei futuri aggiornamenti perché la correzione della sicurezza ci sarà in tutte le release future come cosa naturale. (Dettagli sulle procedure della release sono discussi in sezione chiamata «Le Releases di Sicurezza» in Capitolo 7, *Confezione, Rilascio, e Sviluppo Quotidiano*.)

Se o meno la correzione pubblica comporta una nuova release, fate l'annuncio con circa la stessa priorità che in una nuova release: mandate una email alle mailing list annuncia del progetto, fate un nuovo comunicato stampa, aggiornate la voce di Freshmet, ecc.. Mentre non dovreste mai minimizzare l'esistenza di un bug di sicurezza al di fuori della relazione con la reputazione del progetto, potete impostare certamente il tono e la prominenza di un annuncio di sicurezza perché sia adatto alla severità del problema. Se il buco nella sicurezza è solo una rischiosità secondaria relativa alle informazioni, non un metodo di attacco che consente all'intero computer dell'utente di essere rilevato, allora esso non può giustificare tanta agitazione. Potete anche decidere di non distrarre la mailing list `annuncia` con esso. Dopotutto, se il progetto guarda al lupo ogni volta, gli utenti potrebbero finire col pensare che il software sia meno sicuro di quanto in realtà lo sia, e potrebbero anche non credervi quando avete da annunciare veramente un grosso problema. Vedere http://cve.mitre.org/about/terminology.html per una buona introduzione al problema del giudizio sulla severità.

In generale, se siete incerti su come trattare un problema di sicurezza, trovate qualcuno esperto e parlategli di esso. Valutare e gestire le vulnerabilità è molto una abilità acquisita, ed è facile fare dei passi falsi le prime volte.

Capitolo 7. Confezione, Rilascio, e Sviluppo Quotidiano

Questo capitolo parla di come i progetti di software libero confezionano e rilasciano i loro software e di come si organizzano le procedure generali di sviluppo su questi obiettivi.

Una differenza principale fra i progetti open source e quelli proprietari è il difetto di un controllo centralizzato sul team di sviluppo. Quando sta venendo preparata una nuova release, questa differenza è specialmente forte: una compagnia può richiedere al suo team di sviluppo di focalizzarsi sula prossima release, mettendo da parte lo sviluppo di nuove funzionalità e la correzione di bugs non critici fino a che il rilascio non è avvenuto. I gruppi di volontari no sono così monolitici. La gente lavora al progetto per tutti i tipi di motivazioni, e quelli non interessati a favorire una data release vogliono continuare il lavoro di sviluppo mentre la release sta andando avanti. Poiché lo sviluppo non si ferma, i processi di rilascio tendono a impiegare più tempo, ma sono meno distruttivi, dei processi di rilascio commerciali. Questo è in po' come ritrovare la strada maestra. Ci sono due modi di riparare una strada: potete chiuderla completamente, così che il gruppo che sta riparando la strada maestra può sciamare su di essa alla piena capacità finché il problema è risolto, o potete lavorare su una coppia di corsie alla volta, lasciando le altre aperte al traffico. Il primo modo è molto efficace *per il gruppo di riparazione*, but not for anyone else—ma non per gli altri la strada è chiusa completamente fino a che il lavoro non è completato. Il secondo modo richiede molto più tempo comporta più problemi per il gruppo di riparazione (ora essi devono lavorare con meno gente e meno equipaggiamento, in condizioni più ristrette, con bandiere per rallentare e dirigere il traffico, ecc..), ma almeno la strada rimane utilizzabile, sebbene non con la sua piena capacità.

I progetti open source tendono a funzionare nel secondo modo. Infatti per un pezzo di software maturo con diverse linee di rilascio mantenute simultaneamente, il progetto è un genere di stato di permanente riparazione di una strada. C'è sempre una coppia di corsie chiuse ma un basso livello di inconveniente di fondo vien sempre tollerato dal gruppo di sviluppo nel complesso, cosicché le release vengono effettuate con una programmazione regolare.

Il modello che rende ciò possibile si estende a più che una sola release. E' il principio di mettere in parallelo operazioni che non sono mutualmente interdipendenti un principio che non è in nessun modo unico dello sviluppo open source, certamente, ma è un principio che i progetti open source implementano fra propri in particolar modo. Essi non possono permettersi di infastidire molto sia il gruppo di lavoro sulla strada sia il regolare traffico, ma non possono permettersi anche di avere gente dedita a fare affidamento sui coni arancione e a fare segnalazioni lungo il traffico. Così essi sono attratti da processi che hanno piatti, costanti livelli di spese generali amministrative, piuttosto che picchi e valli. I volontari vogliono generalmente lavorare con piccoli ma consistenti quantità di incomodo. La prevedibilità permette loro di venire e andare senza senza preoccuparsi se il loro programma contrasterà con ciò che sta avvenendo nel progetto. Ma se il progetto fosse soggetto a un importante programma in cui alcune attività escludessero altre attività, il risultato sarebbe un sacco di sviluppatori seduti a oziare per un sacco di tempo la qualcosa sarebbe non solo inefficiente ma tediosa, e quindi dannosa, in quanto uno sviluppatore annoiato è equivalente presto a un ex sviluppatore.

Il lavoro di release è usualmente la più notevole operazione che avviene in parallelo con lo sviluppo, così i metodi descritti nella prossima sezione sono per lo più attrezzati a rendere possibili le releases. Comunque notate che essi si applicano anche ad altre operazione eseguibili in parallelo, come la traduzione e l'internazionalizzazione, grandi cambiamenti alle API fatti generalmente lungo tutto il codice base, ecc..

Numerazione delle Releases

Prima di parlare di come fare una release, vediamo come chiamare una release, che vuol dire sapere veramente cosa significa per un utilizzatore. Una release significa che:

- I vecchi bugs sono stati corretti. Questa è probabilmente l'unica cosa sulla cui verità gli utilizzatori possono contare.
- Nuovi bugs si sono aggiunti. Questa è anche una cosa su cui si può contare, tranne a volte nel caso di releases di sicurezza o di altre cose che avvengono una sola volta (vedere sezione chiamata «Le Releases di Sicurezza» più avanti in questo capitolo).
- Nuove funzionalità possono essere state aggiunte.
- Nuove opzioni di configurazione possono essere state aggiunte, o il fine delle vecchie opzioni può essere stato cambiato di poco. Le procedure di installazione anche possono essere cambiate sin dalla ultima release, sebbene uno speri di no.
- Possono essere stati introdotti cambiamenti incompatibili, sicché alcuni formati di dati usati dalle versioni più vecchie non sono più utilizzabile senza sopportare qualche sorta di (possibilmente manuale) passo a senso unico.

Come potete vedere non tutte queste cose sono buone. Questo è il motivo per cui utilizzatori di esperienza si avvicinano alle nuove releases con qualche apprensione, specialmente quando il software è maturo e stavano già per la maggior parte facendo quello che volevano (o pensavano quello che volevano). Anche l'arrivo di nuove funzionalità è una benedizione di diversa specie, in quanto può significare che il software si comporterà ora in maniera imprevista.

Lo scopo della numerazione delle versioni, quindi, è duplice: ovviamente dovrebbero comunicare senza ambiguità l'orine delle releases (cioè, guardando a una di due releases, uno può sapere quella che viene dopo), ma anche dovrebbe indicare in modo quanto più compatto possibile il grado e la natura dei cambiamenti nella release.

Tutto questo in un numero? Si, più o meno, si. La strategia della numerazione delle releases è una delle più vecchie superflue discussioni qua e là (vedere sezione chiamata «Più semplice l'argomento, più lungo il dibattito» in Capitolo 6, *Comunicazione*), ed è improbabile che il mondo pattuisca un unico completo standard molto presto. Comunque poche buone strategie sono emerse, insieme a un principio su cui si è universalmente d'accordo: *essere in armonia*. Scegliete uno schema di numerazione, documentatelo, e seguitelo. I vostri utilizzatori vi ringrazieranno.

I Componenti del Numero di Rilascio

Questa sezione descrive le convenzioni formali della numerazione delle releases, e presuppone, una piccola precedente conoscenza. Se già avete familiarità con queste convenzioni, potete saltare questa sezione.

I numeri delle releases son gruppi di cifre separate da punti:

Scanley 2.3
Singer 5.11.4

...e così via. I punti *non* ono punti decimali, essi sono solamente separatori; "5.3.9" sarebbe seguito da "5.3.10". Pochi progetti hanno fatto intendere diversamente, il più famoso il kernel Linux con la sua sequenza "0.95", "0.96"... "0.99" leading up to Linux 1.0, ma la convenzione che i punti non sono decimali è fermamente stabilita e deve essere considerato uno standard. Non c'è un limite al numero dei

componenti (parti di cifre non contenenti il punto), ma la maggior parte dei progetti non va oltre tre o quattro. La ragione diventerà chiara avanti.

In aggiunta a componenti numerici i progetti a volte aggiungono un'etichetta descrittiva tipo “Alpha” o “Beta” (vedere Alpha e Beta), per esempio:

Scanley 2.3.0 (Alpha)
Singer 5.11.4 (Beta)

Un qualificatore Alpha o Beta significa che quella release *precede*una futura release che avrà lo stesso numero senza il qualificatore Così, "2.3.0 (Alpha)" porta alla fine a "2.3.0". Per far entrare queste release candidate in una riga, gli stessi qualificatori posso avere dei meta-qualificatori. Per esempio qui c'è una serie di releases che sarebbero rese disponibili al pubblico:

Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)
Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0

Notate che quando essa ha il qualificatore “Alpha” Scanley "2.3" si scrive "2.3.0". I due numeri sono equivalenti—tutti i componenti 0 seguenti possono sempre essere eliminati per brevità ma quando un qualificatore è presente, la brevità smette di esistere comunque, come pure uno potrebbe scegliere la completezza invece.

Altri qualificatori semi regolari usano includere “Stabile”, “Instabile”, “Sviluppo” e “RC” (per “Release Candidate”). Le più largamente usate sono comunque "Alpha" e "Beta", con “RC” che concorre da vicino per il terzo posto, ma notate che “RC” include sempre un meta-qualificatore numerico. Cioè non rilasciate "Scanley 2.3.0 (RC)", voi rilasciate "Scanley 2.3.0 (RC 1)", seguita dalla RC2, etc.

Queste tre etichette “Alpha”, “Beta” e “RC” sono piuttosto largamente conosciute ora, e raccomando di non usare le altre, anche se a prima vista sembrano scelte migliori perché sono parole normali, non gergo. Ma la gente che installa software da release ha già familiarità con quelle tre, è non c'è motivo per fare cose gratuitamente in modo differente dal modo in cui chiunque altro le fa.

Sebbene i punti nei numeri delle realases non siano punti decimali, indicano un significato del valore della posizione. Tutte le realeases "0.X.Y" precedono la "1.0" (che è equivalente alla “1.0.0”, certo). La"3.14.158" precede immediatamente la "3.14.159" e non immediatamente precede la "3.14.160" come la "3.15.qualcosa", e così via.

Una politica di numerazione delle releases rende capace l'utilizzatore di di guardare ai numeri di rilascio per lo stesso pezzo di software e dire, proprio dai numeri, le importanti differenze fra queste due relaeases. In un tripico sistema a tre componenti il primo componente è il *numero maggiore*, il secondo è il *il secondo è il numero minore*, il terzo è il *micro numero*. FPer esempio, la release "2.10.17" è la diciassettesima micro-relaese nella decima linea di release minore entro la seconda serie di release maggiore. Le parole “linea” e “serie” sono usate informalmente qui, ma significano ciò che uno si aspetterebbe. Una serie maggiore sono semplicemente tutte le releases che hanno in comune lo stesso numero maggiore, e una serie minore (o linea minore) sono semplicemente tutte le releases che hanno in comune lo stesso numero maggiore *e* lo stesso numero minore. Cioè la "2.4.0" e la "3.4.1" non si trovano nella stessa serie minore anche se ambedue hanno “4” come numero minore; d'altra parte la "2.4.0" e la "2.4.2" si trovano nella stessa linea minore, sebbene non siano adiacenti se la “2.4.1” è stata rilasciata fra di esse.

Il significato di questi numeri è esattamente ciò che vi aspettereste: un aumento del numero maggiore indica che il cambiamento maggiore è avvenuto; un cambiamento del numero minore indica che sono

avvenuti cambiamenti minori, e un cambiamento del micro numero indica cambiamenti trascurabili. Alcuni progetti aggiungono un quarto componente, usualmente chiamato *numero di patch*, specialmente per un un controllo più fine sulle differenze fra le loro releases (in maniera disorientante, altri progetti usano “patch” come sinonimo di “micro” in un sistema. Ci sono anche progetti che usano un sistema a tre componenti). Ci sono anche progetti che usano l'ultimo componente come *numero di fabbricazione*, incrementato ogni volta che il software è costruito, e non rappresenta nessun cambiamento al di fuori dell'allestimento. Ciò aiuta il progetto a mettere in relazione ogni rapporto di bugs con uno specifico allestimento, ed è forse molto utile quando i pacchetti binari sono il metodo di distribuzione di default.

Sebbene ci siano molte differenti convenzioni su quanti componenti usare, e cosa significhino i componenti, le differenze tendono a diventare piccole, voi tenete una piccola tolleranza, ma non troppo. Le prossime sue sezioni parlano delle due convenzioni più largamente usate.

La Strategia Semplice

La maggior parte dei progetti ha delle regole su che tipo di cambiamenti sono permessi nella release se uno sta incrementando solo il micro numero, differenti regole per il numero minore e ancora differenti regole per il numero maggiore. Non c'è uno standard stabilito per queste regole ancora, ma qui io descriverò una politica che è stata usata con successo da più di un progetto. Voi potete giusto adottare questa politica nei vostri porgetti, ma anche se non lo volete, questo è tuttavia un buon esempio del tipo di informazione che i numeri di release dovrebbero portare con sé. Questa politica è adattata dal sistema di numerazione usato dal progetto APR, vedere http://apr.apache.org/versioning.html.

1. I cambiamenti dei micro numeri (cioè i cambiamenti alla stessa linea minore) devono essere sia in avanti compatibili che indietro compatibili. Cioè, i cambiamenti dovrebbero essere solo le correzioni di bugs o accrescimenti molto piccoli delle funzionalità esistenti. Nuove funzionalità non dovrebbero essere introdotte in una micro release.

2. Cambiamenti del numero minore (cioè entro la linea maggiore) devono essere compatibili all'indietro, ma non necessariamente in avanti. E' normale aggiungere nuove funzionalità ad una release minore, ma non troppe in un sola volta.

3. I cambiamenti al numero maggiore segnano i limiti della compatibilità. Una nuova release maggiore può essere incompatibile sia in avanti che all'indietro. Ci si aspetta che una release maggiore abbia nuove funzionalità e può avere interi nuovi d set di funzionalità.

Che significhi *compatibile in avanti* e *compatibile all'indietro* , esattamente, dipende da ciò che il vostro software fa, ma nel contesto, non sono aperte a tante interpretazioni. Per esempio, se il vostro software è una applicazione client/server, allora “compatibile all'indietro” significa che l'aggiornamento del server alla 2.6.0 non dovrebbe causare per i clients la perdita di funzionalità o comportamenti differenti da quelli di prima (eccetto per i bugs che sono stati corretti, certo). D'altra parte, l'aggiornamento di uno di quei clients alla 2.6.0 , insieme al server, potrebbe rendere disponibili *nuove* funzionalità per quel client, funzionalità di cui i client 2.5.4 non sanno come avvantaggiarsi. Se ciò avviene, l'aggiornamento *non* è “compatibile in avanti”: chiaramente non potete ora tornare indietro con quel client alla 2.5.4 e mantenere tutte le funzionalità che aveva nella 2.6.0, perché alcune di quelle funzionalità erano nuove nella 2.6.0.

Questo è il motivo per cui le micro releases sono essenzialmente per le correzioni dei bugs. Esse devono rimanere compatibili in entrambe le direzioni: se voi aggiornate dalla 2.5.3 alla 2.5.4, poi cambiate idea e tornate indietro alla 2.5.3, nessuna funzionalità dovrebbe andar perduta. Certo i bugs corretti riapparirebbero dopo il ritorno alla precedente versione, ma voi non perdereste nessuna funzionalità, eccetto per il fatto che i bugs reintrodotti impediscono l'uso di alcune funzionalità esistenti.

I protocolli client/server sono giusto uno dei campi dalle molte possibili compatibilità. Un' altra è la formattazione dei dati: il software scrive i dati in uno spazio permanente? Se così, i formati che legge

devono seguire le leggi di compatibilità promesse dalla politica dei numeri di rilascio. La versione 2.6.0 deve poter leggere files scritti nella 2.5.4, ma può in silenzio aggiornare il formato a qualcosa che la 2.5.4 non può leggere, perché la possibilità di ritornare indietro non è un requisito della 2.6.0 per via delle delimitazioni riguardanti un numero minore. Se il vostro progetto distribuisce librerie di codice per l'impiego in altri programmi, allora le API sono anche un dominio di compatibilità: dovete essere sicuri che le regole di compatibilità per il sorgente e il binario siano dette in maniera tale che l'utilizzatore informato non debba mai chiedersi se è sicuro e opportuno aggiornare o no. Egli deve essere capace di guardare i numeri è e saperlo istantaneamente.

In questo sistema voi non avete una chance per una fresca partenza finché non incrementate il numero maggiore. Questo può essere spesso un reale inconveniente: ci possono essere funzionalità che voi volete aggiungere, o protocolli che volete ridisegnare, cose che non possono essere semplicemente fatte mentre mantenete la compatibilità. Non c'è una soluzione magica a questo, eccetto che cercare di disegnare le cose in modo estensibile in primo luogo (una argomento che facilmente merita il proprio libro, ma certamente fuori tema in questo). Ma pubblicare una politica di compatibilità delle releases, e aderirvi, è una parte delle distribuzione del software a cui non si può sfuggire. Una sgradevole sorpresa può allontanare un sacco di utilizzatori. La politica appunto descritta è buona in parte perché è già abbastanza diffusa, ma anche perché è facile da spiegare e da ricordare, anche per coloro che non hanno ancora familiarità con essa.

E' generalmente convenuto che queste regole non si applicano alle release pre-1.0 (sebbene la vostra politica di release lo stabilisca esplicitamente, giusto per essere chiari). Un progetto che sia ancora allo sviluppo iniziale può rilasciare la 0.1, 0.2, 0.3 e così via in sequenza, finché non sia pronto per la 1.0, e le differenze fra queste releases possono essere arbitrariamente grandi. I micro numeri nelle releases pre-1.0 sono opzionali. A seconda della natura del vostro progetto e delle differenze fra le releases potreste trovare utile avere la 0.1.0, la 0.1.1. ecc.., oppure no. Le convenzioni per le releases pre-1.0 sono piuttosto permissive, principalmente perché la gente capisce che forti costrizioni di compatibilità intralcerebbero molto il primo sviluppo, e anche perché i primi che le adottano tendono ad essere indulgenti comunque.

Ricordate che tutte queste ingiunzioni si applicano a questo sistema a tre componenti. Il vostro progetto potrebbe venir su con un differente sistema a tre componenti, oppure potrebbe decidere di non avere bisogno di un così fine granulosità e usare invece un sistema a due componenti. La cosa importante è deciderlo per tempo, rendere pubblico ciò che i componenti significano, e aderire ad essi.

La Strategia pari/dispari

Alcuni progetti usano la parità del componente numero minore per indicare la stabilità del software. Pari significa stabile, dispari significa instabile. Ciò si applica solo al numero minore, non al numero maggiore a al micro numero. Incrementi nel micro numero ancora significano correzioni di bug (non nuove funzionalità), e incrementi nel numero maggiore ancora indicano grossi cambiamenti, nuovi set di funzionalità, ecc..

Il vantaggio del sistema pari/dispari, che è stato usato dal progetto kernel di Linux fra gli altri, è che offre una via per rilasciare nuove funzionalità per la fase di prova, senza costringere gli utilizzatori della produzione a un codice potenzialmente instabile. Le persone possono vedere dai numeri che la "2.4.21" va bene per l'installazione sul loro attivo web server, ma che la "2.5.1" dovrebbe essere usata solamente per esperimenti sulle work station. Il team di sviluppo gestisce i report di bugs che arrivano dalla serie minore (numerata dispari), e quando le cose incominciano a sistemarsi dopo un certo numero di micro releases in quella serie, incrementano il numero minore (così facendolo diventare pari), riportano il micro numero a “0”, e rilasciano il pacchetto presumibilmente stabile.

Questo sistema conserva, o almeno non va in conflitto con le linee guida date prima. Esso semplicemente carica ulteriormente il numero minore di qualche extra informazione. Questo costringe d'altronde il numero minore ad essere incrementato di circa due volte tanto spesso quanto sarebbe

necessario, ma non c'è un gran male in ciò. Il sistema pari dispari è probabilmente il migliore per i progetti che hanno cicli di release molto lunghi, e che per loro natura hanno una grande fetta di utilizzatori conservatori che valutano la stabilità al di sopra delle nuove funzionalità. Questo comunque non è il solo modo di fare il test di nuove funzionalità allo stato selvaggio, comunque. sezione chiamata «Stabilizzare una Release» Più avanti in questo capitolo si descrive un altro, forse più comune, metodo di rilasciare al pubblico codice potenzialmente instabile, contrassegnato in modo che le persona abbiano un'idea del compromesso rischio/beneficio vedendo il nome della release.

Rami Di Release

Dal punto di vista dello sviluppatore un progetto di software libero è in continuo stato di release sviluppatori eseguono sempre l'ultimo codice disponibile, perché vogliono scoprire i bugs, e perché seguono il progetto abbastanza da vicino da essere capaci di tirarsi indietro da aree instabili per quanto riguarda le funzionalità. Essi spesso aggiornano la loro copia del software ogni giorno, a volte più di una volta al giorno, e quando registrano un cambiamento, essi possono ragionevolmente aspettarsi che ogni altro sviluppatore lo riceverà entro ventiquattro ore.

Come, allora un progetto dovrebbe creare una release formale? Dovrebbe ricevere una istantanea dell'albero in un momento in tempo, farne il pacchetto, e passarlo al mondo come, diciamo, la versione "3.5.0"?. Il senso comune dice di no. Primo, ci può non essere un momento nel tempo in cui l'intero albero di sviluppo è pulito e pronto per il rilascio. Le funzionalità già cominciate potrebbero trovarsi in vari stadi di completamento. Qualcuno potrebbe aver cercato in un cambiamento più importante dei bug, ma il cambiamento potrebbe essere controverso e sotto dibattito al momento in cui la foto è stata fatta. Se così, non funzionerebbe ritardare la fotografia fino a quando il dibattito non termini, perché un altro dibattito non collegato potrebbe iniziare nel frattempo, e allora voi dovreste attendere che anche *quello* termini. Non è garantito che questo processo termini.

In ogni caso, usare fotografie dell'intero albero per le releases interferirebbe con il lavoro di sviluppo in corso, anche se l'albero potrebbe essere messo in uno stato di rilascio. Per esempio questa fotografia potrebbe andare per la "3.5.0"; presumibilmente la successiva fotografia sarebbe la "3.5.1" e conterrebbe per lo più correzioni dei bugs trovati nella 3.5.0. Ma se ambedue sono fotografie dello stesso albero, cosa si suppone che gli sviluppatori facciano nel tempo fra le due releases? Essi non possono aggiungere nuove funzionalità; le linee guida di compatibilità non lo permettono. Ma non tutti saranno entusiasti di correggere i bugs nel codice 3.5.0. Alcuni possono avere alcune funzionalità che stanno cercando di completare, e si arrabbieranno se saranno obbligati a scegliere fra il sedere oziosi e lavorare a cose alle quali non sono interessati, giusto perché i processi di rilascio del progetto chiedono che l'albero di sviluppo rimanga fermo in maniera non naturale.

La soluzione a questi problemi è usare sempre una *release ramo*. Una release ramo è appunto un ramo nel sistema di controllo della versione (vedere *Ramo (branch)*), sul quale il codice destinato a questa release può essere isolato dalla linea principale dello sviluppo. Il concetto di rami di release non è certamente originale del software libero; molti sviluppi commerciali lo usano anche. Comunque, in ambienti commerciali, i rami di release sono a volte considerati un lusso una specie di formale “miglior pratica” di cui nell'entusiasmo di una scadenza maggiore, se ne può fare a meno mentre ognuno nel team si affanna a stabilizzare l'albero principale.

I rami di release sono quasi richiesti nel software open source, comunque. Io ho visto progetti fare una release senza di essi, ma il risultato è stato sempre che alcuni sviluppatori stessero oziosi mentre altri—usualmente una minoranza—lavoravano a fare uscire la release fuori della porta. Il risultato è usualmente cattivo per molti versi. Primo, la velocità dello sviluppo principale è diminuita. Secondo. La qualità erea peggiore di quanto sarebbe stato necessario, perché c'erano solo poche persone a lavorare ad essa, ed essi si affrettavano a finire in modo che ogni altro potesse tornare a lavorare. Terzo, esso divide il team di sviluppo psicologicamente, dando luogo a una situazione in cui differenti tipi di lavoro interferiscono con ogni altro tipo senza necessità. Gli sviluppatori che restano oziosi sarebbero

probabilmente felici di collaborare con una certa attenzione al ramo di release, nella misura in cui quella sarebbe una scelta che potrebbero fare in accordo con i loro programmi e interessi. Ma senza il ramo, la loro scelta diventa “Devo partecipare al progetto o no?” invece di “Devo partecipare alla release oggi, o lavorare a quella funzionalità che stavo sviluppando nella linea principale del codice?”

Il Meccanismo Dei Rami di Release

L'esatto meccanismo della creazione di un ramo di release dipende dal vostro sistema di controllo della versione, certo, ma i concetti generali sono gli stessi nella maggior parte dei sistemi. Un ramo usualmente vien fuori da un altro ramo o dal tronco. Tradizionalmente, il tronco è dove si ha la linea principale dello sviluppo, libera dai vincoli della release. Il primo ramo della release, quello che porta alla release “1.0”, vien fuori dal tronco. Nel CVS il comando di ramo sarebbe qualcosa come

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_0_X
```

o in Subversion, come questo:

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.0.x
```

(Tutti questi esempi accettano il sistema di numerazione a tre componenti. Mentre non posso mostrare i comandi per ogni sistema di controllo della versione, farò degli esempi in CVS e Subversion e spero che i corrispondenti comandi in altri sistemi possano essere dedotti da questi due.)

Notare che noi creammo il ramo "1.0.x" (con una lettera "x") invece di "1.0.0". Questo perché la stessa linea minore cioè lo stesso ramo—sarebbe stato usato per le micro releases in quella linea. Il reale processo di stabilizzazione dei rami è trattato sezione chiamata «Stabilizzare una Release» più avanti in questo capitolo. Qui noi ci occupiamo appunto dell'interazione fra il sistema di controllo della versione con il processo di release. Quando il ramo di release è stabilizzato e pronto, è il momento di tracciare una fotografia del ramo:

```
$ cd RELEASE_1_0_X-working-copy
$ cvs tag RELEASE_1_0_0
```

or

```
$ svn copy http://.../repos/branches/1.0.x http://.../repos/tags/1.0.0
```

Questa traccia ora rappresenta lo stato esatto dell'albero origine nella release 1.0.0 (ciò è utile nel caso che qualcuno abbia bisogno di prelevare una vecchia versione dopo che distribuzioni confezionate e i binari siano stati dismessi). La successiva micro release nella stessa linea è preparata nello steso modo nel ramo 1.0.x, e quando è pronta, una traccia è fatta per la 1.0.1. Più avanti, risciacquatura, si ripete con la 1.0.2, e così via. Quando è il tempo di partire pensando alla release 1.1.x, create un nuovo ramo dal tronco:

```
$ cd trunk-working-copy
$ cvs tag -b RELEASE_1_1_X
```

or

```
$ svn copy http://.../repos/trunk http://.../repos/branches/1.1.x
```

Il mantenimento può continuare in parallelo lungo la 1.0.x e la 1.1.x e un rilascio può essere fatto indipendentemente da ambedue le differenti linee. La vecchia serie è raccomandata per gli amministratori di sito più conservatori che possono non voler fare il grosso salto alla (diciamo) 1.1 senza una attenta preparazione. Intanto, persone più avventurose prendono la più recente release sulla linea più alta, per essere sicuri di fare proprie le più recenti funzionalità, anche a rischio di una piuttosto grande instabilità.

Questa non è l'unica strategia delle releases ramo, certo. In alcune circostanze, può neanche essere la migliore, sebbene abbia funzionato bene per i progetti in cui è stata impiegata. Usate una strategia che sembra funzionare, ma ricordate i punti principali: il proposito di una release ramo è quello di isolare il lavoro di rilascio dalle fluttuazioni dello sviluppo giornaliero, e dare al progetto una entità fisica intorno alla quale organizzare il processo di rilascio. Il processo è descritto in dettaglio nelle successiva sezione.

Stabilizzare una Release

La *stabilizzazione* è il processo mettere una ramo di release in uno stato in cui si può rilasciare; cioè il processo di decidere quali cambiamenti ci saranno nelle release, quali no, e dar forma al contenuto del ramo di conseguenza.

Ci sono un sacco di di potenziali pene in quella parola, “decidere”. La corsa alla funzionalità dell'ultimo minuto è un fenomeno familiare nei progetti di software in collaborazione: appena gli sviluppatori vedono che la release sta per avvenire, si agitano a finire il loro correnti cambiamenti, per non perdere il battello. Questo, certamente, è l'esatto opposto di ciò che volete al momento del rilascio. Sarebbe molto meglio per la gente lavorare alle funzionalità in una confortevole ritmo, e non preoccuparsi e non preoccuparsi tanto se i loro cambiamenti riusciranno a farcela per questa release o per la prossima. Più uno cerca di imbottire i cambiamenti nella release all'ultimo minuto, più il codice è destabilizzato, e (usualmente) più bugs si creano.

La maggior parte degli ingegneri di software convengono in teoria su abbozzi di criteri circa quali cambiamenti debbano essere consentiti in una linea di rilascio durante il suo periodo di stabilizzazione. Ovviamente correzioni per importanti bugs dovrebbero entrarci, specialmente per bugs senza correzioni che non risolvono il problema. Gli aggiornamenti della documentazione vanno bene, così come le correzioni ai messaggi di errore (eccetto quando sono considerati parte dell'interfaccia e devono rimanere stabili). Molti progetti anche consentono certi cambiamenti non rischiosi e non di base di entrare durante la stabilizzazione, e si possono avere formali linee guida per la misurazione del rischio. Ma nessun ammontare di formalizzazione può ovviare al bisogno del giudizio umano. Ci saranno casi in cui il progetto deve semplicemente prendere una decisione se un dato cambiamento deve entrare in una release. Il pericolo è che siccome ognuno vuol vedere il suo cambiamento preferito ammesso nella release, ci sarà una gran quantità di gente motivata a consentire cambiamenti, e gente non abbastanza motivata a impedirli.

Così, il processo di stabilizzazione di una release consiste per lo più nel creare un meccanismo per dire “no”. Il trucco per un progetto open source, in particolare, è venir fuori con modi di dire “no” che non abbiano molto come risultato dare la sensazione di una ferita o creare il disappunto degli sviluppatori, e che anche non impediscano che cambiamenti validi entrino nella release. Ci sono molti modi per fare ciò. E' piuttosto facile inventare sistemi che soddisfino questi criteri, una volta che il team si è focalizzato su di essi come criteri importanti. Qui descriverò due dei più popolari sistemi, agli estremi dello spettro, ma non permetto che questo scoraggi il vostro progetto dall'essere creativo. Una abbondanza di altri espedienti è possibile; questi sono giusto due che ho visto funzionare in pratica.

Dittatura Da Parte del Proprietario Della Release

Il gruppo conviene di permettere a una persona di essere *proprietario della release*. Questo proprietario ha la parola finale sui cambiamenti che devono entrare nella release. Certo, ci si aspetterebbe ed è

normale che ci siano discussioni e argomentazioni, ma alla fine il gruppo deve assegnare al proprietario la sufficiente autorità per prendere le decisioni finali. Affinché questo sistema funzioni, è necessario scegliere una persona con la competenza tecnica per comprendere tutti i cambiamenti e la posizione sociale e le capacità di navigare fra le discussioni puntando alla release senza causare tanti sensi di risentimento.

Un comportamento comune del proprietario è dire “non penso che ci sia qualcosa di sbagliato in questo cambiamento, ma non abbiamo avuto abbastanza tempo per provarlo, per cui non deve entrare nelle release”. Ciò aiuta molto se il proprietario della release ha larghe conoscenze tecniche del progetto, e ha la capacità di rendere conto del perché il cambiamento potrebbe esser potenzialmente destabilizzante (per esempio la sua interazione con altre parti del software, o questioni di portabilità). La gente potrà a volte chiedere che tali decisioni siano giustificate o sostenere che il cambiamento non è un rischio come sembra. Queste conversazioni non devono essere provocatorie, nella misura in cui il proprietario della release è capace di prendere in considerazione tutte le argomentazioni obbiettivamente e non come un colpo alle sue gambe

Notate che non è necessario che il proprietario della release sia la stessa persona del leader del progetto (nei casi in cui c'è un leader del progetto; vedere sezione chiamata «I Dittatori Benevoli» in Capitolo 4, *L'Infrastruttura Sociale e Politica*). Infatti a volte è bene assicurarsi che non siano la stessa persona. Le capacità che fanno un buon leader di sviluppo non sono necessariamente le stesse che fanno un buon proprietario di release. In una cosa così importante come il processo di release, può essere saggio avere qualcuno che controbilanci il giudizio del leader di progetto.

Contrastate il ruolo del proprietario di release con un un ruolo meno dittatoriale descritto in sezione chiamata «Il manager di release» più avanti in questo capitolo.

Votare Il Cambiamento

All'estremo opposto della dittatura da parte del proprietario di release, gli sviluppatori possono semplicemente votare su quali cambiamenti includere nella release. Comunque, poichè la funzione più importante per la stabilizzazione delle release è *escludere* cambiamenti, è importante creare un sistema di voto in modo tale che fare cambiamenti alla release comporti una azione positiva da parte di più di uno sviluppatore. Per inserire un cambiamento ci dovrebbe essere bisogno più di una semplice maggioranza (vedere sezione chiamata «Chi Vota?» in Capitolo 4, *L'Infrastruttura Sociale e Politica*). Diversamente un voto a favore e uno contro un dato cambiamento sarebbe sufficiente per inserirlo nella release e si creerebbe una sciagurata dinamica per cui ciascuno sviluppatore voterebbe per i propri cambiamenti, mentre sarebbe riluttante a votare contro i cambiamenti degli altri, per paura di possibili ritorsioni. Per evitare ciò, il sistema dovrebbe essere congegnato in modo che sottogruppi di sviluppatori debbano agire in cooperazione per inserire cambiamenti nella release. Ciò significa non solo che più persone revisionano ogni cambiamento, ma rende uno sviluppatore individuale meno indeciso nel votare contro un cambiamento, perché egli sa che nessuno in particolare fra quelli che votarono per esso può prendere il suo voto contro come un affronto personale. Più grande è il numero di persone coinvolte, più numerose diventano le discussioni sui cambiamenti e meno numerose quelle sugli individui.

Il sistema che noi usiamo nel progetto Subversion sembra aver centrato un buon bilanciamento, per cui io lo raccomando qui. Affinché un cambiamento sia apportato a un ramo di release, almeno tre sviluppatori devono votare a favore di esso, e nessuno contro. Un singolo voto “no” è sufficiente a impedire che il cambiamento sia incluso; cioè un voto “no” in un contesto di release è equivalente a un veto (vedere sezione chiamata «I Veti»). Naturalmente ogni voto di questo tipo deve essere accompagnato da una giustificazione, e in teoria si potrebbe non tener conto del veto, se abbastanza gente ritenesse che esso è non ragionevole e obbliga a un voto speciale su di esso. In pratica, ciò non è mai successo, e prevedo che non succederà mai. Le persone sono conservatrici nei confronti delle release in ogni modo, e quando qualcuno si sente abbastanza fortemente a favore del veto nei confronti dell'inclusione di una cambiamento, c'è usualmente una buona ragione per ciò.

Poiché la procedura di rilascio è deliberatamente orientata verso il conservatorismo, le giustificazioni portate per il veto sono talvolta procedurali piuttosto che tecniche. Per esempio, una persona può ritenere che un cambiamento sia ben scritto e che sia improbabile che causi nuovi bugs, ma vota contro la sua inclusione nella micro release semplicemente perché è troppo grossa magari apporta nuove funzionalità, o in qualche modo sottile non riesce a seguire completamente le linee guida di compatibilità. Io occasionalmente ho visto anche sviluppatori porre il veto a qualcosa semplicemente perché avevano una sensazione viscerale che il cambiamento avesse bisogno di ulteriori prove, anche se essi non potevano individuare bugs in esse con un esame interno. Le persone si lagnavano un poco, ma il veto resisteva e il cambiamento non veniva incluso nella release (non ricordo se qualche bug veniva trovato o no in una ulteriore prova, comunque).

Portare avanti una stabilizzazione di release in collaborazione

Se il vostro progetto opta per un sistema di voto per il cambiamento, è imperativo che i meccanismi di organizzazione delle schede di voto e di votare sia il più adatto possibile. Anche se c'è una pletora di software open source disponibile, in pratica la cosa più facile da fare è giusto preparare nel ramo di release un file di testo, chiamato `STATO` or `VOTI` o qualcosa di simile. Questo file elenca ogni cambiamento proposto per l'inclusione insieme a tutti i voti a favore e contro, più eventuali note o commenti (Proporre un cambiamento non significa necessariamente votarlo, strada facendo, sebbene le due cose vadano insieme). Una voce in un tale file appare così:

```
* r2401 (issue #49)
  Prevent client/server handshake from happening twice.
  Justification:
    Avoids extra network turnaround; small change and easy to review.
  Note:
    Ciò fu discusso in http://.../mailing-lists/message-7777.html
    e altrimessaggi in quel thread.
  Voti:
    +1: jsmith, kimf
    -1: tmartin (breaks compatibility with some pre-1.0 servers;
                 admittedly, those servers are buggy, but why be
                 incompatible if we don't have to?)
```

In questo caso il cambiamento ottenne due voti a favore, ma ad esso fu messo il veto da tmartin, che diede ragione del veto in una nota scritta fra parentesi. Non ha importanza l'esatto formato della annotazione; qualunque cosa il vostro progetto corregga è ben fatto forse la spiegazione di tmartin per il veto dovrebbe andare in “Note”: la sezione o magari la descrizione del cambiamento dovrebbe comprendere una intestazione “Descrizione:” per adattarsi alle altre sezioni. La cosa importante è che tutte le altre informazioni necessarie per valutare il cambiamento devono essere raggiungibili, e che il meccanismo per dare il voto siano quanto più leggero possibile. Il cambiamento proposto è individuato dal suo numero di revisione nel deposito (in questo caso una singola revisione, la r2401, sebbene un cambiamento proposto potrebbe appunto facilmente consistere in revisioni multiple). Si conviene che la revisione si riferisca a un cambiamento fatto sul tronco; se il cambiamento fosse già nel ramo di release, non ci sarebbe bisogno di votare su di esso. Se il vostro sistema di controllo della versione non ha una chiara sintassi per far riferimento a un singolo cambiamento, allora il progetto dovrebbe crearne una. Affinché il votare sia pratico, ciascun cambiamento in considerazione dovrebbe essere identificato senza ambiguità.

Queste proposte o voti per un cambiamento sono affidabili per essere sicuri che si applichino in modo pulito al ramo di release, cioè, si applichino senza conflitti (vedere *conflitto*). Se ci sono conflitti, allora la nuova voce dovrebbe puntare o a una patch apposita che si applica in modo pulito, o a un ramo temporaneo che sostiene una apposita versione del cambiamento, per esempio:

```
* r13222, r13223, r13232
  Riscrive libsvn_fs_fs's auto-merge algorithm
  Giustificazione:
    unacceptable performance (>50 minutes for a small commit) in
    a repository with 300,000 revisions
  Ramo:
    1.1.x-r13222@13517
  Voti:
    +1: epg, ghudson
```

Questo esempio è preso dalla vita reale; esso proviene dal file `STATO` per il processo di rilascio Subversion 1.1.4. Notare che esso usa le revisioni originali come gradi del cambiamento, anche se c'è un ramo con una versione del cambiamento con un conflitto superato (il ramo combina anche tre revisioni di tronco in una, la r13517, per rendere più facile l'unione dei cambiamenti nella release, dovrebbe essere approvato). Sono fornite la revisioni originali, perché esse sono le entità più facili da revisionare, perché hanno i messaggi di log originali. Il ramo temporaneo non dovrebbe avere questi messaggi di log; per evitare duplicazione di informazioni (vedere sezione chiamata «Singolarità dell'informazione» in Capitolo 3, *L'Infrastruttura Tecnica*), l messaggio di log del ramo per la r13517 dovrebbe dire semplicemente “Adatta la r13222, r13223, e 13232 per il backport al ramo 1.1.x”. Tutte le altre informazioni sui cambiamenti possono essere inviate alle loro revisioni originali.

Il manager di release

Il processo reale di unire (vedere *Unire(merge)*) cambiamenti approvati nel ramo di release può essere effettuato da un qualunque sviluppatore. Non c'è bisogno che sia una persona il cui compito sia quello di unire i cambiamenti; se c'è una gran quantità di cambiamenti, la cosa migliore può essere quella di suddividere il carico.

Comunque, sebbene l'unione dei cambiamenti e il voto avvengono in un modo di fare decentrato, nella pratica c'è una o due persone che guidano il processo di release. Il ruolo è benedetto talvolta come *manager di release*, ma è completamente differente da un proprietario di release (vedere sezione chiamata «Dittatura Da Parte del Proprietario Della Release» precedentemente in questo capitolo) che ha la parola finale sui cambiamenti. I managers di release tengono traccia di quanti cambiamenti sono al momento sotto considerazione, quanti sono stati approvati, quanti sembrano da approvarsi, ecc... Se essi hanno l'impressione che cambiamenti importanti non stanno avendo sufficiente attenzione, e potrebbero essere lasciati fuori per una manciata di voti, essi potrebbero brontolare con gli altri sviluppatori per ottenere una revisione o un voto. Quando un gruppo di cambiamenti viene approvato, queste persone possono prendersi il compito di unirle al ramo di release; è cosa buona se gli altri lasciano questo compito a loro, fintanto che ognuno ritiene che essi non sono obbligati a fare tutto il lavoro a meno che non siano esplicitamente incaricati di farlo. Quando viene il tempo di far uscire la release (vedere sezione chiamata «Prove e Rilascio» più avanti in questo capitolo), i managers di release si prendono anche la cura della logistica della creazione dei pacchetti della release finale, raccogliendo le firme digitali, uploadando i pacchetti, e facendo gli annunci pubblici.

Impacchettamento

La forma canonica per la distribuzione del software libero è il codice sorgente. Questo è vero indipendentemente dal fatto se il codice gira nella forma di sorgente (cioè può essere interpretato come Perl, Python, PHP, ecc..) o prima ha bisogno di esser compilato (come C, C++, Java, ecc..). Con il software compilato, gli utilizzatori probabilmente non compileranno da sè i sorgenti, ma invece installeranno da pacchetti binari per-costruiti (vedere sezione chiamata «Pacchetti Binari» più avanti in questo capitolo). Comunque questi pacchetti binari sono tuttavia derivati da una distribuzione principale del sorgente. L'importanza del pacchetto sorgente sta nel definire senza ambiguità la release. Quando

il progetto distribuisce la, quello che vuol dire specificatamente è “I tre files del codice sorgente, che quando compilati (se necessario) e installati, producono la Scanley 2.5.0."

C' è uno standard ragionevolmente rigido su come devono presentarsi le release sorgenti. Si potrebbero vedere occasionalmente delle deviazioni da questo standard, me sono l'eccezione, non la regola. A meno che non ci sia una convincente ragione per fare diversamente, anche il vostro progetto dovrebbe seguire questo standard.

Il Formato

Il codice sorgente dovrebbe essere inviato in formati standard per trasportare alberi di directory. Per i sistemi operativi Unix e pseudo Unix, la convenzione è usare il formato TAR, compresso da **compress**, **gzip**, **bzip** o **bzip2** che sembra fare la compressione anche bene, in modo che non c'è bisogno di comprimere l'archivio dopo averlo creato.

> **TAR Files**
>
> *TAR* stands for "Archivio per nastro", perché il formato TAR rappresenta l'albero di directory come un flusso di dati lineari, che lo rende ideale per salvare l'albero di directory su nastro. La stessa proprietà lo rende anche lo standard per distribuire gli alberi di directory come un singolo file. Produrre files tar (o *tarballs*) è molto facile. Su qualche sistema il comando tar può produrre un archivio compresso da sé; in altri è usata un programma di compressione a parte.

Nome E Disposizione

Il nome del pacchetto dovrebbe consistere nel nome del software, più il numero di release, più il suffisso di formato per il tipo di archivio. Per esempio, Scanley 2.5.0, impacchettato per Unix che usa la compressione GNU Zip (gzip), apparirebbe come questo:

scanley-2.5.0.tar.gz

o per Windows che usa la compressione zip:

scanley-2.5.0.zip

Ciascuno di questi archivi, quando è estratto, dovrebbe creare un unico albero di directory chiamato `scanley-2.5.0` nella directory corrente. Sotto la nuova directory, il codice sorgente dovrebbe essere sistemato in una disposizione pronta per la compilazione (se c'è bisogno di compilazione) e per l'installazione. Nel livello più alto dell'albero delle directory ci dovrebbe essere un file di testo piano `README` che spieghi ciò che il software fa e quale release è, e che dà le indicazioni per altre risorse, come il sito del progetto, altri file di interesse, ecc.. Fra questi altri files ci dovrebbe essere un file fratello `INSTALL` del file `README` che dia istruzione su come costruire e installare il software per tutti i sistemi operativi che supporta. Come menzionato in sezione chiamata «Come Applicare Una Licenza Al Vostro Software» in Capitolo 2, *Partenza*, ci dovrebbe essere anche un file `COPYING` or `LICENSE` che fornisca i termini di distribuzione.

Ci dovrebbe essere anche un file `CHANGES` (a volte chiamato `NEWS`), che spieghi ciò che c'è di nuovo in quella release. Il file `CHANGES` accumula le liste dei cambiamenti per tutte le releases, in ordine cronologico inverso, in modo che la lista per quella release appaia in cima al file. A completare la lista c'è usualmente l'ultima cosa fatta per stabilizzare il ramo di release; alcuni progetti scrivono la lista un pò alla volta mentre si sviluppano, altri preferiscono salvarla alla fine di tutto, e hanno una persona a scriverla, che prende le informazioni setacciando i log del controllo di versione. La lista appare come qualcosa del genere:

```
Version 2.5.0
(20 December 2004, da /rami/2.5.x)
http://svn.scanley.org/repos/svn/tags/2.5.0/

 New features, enhancements:
    * Added regular expression queries (issue #53)
    * Added support for UTF-8 and UTF-16 documents
    * Documentation translated into Polish, Russian, Malagasy
    * ...

 Bugfixes:
    * fixed reindexing bug (issue #945)
    * fixed some query bugs (issues #815, #1007, #1008)
    * ...
```

L'elenco può facilmente essere tanto lungo quanto è necessario, ma non vi preoccupate di includere ogni piccola correzione di bug e accrescimento di funzionalità. Il suo proposito è solamente dare agli utilizzatori una visione d'insieme di quale sarebbe il guadagno ad aggiornare alla nuova release. Infatti, la lista dei cambiamenti è abitualmente inclusa nelle email di annuncio (vedere sezione chiamata «Prove e Rilascio» più avanti in questo capitolo), in modo che la scriviate con il pubblico nella mente.

CHANGES A Confronto Con il Changelog

Tradizionalmente un file chiamato Changelog elenca ogni cambiamento apportato a un progetto cioè ogni revisione inviata al sistema di controllo di versione. Ci sono vari formati per il file Cahngelog; i dettagli dei formati non sono importanti qui, in quanto essi contengono tutti la stessa informazione: la data del cambiamento, il suo autore, e un breve sommario (o giusto un messaggio di log per quel cambiamento).

Un file `CHANGES` è differente. Anch'esso è una lista dei cambiamenti, ma solo quelli ritenuti importanti da vedere da parte di un certo pubblico, e con metadati come la data esatta e l'autore spogli. Per evitare confusioni non usate termini in modo scambievole. Alcuni progetti usano “NEWS” invece di "CHANGES"; sebbene ciò evita la possibilità di di una confusione con “Changelog”, è un po' un termine improprio, poiché un file CHANGES conserva un sacco di informazioni sui cambiamenti per tutte le releases, e quindi un sacco di vecchie notizie in aggiunta alle nuove notizie in cima.

I files Changelog possono lentamente scomparire comunque. Essi erano utili nei giorni in cui in cui il CVS era l'unica scelta per un sistema di controllo di versione, perché i dati sui cambiamenti non erano facilmente estraibili da CVS. Comunque, con i più recenti sistemi di controllo della versione, i dati che si usavano tenere nel Changelog possono essere richiesti al deposito del controllo della versione in ogni momento, rendendo inutile per il progetto tenere un file statico contenente quei dati nei fatti peggiore che l'inutilità, poiché il Changelog duplicherebbe i messaggi di log già immagazzinati nel deposito.

Usualmente ci sono poche differenze, per esempio perché il pacchetto contiene alcuni file generati necessari per la compilazione e la configurazione (vedere sezione chiamata «Compilazione e Installazione» più avanti in questo capitolo), o perché contiene un software di terze parti di cui il progetto non fa manutenzione, ma ciò è richiesto e ciò è probabile che gli utilizzatori non abbiano. Ma anche se l'albero distribuito corrispondesse esattamente ad alcuni alberi di sviluppo nel deposito del controllo di versione, la distribuzione stessa non sarebbe un copia funzionante (vedere *copia di lavoro*). Si suppone che la release un punto di riferimento statico una particolare, immutabile configurazione dei files sorgenti. Se essa fosse una copia funzionante, il danno sarebbe che gli utilizzatori potrebbero aggiornarla, e successivamente pensare di avere ancora la release quando nei fatti egli ha qualcosa di differente.

Ricordate che il pacchetto è lo stesso indipendentemente dalla confezione. La release cioè la precisa entità riferita a quando qualcuno dice "Scanley 2.5.0"—è l'albero creato estraendo un file zip o tar. Così il progetto potrebbe offrire tutti questi per il download:

scanley-2.5.0.tar.bz2
scanley-2.5.0.tar.gz
scanley-2.5.0.zip

...ma l'albero sorgente estraendoli deve essere lo stesso. Quell'albero sorgente è la distribuzione.; la forma in cui è scaricato è un pura questione di convenienza. Certe differenze secondarie fra i pacchetti sorgente sono ammissibili: per esempio, nel pacchetto Windows i file di testo devono avere un fine linea con CRLF (Carriage Return and Line Feed), mentre il pacchetto Unix deve usare giusto LF. Gli alberi possono essere sistemati in maniera poco differente fra i pacchetti destinati a sistemi operativi differenti, anche, se questi sistemi operativi richiedono diversi tipi sistemazione per la compilazione. Comunque, queste sono di base le tutte le trasformazioni secondarie. I file sorgenti di base devono essere gli stessi nell'impacchettamento di una data release.

Mettere le maiuscole o non metterle

Quando ci si riferisce a un progetto col nome, la gente generalmente lo nomina con la maiuscola come un nome proprio. E mette le maiuscola agli acronimi, se ce ne sono ecc.. Se la maiuscola debba essere usata nel pacchetto tocca al progetto `Scanley-2.5.0.tar.gz` o `scanley-2.5.0.tar.gz` sarebbero ottimi, per esempio (io preferisco personalmente la seconda, perché non mi piace che la gente pigi il tasto shift, ma un gran numero di progetti inviano pacchetti con la maiuscola). La cosa importante è che le directory che si creano estraendo il pacchetto tar usino anche la maiuscola. Non si dovrebbero essere sorprese: l'utilizzatore dovrebbe essere di predire con perfetta accuratezza il nome della directory che so creerà quando estrarrà una distribuzione.

Pre-releases

Quando inviate una pre-release o una candidate release il qualificatore è con esattezza una parte del numero di release, così includetelo nel nome del pacchetto. Per esempio, la sequenza ordinata di releases alfa e beta date prima in sezione chiamata «I Componenti del Numero di Rilascio» dovrebbero apparire nel nome del pacchetto come:

scanley-2.3.0-alpha1.tar.gz
scanley-2.3.0-alpha2.tar.gz
scanley-2.3.0-beta1.tar.gz
scanley-2.3.0-beta2.tar.gz
scanley-2.3.0-beta3.tar.gz
scanley-2.3.0.tar.gz

La prima si estrarrebbe in una directory chiamata `scanley-2.3.0-alpha1`, la seconda nella `scanley-2.3.0-alpha2`, e così via.

Compilazione e Installazione

Per software che richiede compilazione e installazione dal sorgente, si sono usualmente delle procedure standard che si presume gli utilizzatoti esperti siano capaci di seguire. Per esempio, per programmi scritti in C, C++, o certi altri linguaggi compilati, lo standard per sistemi tipo Unix e che l'utilizzatore batta:

```
$ ./configure
$ make
```

```
# make install
```

Il primo comando individua come sa l'ambiente e prepara il processo di allestimento (ma non installa), e il secondo comando installa sul sistema. I primi due comandi sono impartiti come utente normale, il terzo come radice. Per maggiori dettagli su come settare il sistema, vedere l'eccellente libro *GNU Autoconf, Automake, and Libtool* di Vaughan, Elliston, Tromey, e Taylor. Esso è pubblicato su carta da New Riders, e il suo contenuto è anche disponibile in forma freeware online a http://sources.redhat.com/autobook/.

Questo non è il solo standard, sebbene sia uno dei più diffusi. Il sistema di allestimento Ant (http://ant.apache.org/) sta guadagnando popolarità, specialmente con progetti scritti in Java, ed ha le sue proprie procedure standard per l'allestimento e l'installazione. Inoltre, certi linguaggi di programmazione come Perù e Pathos, raccomandano che, per la maggioranza del programmi scritti in quel linguaggio sia usato lo stesso metodo (per esempio moduli Perl usano il comando **perl Makefile.PL**). Se non è ovvio per voi quale debba essere lo standard applicabile per il vostro progetto, chiedete a uno sviluppatore esperto; potete con sicurezza accettare che *qualche* standard si applica, anche se non sapete quale venga prima.

Quale che sia lo standard adatto per il vostro progetto, non deviate da esso a meno che non dobbiate certamente farlo. Le procedure standard per l'installazione sono in pratica dei responsi automatici a stimoli specifici per un sacco di amministratori di sistema attualmente. Se essi vedono richieste di aiuto be note documentate nel file, ciò istantaneamente genera in essi la credenza che il vostro progetto è generalmente fuori dalle convenzioni, e anche che sarebbe riuscito bene in altre cose. Anche, come discusso in sezione chiamata «Downloads» in Capitolo 2, *Partenza*, avere una procedura standard di allestimento piace ai potenziali sviluppatori.

Su Windows gli standards per l'allestimento e per l'installazione sono un po' meno fissi. Per progetti che richiedono la compilazione, la convenzione generale sembra essere inviare un albero che si può sistemare in un modello di area di lavoro/progetto dell'ambiente di sviluppo standard di Microsoft (Developer Studio, Visual Studio, VS.NET, MSVC++, ecc...). A seconda della natura del vostro software, può essere possibile offrire una opzione di allestimento su Windows via ambiente Cygwin. (http://www.cygwin.com/) E certamente, se state usando un linguaggio o un framework di programmazione che viene con le sue convenzioni di allestimento e installazione, per esempio Phiton o Perl dovreste semplicemente usare quello che sia il metodo standard per quel framework, su Windows, Unix, Mac OS X, o ogni altro sistema operativo.

Siate disponibili a metterci un sacco di impegno extra per rendere il vostro progetto conforme agli standards di rilievo per l'allestimento e l'installazione. L'allestimento e l'installazione sono una fase di ingresso: è giusto che le cose diventino più difficili dopo di essi, se è fuori discussione che lo siano, ma sarebbe un disonore nei confronti degli utilizzatori o degli sviluppatori se la prima interazione col software richiedesse passi imprevisti.

Pacchetti Binari

Sebbene la release formale sia un pacchetto di codice sorgente, la maggior parte degli utilizzatori installeranno da pacchetti binari, sia che siano forniti dal meccanismo di distribuzione del software del sistema operativo, sia che siano ottenuti manualmente dal sito del progetto o da terze parti. Qui “binario” non significa necessariamente “compilato”; significa giusto una forma pre-configurata del pacchetto che permette a un utilizzatore di installarla sul proprio computer senza passare attraverso le solite procedure di allestimento e installazione. Su RedHat GNU/Linux, è il sistema RPM; su Debian GNU/Linux, È il sistema APT (`.deb`); su Windows, di solito i files `.MSI` il file autoistallante `.exe`.

Se il file binario sia assemblato da gente strettamente associate al progetto, o da distanti terze parti, gli utilizzatori tendono a *considerarli* equivalenti alle releases ufficiali del progetto, e depositeranno i problemi nel tracciatore di bug apposito per il comportamento dei pacchetti binari. Quindi, è

nell'interesse del progetto fornire pacchetti con chiare linee guida, e lavorare moltissimo con esse per assicurarsi che quello che producono rappresenti il software in modo pulito e accurato.

La cosa principale che gli impacchettatori hanno bisogno di sapere è che essi dovrebbero basare sempre i loro pacchetti binari su una release ufficiale originale. Talvolta gli impacchettatori son tentati di tirar fuori una recente incarnazione del codice dal deposito, o di includervi cambiamenti selezionati che furono inviati dopo che la release fu fatta, per fornirli agli utilizzatori con certe correzioni di bugs o altri miglioramenti. Chi fa i pacchetti pensa di star facendo un favore ai suoi utilizzatori nel dare loro codice più recente, ma in realtà questa pratica causa una gran quantità di confusione. I progetti sono preparati a ricevere rapporti di bugs trovati nelle versioni rilasciate, e di bugs trovati nel recente tronco e nel codice del ramo maggiore (cioè trovati da gente lì apposta per far girare codice a rischio di stabilità e produttività). Quando un bug proviene da queste fonti, il risponditore sarà spesso capace di confermare che è noto che quel bug è presente in quell'istantanea, e forse che da allora è stato corretto e che l'utente dovrebbe aggiornare o aspettare la successiva release. Se esso è un precedente bug non noto, l'avere l'esatta release fa si che sia più facile riprodurlo e più facile classificarlo nel tracciatore.

progetti non sono preparati, comunque, a ricevere rapporti di bugs basati su un mezzo intermedio non specifico o su versioni ibride. Tali bugs possono essere difficili da riprodurre; inoltre possono essere dovuti a interazioni non previste in cambiamenti isolati introdotti da successivi sviluppi, quindi causano cattivi comportamenti di cui gli sviluppatori del progetto non dovrebbero essere incolpati. Ho visto una sconcertante gran quantità di tempo sprecata perché un bug era *absent* quando avrebbe dovuto essere presente: qualcuno stava facendo girare una versione leggermente modificata, basata (ma non identica) su una release ufficiale, e quando i bug predetto non si verificava ognuno doveva scavare un fossato per capire il perché.

Tuttavia quelle erano circostanze in cui chi faceva il pacchetto insisteva sul fatto che quelle modifiche alla release originale erano necessarie. Chi faceva i pacchetti avrebbe dovuto essere incoraggiato a tirar fuori questo con gli sviluppatori del progetto e a spiegare i suoi piani. Essi possono trovare consenso, ma mancando questo, almeno avranno notificato al progetto le loro intenzioni, così che il progetto può stare attento a insoliti rapporti di bugs. Gli sviluppatori possono rispondere mettendo un disclaimer sul sito del progetto, e possono chiedere a chi fa i pacchetti di fare la stessa cosa nel posto appropriato, di modo che gli utilizzatori di quel pacchetto binario sappiano che ciò che stanno prendendo non è esattamente la stessa cosa di ciò che il progetto ha rilasciato. Non ci deve essere animosità in questa situazione, sebbene purtroppo spesso ce ne sia. E' solo che chi fa i pacchetti ha degli obiettivi leggermente differenti da quelli degli sviluppatori. Coloro che fanno i pacchetti vogliono che i loro utilizzatori incontrino le migliori funzionalità. Gli sviluppatori vogliono anche questo, certo, ma hanno anche bisogno di essere sicuri di conoscere quali versioni del software ci sono in giro, in modo da poter ricevere rapporti di bugs intellegibili e garantire compatibilità. A volte i loro obiettivi sono in conflitto. Quando ciò avviene è bene avere in mente il fatto che il progetto non ha nessun controllo su coloro che fanno i pacchetti e che i vincoli degli obblighi funzionano in entrambe le direzioni. E' vero che il progetto sta facendo un favore a chi fa i pacchetti semplicemente producendo il software. Ma anche quelli che fanno i pacchetti stanno facendo un favore al progetto e si stanno accollando un lavoro non eccitante per rendere il software più largamente disponibile, spesso per ordine di rilievo. E' bene non essere d'accordo con gli impacchettatori, ma non è bene offenderli. Giusto cercate una soluzione alle cose meglio che potete.

Prove e Rilascio

Dopo che il tar originale è stato prodotto dal ramo stabilizzato di release, incomincia la parte pubblica del processo di rilascio. Ma prima che il tar sia reso disponibile al mondo diffusamente, dovrebbe essere approvato da un numero minimo di sviluppatori, di solito tre o più. L'approvazione non è una semplice questione di guardare dentro alla release per cercare ovvi errori; idealmente gli sviluppatori scaricano il tar, lo allestiscono e lo installano in un sistema pulito, fanno girare la suite di prova di regressione (vedere sezione chiamata «Testing automatizzato») in Capitolo 8, *Gestire i Volontari*, e fanno qualche prova manuale. Supponendo che il tar passi i tests, così come ogni altra lista di criteri che può avere

il progetto, gli sviluppatori firmano il tar che fa uso di GnuPG (http://www.gnupg.org/), PGP (http://www.pgpi.org/), o di qualche altro programma capace di produrre firme PGP-compatibile.

Nella maggior parte dei progetti, gli sviluppatori appunto usano le loro personali firme digitali, al posto di chiavi di progetto condivise, e possono firmare quanti sviluppatori si vuole (cioè, c'è un numero minimo, ma non un massimo). Più sviluppatori firmano, più tests subisce la release, e anche più grande è la probabilità che un utilizzatore impacciato con la sicurezza possa trovare da sé una percorso di fiducia digitale nei confronti del tar.

Una volta approvata, la release (cioè tutti i tar, i files zip, e tutti gli altri formati con cui viene fatta la distribuzione), dovrebbe essere collocata nell'area di download del progetto, accompagnata dalle firme digitali e dalle somme di controllo MD5/SHA1 (vedere http://en.wikipedia.org/wiki/Cryptographic_hash_function). Ci sono vari standards per fare ciò. Un modo è accompagnare ciascun pacchetto di release con un file che dia le corrispondenti firme digitali e uno che dia le somme di controllo. Per esempio, se una delle releases impacchettate è la `scanley-2.5.0.tar.gz`, sistemate nella stessa directory un file `scanley-2.5.0.tar.gz.asc` contenente le firme digitali per il tar, un altro file contenente le sue somme di controllo MD5, e opzionalmente un altro file `scanley-2.5.0.tar.gz.md5` e opzionalmente un altro file `scanley-2.5.0.tar.gz.sha1`, contenente la somma di controllo SHA1. Un altro modo di fornire il controllo è quello di riunire tutte le firma digitali per tutti i pacchetti rilasciati in un singolo file, lo `scanley-2.5.0.sigs`; lo stesso può essere fatto per le somme di controllo.

In realtà non importa in modo in cui lo facciate. Solo tenetene un semplice schema, descrivetelo chiaramente, e siate coerenti da release in release. Lo scopo di tutto questo firmare digitalmente e di far tutte queste somme di controllo è quello di fornire agli utilizzatori un modo per verificare che la copia che ricevono non sia stata corrotta in modo doloso. Gli utilizzatori sono sul punto di di far girare questo codice sul loro computer se il codice è stato corrotto, uno che vuol fare un attacco può presto avere una posta aperta a tutti i loro dati. Vedere sezione chiamata «Le Releases di Sicurezza» più avanti in questo capitolo per maggiori dettagli sulla paranoia.

Le Releases Candidate

Per le releases importanti contenenti molti cambiamenti, molti progetti preferiscono metter fuori le *release candidates* prima, cioè, `scanley-2.5.0-beta1` prima della `scanley-2.5.0`. Lo scopo di una candidate è quello di sottoporre a una estesa fase di testing prima di darle il benelacito di release ufficiale. Se si trovano problemi, essi vengono corretti un ramo di release ed viene esibita una nuova candidate release. (`scanley-2.5.0-beta2`). Il ciclo continua finché non sono rimasti più bugs inaccettabili, punto nel quale l'ultima release candidate diventa la release ufficiale, cioè l'unica differenza fra la release candidate e la release reale è la rimozione del qualificatore dal numero di versione.

Sotto moltissimi aspetti, una candidate release deve essere considerata allo stesso modo di una vera release. Il qualificatore *alpha*, *beta*, o *rc* è sufficiente a mettere in guardia gli utilizzatori conservatori dall'aspettare fino alla vera release, e certamente le email di annuncio delle candidate releases dovrebbero mettere in evidenza che il loro scopo è quello di sollecitare reazioni. Tranne questo date alle candidate release la stessa quantità di attenzione che date alle normali release. Dopotutto, voi volete che la gente usi le candidates, a causa della esposizione migliore per scoprire bugs non scoperti, e anche perché voi non sapete mai quale candidate si conclude dando inizio alla release ufficiale.

Annunciare le Releases

Annunciare una release è come annunciare ogni altro evento, e dovrebbe usare le procedure descritte in sezione chiamata «La Pubblicità» in Capitolo 6, *Comunicazione*. Ci sono tuttavia poche nuove cose specifiche da fare.

Ogni volta che date l'URL al tar scaricabile della release, assicuratevi di dare anche la somma di controllo MD5/SHA1 e i puntatori al file delle firma digitali. Poiché gli annunci avvengono in molte tribune (mailing lists, pagine di news, ecc..), ciò significa che gli utilizzatori possono prendere le somme di controllo da molte fonti, il che dà ai titubanti sulla sicurezza una assicurazione extra che le somme di controllo stesse non sono state corrotte. Dare il link al file delle firme digitali molte volte non rende queste firme digitali più sicure, ma rassicura la gente (specialmente quelli che non seguono il progetto da vicino) che il progetto prende la sicurezza seriamente.

Nelle email di annuncio, e sulle pagine di news che contengono più che una semplice fascetta pubblicitaria sulla release, assicuratevi di inserire la porzione rilevante del file CAMBIAMENTI, così la gente può vedere perché dovrebbe essere loro interesse aggiornare. Ciò è importante sia con le candidate releases sia con le releases finali; la presenza di correzioni di bug e di nuove funzionalità è importante nel tentare la gente a provare una candidate release.

Infine, non dimenticate di ringraziare il team di sviluppo, quelli che hanno fatto i tests, e tutti quelli che hanno dedicato tempo ad archiviare buoni report di bugs. Non distinguerteli per nome, comunque, a meno che non ci sia qualcuno responsabile di una grossa parte del lavoro, il valore del quale sia riconosciuto da ognuno nel progetto. Giusto stiate attenti a non scivolar giù con la tendenza scivolosa all'inflazione dei riconoscimenti(vedere sezione chiamata «Riconoscimenti» in Capitolo 8, *Gestire i Volontari*).

Avere in Manutenzione più di Una Linea di Release

La maggior parte dei progetti maturi fanno la manutenzione a più di una linea di release in parallelo. Per esempio, dopo che vien fuori la 1.0.0, quella linea dovrebbe continuare con le micro releases (di correzione bugs) 1.0.1, 1.0.2, ecc. Notare che il solo rilascio della 1.1.0 non è una ragione sufficiente per terminare la linea 1.0.x. Per esempio, alcuni utilizzatori hanno l'abitudine di non aggiornare mai alla prima release in una serie minore o maggiore—lasciano gli altri a rimuovere i bugs, per esempio dalla 1.1.0, e aspettano fino alla 1.1.1. Ciò non è necessariamente egoista (ricordate che essi stanno rinunciando alle correzioni di bugs e anche a nuove funzionalità); è solo che, per una qualsiasi ragione, hanno deciso di essere molto prudenti con gli aggiornamenti. Di conseguenza, se il progetto viene a conoscenza di un bug importante nella 1.0.3 giusto prima che stia rilasciando la 1.0.3, dovrebbe essere un pò rigido nel mettere appunto la correzione del bug nella 1.1.0 e a dire a tutti i vecchi utilizzatori della 1.0.x che dovrebbero aggiornare. Perché non rilasciare sia la 1.1.0 che la 1.0.4, in modo che ognuno sia felice?

Dopo che la linea 1.1.x è in cammino, potete dichiarare che la 1.0.x è alla *fine della vita*. Questo dovrebbe essere annunciato ufficialmente. L'annuncio dovrebbe essere unico, o dovrebbe essere menzionato come parte dell'annuncio della release 1.1.x; comunque fatelo, gli utilizzatori hanno bisogno di sapere che la vecchia linea sta venendo gradualmente eliminata, così che di conseguenza possano prendere la decisione di aggiornare.

Alcuni progetti stabiliscono un intervallo di tempo durante il quale si impegnano a supportare la linea di release precedente. In un contesto open source “supportare” significa accettare i reports di bugs su quella linea e creare release di manutenzione quando vengono trovati bugs significativi. Altri progetti non vi dedicano una quantità di tempo definita, ma tengono d'occhio i reports di bugs che arrivano per misurare quanta gente sta usando la vecchia linea. Quando la percentuale scende sotto un certo livello essi dichiarano la fine della vita per quella linea e smettono di supportarla.

Per ogni release assicuratevi di avere una *versione obiettivo* o *una pietra miliare obiettivo* nel tracciatore di bug, così la gente che archivia i bugs saprà fare così nei confronti della propria release. Non dimenticate nemmeno di avere un obiettivo “sviluppo” o “ultimo” per le più recenti fonti di sviluppo, poiché alcune persone non solo sviluppatori attivi rimangano avanti rispetto alla release ufficiale.

Le Releases di Sicurezza

La maggior parte dei dettagli sulla gestione dei bug sulla sicurezza era trattata in sezione chiamata «Annunciare le Vulnerabilità della Sicurezza» in Capitolo 6, *Comunicazione*, ma lì ci sono dettagli specifici per creare releases di sicurezza.

Una *release di sicurezza* è una release fatta solo per chiudere falle nella sicurezza. Il codice che corregge i bugs non può essere reso pubblico finché la release non è disponibile, il che significa non solo che i bugs non possono essere inviati al deposito fino al giorno della release, ma anche che la release non può essere testata pubblicamente prima che esca. Ovviamente gli sviluppatori possono esaminare i bugs fra di loro, e testare la release in privato, ma la diffusione nel mondo del testing reale non è possibile.

A causa di questa mancanza di testing, una release sulla sicurezza dovrebbe sempre essere fatta di alcune releases esistenti più le correzioni dei bugs di sicurezza, con *nessun altro cambiamento*. Questo perché più cambiamenti inviate senza testing, più probabilmente quell'unico fra essi causerà un nuovo bug, forse un nuovo bug di sicurezza. La prudenza è anche di famiglia per gli amministratori che possono aver bisogno di aprire le correzioni sulla sicurezza, ma la cui politica di aggiornamento preferisce di non aprire nessun altro cambiamento nello stesso tempo.

Il creare una release di sicurezza a volte comporta qualche inganno minore. Per esempio il progetto può aver lavorato alla release 1.1.3, con certe correzioni di bugs già pubblicamente dichiarate, quando arriva un report di bugs. Naturalmente gli sviluppatori non possono parlare di problemi si sicurezza fino a quando non rendono disponibile la correzione. Fino ad allora essi devono continuare a parlare pubblicamente come se la 1.1.3 sarà ciò che si è sempre pianificato che sia. Ma quando la 1.1.3 finalmente esce, differirà dalla 1.1.2 solo per le correzioni dei bugs sulla sicurezza, e tutte le altre correzioni saranno state rinviate alla 1.1.4 (che certamente ora conterrà *anche* le correzioni sulla sicurezza, come le conterranno tutte le releases future).

Potreste aggiungere un componente extra in una release esistente per indicare che essa contiene solo cambiamenti sulla sicurezza. Per esempio la gente dovrebbe saper dire, giusto dai numeri che la 1.1.2.1 è una release di sicurezza nei confronti della 1.1.2 e dovrebbero sapere che ogni release più “alta” di quella (per esempio, 1.1.3, 1.2.0, ecc...) contiene le stesse correzioni sula sicurezza. Per quelli ben informati, questo sistema porta un sacco di informazioni. D'altra parte, per quelli che non seguono il progetto da vicino, può essere causa di confusione vedere un numero di release a tre componenti la maggior parte delle volte con una release a quattro componenti inserita apparentemente a caso. La maggior parte dei progetti che ho visto scelgono la coerenza e semplicemente e scelgono il numero successivo regolarmente programmato per le releases di sicurezza, anche quando ciò significa far slittare di uno le altre releases programmate.

Le Releases e Lo Sviluppo Quotidiano

Il mantenimento di releases parallele simultaneamente ha ripercussioni su come vien fatto lo sviluppo quotidiano. In particolare ciò rende obbligatoria una disciplina che sarebbe raccomandata comunque: ottenere che ogni invio sia un singolo cambiamento logico, e non mescoli nello stesso invio cambiamenti non correlati. Se un cambiamento è troppo grande o troppo dirompente se fatto in un unico invio, dividetelo in N invii, dove ogni invio sia una ben ripartita sottosezione del cambiamento completo, e non includa nulla di non correlato al cambiamento completo.

Qui c'è un esempio di invio mal concepito:

```
------------------------------------------------------------------------
r6228 | jrandom | 2004-06-30 22:13:07 -0500 (Wed, 30 Jun 2004) | 8 lines
```

```
Correzione del problema #1729: Rebdere l'indicizzzazione gradevole
avvisa l'utente quando un file sta cambiando mentre viene indicizzato.

* ui/repl.py
  (ChangingFile): Nuova classe di eccezione.
  (DoIndex): Gestire la nuova eccezione.

* indexer/index.py
  (FollowStream): Far comparire la nuova eccezione se il file sta
   cambiando durente l'indicizzazione.
  (BuildDir): Senza correlazione, rimuovere i commenti obsoleti,
  riformattare del codice, e correggere la ricerca degli errori
  mentre si crea una directory.

Altre pulizie non correlate:

* www/index.html: Corregge alcuni refusi, fissa la data della nuova release.
----------------------------------------------------------------------
```

Il problema con esso diventa evidente non appena qualcuno ha bisogno di fare il port della `BuildDir` ricerca e correzione degli errori su un ramo per una release di manutenzione in arrivo. Chi fa il port non vuole nessuno degli altri cambiamenti per esempio la correzione al problema #1729 non è stata approvata affatto per il ramo di manutenzione e le modifiche del `index.html` sarebbero semplicemente irrilevanti qui. Ma egli non riesce ad afferrare il cambiamento alla `BuildDir` attraverso la funzionalità di unione dello strumento del controllo di versione, perché al sistema di controllo di versione era stato detto che il cambiamento è logicamente raggruppato con tutte quelle altre cose non correlate. Infatti, il problema dovrebbe diventare evidente anche prima dell'unione. Il semplice elencare i cambiamenti per il voto diventerebbe problematico: invece di dare solo il numero di revisione, il proponente dovrebbe fare una speciale patch o cambiare ramo per isolare la porzione di invio che viene proposta. Il che sarebbe un sacco di lavoro da sopportare per gli altri, e tutto perché chi fa l'invio originale non potrebbe essere seccato a suddividere le cose in gruppi logici.

In realtà quell'invio avrebbe dovuto essere *quattro* separati invii: uno per correggere il problema #1729, un altro per rimuovere i commenti obsoleti e riformattare il codice nella `BuildDir`, un altro per correggere la ricerca degli errori nella `BuildDir`, e infine uno per modificare l' `index.html`. Il terzo di questi invii sarebbe l'unico proposto per il ramo di mantenimento della release.

Certo, la stabilizzazione della release non è l'unica ragione per cui è desiderabile avere che ogni invio sia l'unico cambiamento logico. Psicologicamente un invio semanticamente unificato è più facile da revisionare, e più facile da riportare al punto di partenza se necessario (in alcuni sistemi di controllo della versione riportare al punto di partenza è effettivamente una speciale forma di unione, comunque). Un piccolo addestramento aperto sulla parte di ognuno può evitare al progetto un sacco di mal di testa più tardi.

Pianificare le Releases

Un campo in cui i progetti open source si sono differenziati dai progetti proprietari è la pianificazione delle releases. I progetti proprietari di solito hanno scadenze più decise. A volte ciò avviene perché i clienti sono stati avvisati che un aggiornamento sarebbe disponibile per una certa data, perché c'è bisogno che la nuova release sia coordinata con altri impegni di marketing, o perché il capitalista che rischia, che ha investito in tutta la cosa ha bisogno di vedere alcuni risultati prima di mettervi dentro altri finanziamenti. I progetti open source, invece, erano fino a tempi recenti in gran parte motivati da dilettantismo nel senso più letterale: erano scritti per amore di essi. Nessuno sentiva il bisogno di inviare

prima che tutte le funzionalità fossero pronte, e perché avrebbero dovuto? Non era incredibile che il lavoro di ognuno fosse senza indugi.

Al giorno d'oggi, molti progetti open source sono finanziati dalle compagnie, e sono di conseguenza influenzati dalla cultura della scadenza delle compagnie. Questo è sotto molti aspetti una buona cosa, ma può creare dei conflitti fra le priorità degli sviluppatori pagati e quelle di coloro che stanno offrendo volontariamente il loro tempo. Questi conflitti avvengono spesso sul come e quando programmare le releases. Gli sviluppatori salariati che sono sotto pressione vorranno naturalmente scegliere una data in cui avverrà la release e ottenere che l'attività di ognuno ci si allinei. Ma i volontari possono avere un'altra agenda forse funzionalità che vogliono completare o qualche prova che vogliono fare a riguardo —essi ritengono che la release dovrebbe aspettare.

Non c'è una soluzione che vada bene sempre a questo problema, tranne la discussione e il compromesso, certo. Ma potete rendere minimi la frequenza e i grado di attrito che si verifica, separando *la vita* che ci si ripropone per una data release dalla data in cui deve uscire. Cioè, cercare di indirizzare la discussione verso l'argomento di quali releases il progetto sta creando nel futuro a medio termine, e su quali funzionalità ci saranno in esse, senza prima fare nessuna menzione delle date, eccetto che approssimative congetture con largo margine di errore.[1]. Fissando prima gli insiemi di funzionalità, riducete la complessità delle discussione incentrata su una singola release, e quindi ne migliorate la prevedibilità. Questo crea anche una tendenza inerziale contro chiunque proponga di espandere la definizione della release aggiungendo nuove funzionalità e altre complicazioni. Se i contenuti della release sono del tutto ben definiti, è onere di chi propone giustificarne l'espansione, anche se la data della release non può essere ancora fissata.

Nella biografia multi volume di Thomas Jefferson, *efferson e il suo Tempo*, Dumas Malone racconta la storia di come Jefferson gestì il primo meeting tenutosi per decidere l'organizzazione della futura Università della Virginia. L'Università aveva avuto l'idea di Jefferson al primo posto, ma (come è nel caso di ogni luogo, non solo nei progetti open source, molti altri gruppi erano saliti sul palco rapidamente, ognuno coi propri interessi e agende. Quando essi si adunarono in quel primo meeting per parlare delle cose, Jefferson si assicurò con i disegni di una meticolosa architettura preparata, budgets dettagliati per la costruzione e per l'operatività, un programma di studi proposto, e i nomi delle specifiche facoltà che voleva importare dell'Europa. Nessun altro nella sala era così preparato anche lontanamente; e finalmente l'Università fu fondata più o meno in accordo con i suoi piani. Il fatto che la costruzione andò di molto oltre il budget, e molte delle sue idee invece no, per molte ragioni, si risolse alla fine, in cui Jefferson conosceva perfettamente le cose che sarebbero accadute. Il suo proposito fu strategico: mettersi in mostra nel meeting con qualcosa di così concreto che chiunque altro avrebbe fallito nella parte di proporre semplicemente modifiche ad esso, cosicché il disegno complessivo, e quindi il programma, sarebbe stato in modo grezzo quello che voleva.

Nel caso di un progetto di software libero non c'è un singolo “meeting”, ma invece una serie di piccole proposte fatte per la maggior parte per mezzo del tracciatore di problemi. Ma se avete un certa credibilità nel progetto per incominciare, e partite con l'assegnare varie funzionalità, accrescimenti, e bugs alla release obiettivo nel tracciatore di problemi, in accordo con quelche piano complessivo annunciato, la gente per la maggior parte andrà insieme a voi. Una volta che avete ottenuto che le cose siano sistemate più o meno come volete, la conversazione sulle *date* delle releases andrà avanti in modo più regolare.

E' cruciale, certo, non presentare mai una decisione individuale come scritta nella pietra. Nei commenti associati all'assegnamento di un problema a una specifica futura release, provocate una discussione, e siate genuinamente desiderosi di essere persuasi ogni volta che è possibile. Non esercitate mai il controllo con lo scopo di esercitare solamente il controllo: più a fondo gli altri parteciperanno

[1]Per un approccio alternativo potreste voler leggere la tesi di Martin Michlmayr's Ph.D. *Quality Il Miglioramento Della Qualità Nei Progetti Volontari di Software Open Source: Esplorazione dell'Impatto della Gestione Delle Releases* (http://www.cyrius.com/publications/michlmayr-phd.html). Si tratta dell'uso dei processi di rilascio basati sul tempo, in opposizione a quelli basati sulle funzionalità. Michlmayr fece anche un discorso presso Google sull'argomento, disponibile su Google video a http://video.google.com/videoplay?docid=-5503858974016723264.

al processo di pianificazione delle releases (vedere sezione chiamata «Suddividete i Compiti di Management e i Compiti Tecnici» in Capitolo 8, *Gestire i Volontari*), più facile sarà il persuaderli a condividere le vostre priorità sui problemi che contano veramente per voi.

L'altro modo in cui il progetto può smorzare le tensioni sulla pianificazione delle release è fare releases molto spesso. Quando c'è molto tempo fra le releases, l'importanza di ogni singola releases è amplificato nella mente di ognuno; le persone si sentono così tanto più annientate quando il loro codice non ce la fa ad entravi, perché sanno quanto tempo ci vorrà per il prossimo cambiamento. A seconda della complessità del processo di rilascio e della natura del vostro progetto, qualcosa come tre o sei mesi è il giusto intervallo di solito fra le releases, sebbene le linee di manutenzione possono far uscire micro releases un po' più spesso, se c'è bisogno di esse.

Capitolo 8. Gestire i Volontari

Riuscire a far si che le persone aderiscano a ciò di cui ha bisogno il progetto, e lavorare insieme per raggiungerlo, richiede più che la sola atmosfera cordiale e una mancanza della ovvie disfunzioni. Richiede qualcuno o più di qualcuno, che sappiano gestire tutte le persone coinvolte. Gestire i volontari può non essere un'arte tecnica nello stesso senso della programmazione di un computer, ma può essere un'arte che può essere migliorata con studio e pratica.

Questo capitolo è un afferrare e mettere in borsa tecniche specifiche per la gestione dei volontari. Esso afferra, forse più fortemente che i precedenti capitoli, con Subversion come caso di studio, perché io ho lavorato a quel progetto per scrivere questo ed ho avuto tutte le fonti primarie a portata di mano, e in parte perché è più accettabile lanciale pietre di critica nelle propria serra di vetro piuttosto che in quella di altri. Ma ho visto anche in molti altri progetti i benefici dell'applicare e le conseguenze del non applicare le raccomandazioni che seguono; quando sarà politicamente fattibile dare esempi provenienti da alcuni altri progetti, io lo farò.

Parlando di politiche, questo è un buon momento come nessuno per dilungarsi con quella molto malfamata parola per uno sguardo più da vicino. A molti ingegneri piace pensare alla politica come a qualcosa in cui la gente si imbarca. "*Io sto* solo sostenendo la causa di un miglior corso per il progetto ma *quello* sta sollevando obiezioni per ragioni politiche". Io credo che questa antipatia per la politica (o per quello che si immagina sia la politica) sia particolarmente forte negli ingegneri perché gli ingegneri sono acquisiti all'idea che alcune soluzioni siano oggettivamente superiori ad altre. Cosi, quando qualcuno agisce in modo da sembrare motivato da considerazioni esterne per esempio il mantenimento della sua posizione di influenza, la riduzione dell'influenza di qualche altro, il commercio aperto dei voti, o l'evitare di ferire la sensibilità di qualcuno gli altri partecipanti al progetto possono annoiarsi. Certamente, questo raramente impedisce loro di comportarsi nello stesso modo quando i loro interessi vitali sono in gioco.

Se ritenete “politica” una parola sporca, e sperate di mantenere il vostro progetto libero da essa, arrendetevi proprio ora. La politica è inevitabile ogni volta che la gente deve gestire collettivamente un risorsa condivisa. E' assolutamente razionale che una delle considerazioni che entra nel processo del prendere decisioni da parte di ognuno è la domanda su come una data azione può incidere sulla propria futura influenza nel progetto. Dopotutto, se avete fiducia nel vostro giudizio e nelle vostre capacità, come molti programmatori fanno, allora la possibile perdita di influenza di futura influenza deve essere considerato un risultato tecnico, in un certo senso. Simili ragionamenti si applicano ad altri comportamenti che potrebbero sembrare, nel loro aspetto, come “pura” politica. In effetti non c'è cosa pura politica come questa: è precisamente perché le azioni hanno conseguenze sul mondo reale che la gente diventa politicamente consapevole in primo luogo. La politica, in fin dei conti, è semplicemente una presa di coscienza che devono essere tenute in conto *tutte* le conseguenze delle decisioni. Se una particolare decisione porta al risultato che molti partecipanti trovano tecnicamente soddisfacente, ma comporta un cambiamento nei rapporti di potere che lascia che persone chiave si sentano isolate, il secondo un risultato importante proprio come il primo. Ignorarlo sarebbe di nobili sentimenti, ma miope.

Così, quando leggete il consiglio che segue, e quando lavorate con il vostro progetto personale, ricordate che non c'è nessuno al di sopra della politica. Apparire al di sopra della politica, è solamente una particolare strategia politica, e a volte è molto utile, ma non è mai la realtà. Politico è semplicemente ciò che avviene quando la gente è in disaccordo, e i progetti di successo sono quelli che sviluppano meccanismi politici per gestire costruttivamente i disaccordi.

Ottenere il Massimo dai Volontari

Perché lavorano dei volontari in progetti di software libero?[1]

Quando viene chiesto loro, molti dichiarano che lo fanno perché vogliono produrre buon software, o vogliono essere coinvolti personalmente nel correggere i bugs che a loro interessano. Ma queste ragioni, di solito, non sono tutta la storia. Dopotutto, sapreste immaginarvi un volontario che sta in un progetto, anche se nessuno ha detto una parola di apprezzamento sul suo lavoro, o lo ha ascoltato nelle discussioni? Certo no. Chiaramente le persone spendono tempo sul software libero per ragioni che appunto vanno oltre il desiderio astratto di produrre buon codice. Il capire le vere motivazioni dei volontari vi aiuterà a mettere le cose in modo da attirarli e a mantenerli. Il desiderio di produrre buon codice può esserci fra queste motivazioni, insieme con la sfida e il valore educativo del lavorare su problemi difficili. Ma gli uomini hanno un innato desiderio di lavorare con altri uomini, e di guadagnasi rispetto attraverso attività di collaborazione. I gruppi impegnati in attività di collaborazione devono elaborare norme di comportamento in modo che quello stato sia acquisito e mantenuto attraverso azioni che giovano agli obiettivi del gruppo.

Queste norme non nasceranno da se stesse. Per esempio, in qualche progetto—alcuni sviluppatori esperti sanno probabilmente farne il nome su due piedi —le persone a quanto pare ritengono che lo status si acquisti postando frequentemente e con ricchezza di parole. Essi non sono pervenuti a questa conclusione per caso. Ci sono arrivati perché sono gratificati dal dal fare intricate, lunghe trattazioni, indipendentemente dal fatto che ciò aiuti o meno il progetto. Poi ci sono alcune tecniche per creare un'atmosfera in cui le azioni per acquisire uno status sono azioni costruttive.

La Delega

DLa delega non è solo un modo per suddividere il carico di lavoro; esso è anche uno strumento politico e sociale. Considerate tutte le conseguenze di quando chiedete a qualcuno di fare qualcosa. Il più ovvio effetto è che, se accetta, lui fa il lavoro e voi no. Ma un'altra conseguenza è che egli fa sapere che voi avete avuto fiducia in lui nell'affidargli il compito. Inoltre, se avete fatto la richiesta in un forum pubblico, allora egli sa che gli altri nel gruppo sono anche al corrente di quella fiducia. Egli può aver la sensazione di una certa pressione ad accettare, il che significa che dovete chiedere in un modo che gli permetta di declinare gentilmente se non vuole realmente il lavoro. Se il compito richiede coordinazione con gli altri nel progetto, voi in realtà state chiedendo che egli diventi più coinvolto, obblighi di forma che non avrebbero potuto esserci in altre circostanze, e che forse diventano una forma di autorità in qualche sotto dominio del progetto. Il coinvolgimento aggiuntivo potrebbe spaventarlo, o potrebbe portarlo anche ad impegnarsi in altri modi, per un aumentato senso di impegno complessivo.

A causa di tutte queste conseguenze, spesso ha un senso chiedere a qualche altro di fare qualcosa anche quando sapete che potreste farlo più facilmente e velocemente voi stessi. Certo, c'è talvolta uno stringente argomento di efficienza economica relativamente a questa cosa comunque: forse il costo economico di farlo voi stessi sarebbe troppo alto ci potrebbe essere qualcosa di più importante che potreste fare in quel lasso di tempo. Ma anche quando non si applica l'argomento del costo economico, voi potete *ancora* voler chiedere a qualche altro di intraprendere il lavoro perché a lungo andare voi volete tirarlo più in profondità nel progetto, anche se ciò significhi spendere un tempo extra per aver cura di lui all'inizio. Si applica anche le tecnica inversa: se voi occasionalmente vi offrite volontari per fare qualcosa che qualche altro non vuole o non ha il tempo di fare, vi guadagnerete la sua buona volontà e il rispetto. La delega e la sostituzione non esistono solo per ottenere che un compito individuale sia portato a termine; esse esistono anche per coinvolgere più strettamente la gente nel progetto.

[1]Questa questione fu studiata in dettaglio, con interessanti risultati, in uno scritto di Karim Lakhani e Robert G. Wolf, dal titolo *Perché gli Hackers Fanno Ciò che Fanno: Comprendere Lo Sforzo e le Motivazioni nei Progetti Liberi/Open Source*. Vedere http://freesoftware.mit.edu/papers/lakhaniwolf.pdf.

Distinguere chiaramente fra richiesta e assegnazione

A volte è giusto aspettarsi che una persona accetterà un particolare compito. Per esempio, se qualcuno scrive un bug nel codice, o invia codice che non si conforma alle linee guida del progetto in qualche modo evidente, allora è sufficiente richiamare l'attenzione sul problema e quindi comportarsi come se diate per scontato che la persona vi farà attenzione. Ma ci sono altre situazioni in cui non è in nessun modo chiaro che che voi avete il diritto di aspettarvi l'effetto. La persona potrebbe fare come chiedete, oppure no. Poiché non c'è nessuno a cui piace essere preso senza argomenti, c'è bisogno che stiate attenti a questi due tipi di situazioni e fare su misura le vostre richieste di conseguenza.

Una cosa che quasi tutte le volte provoca una istantanea irritazione nella gente è il fatto che gli venga richiesto di fare qualcosa in un modo che presuppone che voi pensate che è chiaramente sua responsabilità farlo, quando essi la pensano diversamente. Per esempio l'assegnazione di un problema in arrivo è un terreno particolarmente fertile per questo tipo di irritazione. I partecipanti a un progetto di solito sanno chi è esperto in quei campi, così quando arriva un rapporto di bug, ci saranno spesso una o due persone di cui ognuno sa che sono capaci di correggerli. In ogni caso se voi assegnate il problema a una di queste due persone senza il loro previo assenso, egli potrà pensare di essere stato messo in una condizione scomoda. Egli sente la pressione di una aspettativa, ma può anche pensare che è stato punito per la sua esperienza. Dopotutto il modo in cui uno acquisisce esperienza è correggendo i bugs, così qualche altro porrebbe prendersi questo compito! (Notate che il tracciatore di bugs che assegna automaticamente i problemi a persone particolari basandosi sull'informazione esistente nel rapporto di bugs è meno probabile che faccia male, perché ognuno sa che le assegnazioni furono fatte da un processo automatico, e non è indice di aspettative umane.)

Mentre sarebbe simpatico suddividere il carico quanto più possibile in modo regolare, ci sono certe occasioni in cui voi volete giusto incoraggiare le persone che possono correggere un bug nella maniera più veloce possibile. Dato che non potete affrontare una ristrutturazione delle comunicazioni per ogni tale assegnazione (“Vorresti dare un'occhiata questo bug?” “Si” “Okay, ti sto per assegnare questo problema” “Okay”), dovreste fare l'assegnazione nella forma di una richiesta, non comunicando nessuna pressione. Virtualmente ogni tracciatore di problemi permette che un commento sia associato con l'assegnazione di un problema. In quel commento potete dire qualcosa del genere:

> Sto assegnando questo a te, perché tu hai la massima familiarità con questo codice. Sentiti libero di respingerlo se non hai il tempo di dargli un'occhiata, tuttavia. (E fammi sapere se non vorresti ricevere tali richieste in futuro.)

Ciò fa distinzione fra *richiesta* di assegnazione e *accettazione* da parte di chi la riceve, di quella assegnazione. Il pubblico qui non è solo chi fa l'assegnazione, è ognuno: l'intero gruppo assiste a una pubblica conferma dell'esperienza di chi riceve l'assegnazione, ma il messaggio rende anche chiaro che chi riceve l'assegnazione è libero di accettare o declinare la responsabilità.

Seguite dopo aver delegato

Quando chiedete a qualcuno di fare qualcosa, ricordatevi di averlo fatto e seguitelo, non importa in cosa. La maggior parte delle richieste vengono fatte in forums pubblici, e sono all'incirca della forma “Puoi prenderti cura di X?” Facci sapere in ogni caso; non c'è problema se non puoi, solo abbiamo bisogno di saperlo. Potete o non potete ricevere risposta. Se la ricevete e la risposta è negativa, il cerchio è chiuso non avete bisogno di altra strategia per trattare con X. Se c'è una risposta positiva controllate i progressi nel problema e commentate sul progresso che vedete o no (chiunque lavora meglio se sa che qualcuno sta apprezzando il suo lavoro). Se non c'è una risposta dopo pochi giorni, chiedete ancora, o postate dicendo che non avete ricevuto nessuna risposta e siete alla ricerca di qualcun altro che lo faccia. O fatelo proprio voi stessi, ma tuttavia assicuratevi di dire che non avete avuto risposta alla vostra richiesta.

Lo scopo del rendere noto il ritardo nella risposta *non* è quello di mortificare la persona, e la vostra osservazione dovrebbe essere messa nella forma tale da non avere quell'effetto. Lo scopo è semplicemente quello di far sapere che voi tenete traccia di quello che avete chiesto, e che rendete note le reazioni che ricevete. Questo fa si che le persone più probabilmente dicano si la prossima volta, perché essi osserveranno (anche se solo inconsciamente) che voi state probabilmente rendendo noto ogni lavoro che fanno, dato che avete reso noto il molto meno visibile evento che qualcuno ha mancato di rispondere.

Rendete noto ciò a cui la gente è interessata

Un'altra cosa che rende felice la gente è il fatto che vengano resi noti i loro interessi—in generale, più renderete noti e ricorderete gli aspetti della personalità di qualcuno, più egli si sentirà a suo agio, è più vorrà lavorare con un gruppo di cui voi fate parte.

Per esempio, c'era una accentuata differenza nel progetto Subversion, fra quelli che volevano raggiungere una release definitiva 1.0 (cosa che alla fine fecero), e chi voleva principalmente aggiungervi nuove funzionalità su interessanti problemi ma che non aveva cura di quando la 1.0 sarebbe uscita. Nessuna di queste posizioni è migliore o peggiore dell'altra; essi sono solo due differenti tipi di sviluppatori, e tutti e due fanno una gran quantità di lavori nel progetto. Ma noi imparammo velocemente il fatto di *non* dare per scontato che la loro eccitazione per quanto riguarda la guida della 1.0 fosse condivisa da tutti. I media elettronici possono essere molto ingannevoli: potete avere la sensazione di una finalità condivisa, quando, in effetti, essa è condivisa solo dalle persone con cui è vi successo di dover parlare, mentre altri hanno priorità completamente differenti.

Più siete al corrente di che tipo di persone vuole uscire dal progetto, tanto più efficacemente potete far richiesta di loro. Perfino solo il dimostrare una comprensione di cosa vogliono, senza fare nessuna richiesta associata, è utile, per il fatto che conferma a una persona che non è solo una particella in una massa indifferenziata.

Lode e Critica

Lode e critica non sono opposti; sotto molti aspetti essi sono simili. Sono sia una forma primaria cdi attenzione, e sono tantissimo efficaci quando sono specifici piuttosto che generici. Sia dovrebbero essere espressi con obiettivi concreti in mente. Sia possono essere sminuiti per inflazione: lodate troppo o troppo spesso, e svaluterete la vostra lode; lo stesso vale per la critica, sebbene in pratica, la critica provochi una reazione e quindi è un po' più resistente alla svalutazione.

Una importante caratteristica della cultura tecnica è che la critica dettagliata spassionata è spesso presa come una specie di lode (come discusso in sezione chiamata «Riconoscere la maleducazione» in Capitolo 6, *Comunicazione*), a causa della conseguenza che il lavoro di chi la riceve è apprezzato per il tempo richiesto per analizzarlo. Comunque ambedue le condizioni—*dettagliato* e *spassionato*—devono per questa persona risultare veri. Per esempio, se qualcuno fa un cambiamento trasandato al codice è inutile (e in realtà dannoso) fargli seguire una semplice frase del tipo “Questo era trasandato”. La trascuratezza è in fin dei conti una caratteristica della *persona*, non del suo lavoro, ed è importante mantenere le vostre reazioni focalizzate sul lavoro. E' molto più efficace descrivere tutte le cose sbagliate in quel cambiamento, tatticamente e senza malizia. Se questo è il terzo o il quarto cambio trasandato in una riga da parte delle stessa persona, è opportuno dirlo—di nuovo senza rabbia—alla fine della vostra critica, per rendere chiaro che il comportamento è stato notato.

Se qualcuno non si migliora in risposta alla critica, la soluzione non è più critica o più forte critica. La soluzione per il gruppo è rimuovere quella persona dalla posizione di non competenza, in un modo tale da le sensazioni di risentimento quanto più è possibile; vedere sezione chiamata «Gli avvicendamenti» più avanti in questo capitolo per gli esempi. Questa è una occorrenza rara comunque. La maggior parte della gente risponde molto bene alla critica che sia specifica, e che contenga una chiara (anche se non detta) aspettativa di miglioramento.

La lode non dovrebbe offendere i sentimenti di nessuno, certo, ma questo non significa che dovrebbe essere usata per nulla meno con attenzione della critica. La lode è uno strumento: prima di usarla, chiedetevi perché volete usarla. Come regola, non è una buona idea lodare la gente per cose che usualmente fa, o per azioni che sono una parte normale e prevista della partecipazione al gruppo. Se voi doveste farlo, sarebbe difficile sapere quando smettere dovreste lodare *ognuno* perché fa le cose normali? Dopotutto, se lasciate fuori qualcuno, egli si chiederà perché. E' molto meglio esprimere lode e gratitudine con parsimonia, in risposta a inattesi e insoliti sforzi, con l'intento di incoraggiare più sforzi di questi. Quando un partecipante sembra essesi mosso permanentemente in uno stato di alta produttività, adattate la vostra soglia di lode per quella persona in modo consono. Lodi ripetute per comportamenti normali diventano senza senso comunque. Invece, quella persona dovrebbe avvertire che il suo alto livello di produttività è ora considerato come normale e naturale, e solo il lavoro che vada oltre quel livello dovrebbe essere notato particolarmente.

Con questo non voglio dire che i contributi della persona non dovrebbero essere riconosciuti, certo. Ma ricordate che se il progetto è messo su bene, ogni cosa che quello persona fa è già visibile comunque, e così il gruppo saprà (e la persona saprà che che il resto del gruppo sa) ogni cosa che fa. Ci sono anche modi per dare un riconoscimento al lavoro che uno fa per mezzo di altro al posto di lodi dirette. Potete menzionare di passaggio, mentre discutete un argomento correlato, che egli ha fatto un sacco di lavoro in un dato campo ed è l'esperto locale lì; potete pubblicamente consultarlo circa qualche questione sul codice; o magari, più efficacemente, potete fare un massiccio uso ulteriore del lavoro che ha fatto, in modo che veda che gli altri sono a proprio agio nel contare sul risultato del suo lavoro. Probabilmente non è necessario fare queste cose in modo calcolato. Qualcuno che regolarmente dà ampi contributi in un progetto lo saprà, e occuperà una funzione influente senza che faccia nulla di proposito. Di solito non c'è bisogno di fare passi espliciti per assicurare ciò, a meno che voi non abbiate la sensazione, per una qualsiasi ragione, che un collaboratore è sottostimato.

Prevenire la Territorialità

Fate attenzione a che i partecipanti non cerchino di esercitare una proprietà esclusiva su certe aree del progetto, e a coloro che sembrano voler fare tutto il lavoro in quelle aree, fino al punto di assumere la direzione del lavoro che altri incominciano. Tale comportamento può sembrare anche sano all'inizio. Dopotutto in superficie egli sembra una persona che si prede più responsabilità, e mostra una attività maggiore un una data area. Ma a lungo andare, ciò è distruttivo. Quando la gente ha la percezione di un segnale di “non sconfinamento” si astiene. Il risultato è una ridotta revisione in quell'area, e di una maggiore fragilità, perché lo sviluppatore solitario diventa un punto di fallimento che provoca il fallimento dell'intero progetto. Peggio, ciò infrange la collaborazione, lo spirito egualitario del progetto. La teoria dovrebbe essere sempre quella che ogni sviluppatore deve essere il benvenuto nell'essere di aiuto in ogni operazione in ogni momento. Certo, in pratica le cose vanno un po' diversamente: le persona hanno aree in cui sono più o meno influenti, e i non esperti rinviano agli esperti in certi domini del progetto. Ma la chiave è che ciò sia tutto volontario: l'autorità informale è data per scontata se basata sulla competenza e sul giudizio dimostrato, ma non dovrebbe mai essere *acquisita* volontariamente. Anche se la persona che desidera l'autorità e effettivamente competente, è cruciale che eserciti quell'autorità informalmente, attraverso il consenso del gruppo, e che non sia la causa dell'escussione di altri dal lavoro in quel gruppo.

Respingere o modificare il lavoro di qualcuno, è una roba completamente differente. Lì, il fattore decisivo è il contenuto del lavoro, non chi è sembrato agire come custode. Può avvenire che una stessa persona sembra fare la maggior parte del lavoro di revisione per una data area, ma fino a quando egli non cerca di impedire a qualcun altro di fare anche lo stesso lavoro, le cose stanno probabilmente a posto.

Per combattere un incipiente territorialismo, o anche l'apparenza di esso, molto progetti hanno fatto il passo di bandire l'introduzione nei file sorgenti dei nomi degli autori e quelli dei manutentori designati. Io sono francamente d'accordo con questa pratica: noi la seguiamo nel progetto Subversion, ed è più o

meno la politica ufficiale nella Apache Software Foundation. Il membro della ASF la presenta in questo modo:

> *All'Apache Software Foundation noi scoraggiavamo l'uso dei tag author nel codice sorgente. C'erano molte ragioni per questo, oltre alle conseguenze legali. Lo sviluppo collettivo consiste nel lavorare sui progetti come gruppo e aver cura del progetto come gruppo. Dare riconoscimenti è giusto, e dovrebbe essere fatto, ma in un modo che non permetta false attribuzioni, anche per sottinteso. Non c'è una linea chiara su quando aggiungere o rimuovere un tag author. Aggiungete il vostro nome quando cambiate un commento? Quando inserite una correzione in linea. Rimuovete il tag di un altro autore quando cambiate un codice senza cambiare il risultato ed esso sembra al 95% differente? Che fate con persone che vanno in giro a toccare ogni file, a cambiare quanto basta per creare una quantità effettiva di tag, in modo che il loro nome appaia dappertutto?*
>
> *Ci sono modo migliori di dare riconoscimenti, e la nostra preferenza è di usare questi. Da un punto di vista tecnico i tag author non sono necessari; se volete trovare chi ha scritto un pezzo particolare di codice, può essere consultato il sistema del controllo di versione per capirlo. I tags author tendono anche ad essere antiquati. Volete veramente essere contattati in privato su un pezzo di codice che avete scritto cinque anni fa ed essere felici di averlo dimenticato?*

I files sorgente di codice di un progetto sono il cuore della sua identità. Essi dovrebbero rispecchiare il fatto che la comunità degli sviluppatori è nel complesso responsabile per essi, e non è divisa in piccoli feudi.

La gente talvolta parla in favore dei tags author o mantainer nei files sorgente sulla base del fatto che essi danno un riconoscimento visibile a quelli che hanno fatto la maggior parte del lavoro lì. Ci sono due problemi su questo argomento. Primo, i tags inevitabilmente sollevano la imbarazzante questione di quanto lavoro uno debba fare per vedere il suo nome elencato lì pure. Secondo, essi aggiungono il problema del riconoscimento a quello della paternità: aver fatto il lavoro in passato, non implica la paternità dell'aerea in cui il lavoro fu fatto, ma è difficile se non impossibile una tale conclusione quando i nomi individuali sono elencati in cima ai files sorgenti. In ogni caso l'informazione dei riconoscimenti può essere ottenuta dai logs del controllo di versione e per mezzo di altri meccanismi messi al-di-fuori-della-banda, come gli archivi delle mailing lists, in modo tale che non si perda nessuna informazione bandendola dai files sorgente stessi.

Se il vostro progetto decide di bandire i nomi individuali dai file sorgenti, cercate di non debordare. Per esempio, molti progetti hanno un'area `contrib/` in cui sono tenuti piccoli strumenti e scripts di aiuto, spesso scritti da gente che non è per altra via associata al progetto. E' bene che quei files contengano i i nomi degli autori, perché essi non sono effettivamente in manutenzione al progetto nel suo intero. D'altra parte, se uno strumento dato come contributo incomincia ad essere modificato da altre persone nel progetto, alla fine voi potete volerlo spostare in un una locazione meno isolata, e facendo il caso che l'autore originale approvi, rimuovere il nome dell'autore, di modo che il codice appaia come ogni altra risorsa in manutenzione da parte dalla comunità. Se l'autore è permaloso su questo, sono accettabili soluzioni di compromesso, per esempio:

```
# indexclean.py: Rimuoveere i vecchi dati da un indice Scanley.
#
# Autore originale: K. Maru <kobayashi@yetanotheremailservice.com>
# Ora mantenuto da: The Scanley Project <http://www.scanley.org/>
#                    and K. Maru.
#
# ...
```

Ma è meglio evitare tali compromessi, se possibile, e molti autori si stanno persuadendo, perché sono felici del fatto che si sta facendo del loro contributo una parte più integrale del progetto.

La cosa importante è ricordare che c'è una continuità fra il cuore e la periferia di ogni progetto. I principali files di codice sorgente del software sono chiaramente parte del cuore, e dovrebbero essere essere considerati in manutenzione da parte della comunità. D'altra parte, strumenti di accompagnamento o pezzi di documentazione possono essere il lavoro di singoli individui, che li hanno in manutenzione essenzialmente da soli, anche se i lavori possono essere associati, o anche distribuiti, dal progetto. Non c'è bisogno di applicare una regola di taglia unica a ogni file, finché vale il principio che non è permesso che le risorse in manutenzione alla comunità diventino territori personali.

Il Rapporto di Automazione

Cercare di non consentire agli uomini ciò che invece potrebbero fare le macchine. Come regola pratica, automatizzare una operazione comune vale dieci volte almeno lo sforzo che uno sviluppatore dovrebbe impiegare a fare manualmente quella operazione una volta. Per operazioni molto frequenti o molto complesse, il rapporto potrebbe arrivare facilmente a venti o anche di più.

Il pensare a voi stessi come “gestori di progetto”, piuttosto che solo come ad un altro sviluppatore, potrebbe essere un utile atteggiamento qui. A volte gli sviluppatori individuali sono tropo infagottati in lavori di basso livello per vedere il quadro grande e rendersi conto che ognuno sta sprecando un sacco di energie ad eseguire manualmente operazioni automatizzabili. Anche quelli che se ne rendono conto possono non avere il tempo di risolvere il problema: perché ogni esecuzione individuale dell'operazione non è percepita come un enorme carico, nessuno si secca abbastanza di fare qualcosa che la riguarda. Ciò che rende convincente l'automazione è che quel piccolo carico è moltiplicato per il numero di volte che ogni sviluppatore incorre in essa, e quindi quel numero è moltiplicato per il numero degli sviluppatori.

Qui sto usando il termine “automazione” in senso largo, per indicare non solo le ripetute azioni nelle quali una o due variabili cambiano ogni volta, ma ogni tipo di infrastruttura tecnica che assiste gli uomini. Il minimo standard di automazione richiesta per far girare un progetto in questi giorni è descritto in Capitolo 3, *L'Infrastruttura Tecnica* , ma ogni progetto può avere i suoi particolari problemi anche. Per esempio un gruppo che sta lavorando sulla documentazione, potrebbe volere un sito che mostri le più aggiornate versioni dei documenti in ogni momento. Siccome la documentazione è scritta in un linguaggio di markup come l'XML, ci può essere un passo della compilazione, spesso piuttosto intricato, relativo alla creazione di documenti che si possano esporre e che si possano scaricare. Adattare un sito in modo che tale compilazione avvenga automaticamente ad ogni invio può essere complicato e dispendioso come tempo impiegato ma ne vale la pena, anche se vi costa un giorno o più ad allestirlo. Il beneficio complessivo di avere pagine aggiornate in ogni momento è enorme, anche se il costo di *non* non averlo potrebbe sembrare solo un piccolo incomodo in ogni singolo momento, ad ogni singolo sviluppatore.

Fare tali passi non solo elimina tempo sprecato, ma l'oppressione e la frustrazione che ne deriva quando gli uomini fanno dei passi sbagliati (come li faranno inevitabilmente) nel cercare di portare a termine complicate procedure manualmente. Le operazioni dai molteplici passi, deterministiche sono ciò per cui i computers sono stati inventati; riservate agli uomini cose più interessanti.

Testing automatizzato

L'esecuzione di tests automatizzati sono utili per ogni progetto di software, perché il testing automatizzato (specialmente il testing di regressione) permette agli sviluppatori di sentirsi a proprio agio quando cambiano codice in aree con cui non hanno familiarizzato, e così incoraggiano lo sviluppo d'esplorazione. Poiché la ricerca dei guasti è difficile da fare a mano uno essenzialmente deve azzeccare dove potrebbe aver sbagliato qualcosa, e tentare vari esperimenti per provare il contrario—l'avere modi

automatizzati per individuare tali guasti risparmia al progetto *un sacco* di tempo. Ciò anche fa si che la gente sia più rilassata nel migliorare la leggibilità di larghe falciate di codice , e quindi contribuisce alla capacità di tenere in manutenzione il software nel lungo periodo.

Testing di Regressione

Testing di regressione significa fare delle prove per il riapparire di bug già corretti. Il proposito del testing di regressione è quello di ridurre le possibilità che i cambiamenti al codice rovinino il software in modo inatteso. Nella misura in cui il software diventa più grande e complesso, le possibilità di questi effetti collaterali aumentano regolarmente. Una buona progettazione può ridurre la velocità con la quale le possibilità crescono, ma ciò non può eliminare il problema completamente.

Come risultato molti progetti hanno *una suite di test*, un programma a parte che interroga il software del progetto nei modi che si sono conosciuti in passato per provocare determinati bugs. Se la suite di test ha successo nel far si che uno di questi bugs si verifichi, questo è noto come *regression*, col significato che i cambiamenti di qualcuno ha tolto la correzione a un bug precedentemente corretto.

Vedere anche http://en.wikipedia.org/wiki/Regression_testing.

Il testing di regressione non è una panacea. Tanto per dirne una, esso funziona molto bene con programmi con una interfaccia con comandi eseguiti in serie. Il software che è fatto funzionare con una interfaccia utente grafica è molto più difficile da far funzionare da programma. Un altro problema è che la struttura della swite per il test di regressione può spesso essere piuttosto complessa, con una curva di apprendimento e un carico di manutenzione che le sono peculiari. Ridurre questa complessità è una della cose più utili che possiate fare, anche se può richiedere un considerevole ammontare di tempo. La cosa più facile è aggiungere nuovi tests alla suite, più gli sviluppatori faranno così, più pochi bugs sopravviveranno nella release. Ogni sforzo fatto per rendere i tests più facili sarà ripagato molte volte durante vita del progetto.

Molti progetti hanno una regola *“Non sfasciate l'allestimento!”* ,che significa: non fare un invio che renda difficile la compilazione o l'esecuzione del software. Essere la persona che sfascia la costruzione è di solito causa di leggero imbarazzo e di burla. I progetti con una swite per i tests di regressione spesso hanno una regola corollario: non inviare cambiamenti che fanno fallire i tests. E' molto facile osservare questi fallimenti se ci sono esecuzioni automatiche notturne dell'intera suite di tests, con i risultati inviati alle mailing lists dello sviluppo, o a una mailing list dedicata di risultai del testing; questo è un altro esempio di automazione utile.

La maggior parte degli sviluppatori volontari sono disposti a spendere un tempo extra per scrivere tests di regressione, quando il sistema di test è comprensibile ed è facile lavorare con esso. Accompagnare i cambiamenti con tests è intesa come cosa responsabile da fare, ed è una facile opportunità di collaborazione: spesso due sviluppatori si divideranno il lavoro per la correzione di un bug, con uno che scrive la correzione stessa e uno che scrive il test. Il secondo sviluppatore può spesso beccarsi il maggior lavoro, e poiché scrivere un test è già meno soddisfacente che correggere realmente il bug, è imperativo che la suite di test non renda l'esperienza più stressante di quanto debba essere.

Alcuni progetti vanno anche oltre, richiedendo che un test accompagni *ogni* correzione di bug o ogni nuova funzionalità. Se questa sia una buona idea o no dipende da molti fattori: la natura del software, la composizione del team di sviluppo, e la difficoltà di scrivere nuovi tests. Il progetto CVS (http://www.cvshome.org/) ha a lungo avuto questa regola. E' una buona politica in teoria, poiché CVS è un sistema di controllo di versione e quindi molto avverso al rischio di rovinare o maltrattare i dati dell'utilizzatore. Il problema in pratica è che la suite di test di regressione di CVS è un enorme script

di shell (in modo divertente chiamato `sanity.sh`), difficile da leggere e difficile da modificare o estendere. La difficoltà di aggiungere nuovi tests, combinata col requisito che le patches siano accompagnate da nuovi tests, significa che CVS in effetti scoraggia le patches. Quando ebbi modo di lavorare a CVS, talvolta vidi la gente iniziare e anche completare le loro patches al proprio codice di CVS, ma arrendersi quando quando si diceva loro del requisito dell'aggiunta di un nuovo test a `sanity.sh`.

E' normale impiegare più tempo a scrivere un nuovo test di regressione che a correggere il bug originale. Ma CVS portò questo fenomeno all'estremo: uno poteva impiegare ore nel cercare di scrivere il suo test in modo appropriato, e ancora trovarsi in errore, perché ci sono appunto troppo imprevedibili complessità relative al cambiamento di uno script della shell di Bourne di 35.000 linee. Anche gli sviluppatori anziani di CVS si lamentavano quando dovevano aggiungere un nuovo test.

Questa situazione era dovuta a un nostro fallimento su tutta la linea nel considerare il rapporto di automazione. Sebbene il passare a una struttura di test reale—sia costruita personalmente sia dal di fuori, avrebbe richiesto uno sforzo maggiore.[2] Ma il non fare così è costato al progetto molto più, nel corso degli anni. Quante correzioni di bugs non ci sono in CVS oggi, a causa dell'ostacolo di una cattiva suite di test? Non non ne possiamo sapere l'esatto numero, ma è sicuramente molte volte più grande del numero di correzioni o di nuove funzionalità a cui gli sviluppatori potrebbero dover rinunciare per sviluppare un nuovo sistema di test (o integrare un sistema esterno). Quella operazione richiederebbe solo un finito ammontare di tempo, mentre la penalizzazione dell'uso della suite di test corrente continuerà per sempre se non si fa niente.

Il punto non è quello che avere avere requisiti stringenti per scrivere tests è male, né che scrivere il vostro sistema di test come lo script della shell di Bourne è necessariamente male. Il punto è semplicemente che quando il sistema di test diventa una significativo impedimento per lo sviluppo, qualcosa deve essere fatto. Lo stesso vale per ogni processo di routine che diventa una strettoia o una barriera.

Trattate Ogni Utilizzatore Come un Potenziale Volontario

Ogni interazione con un utilizzatore è un'opportunità per procurarsi un nuovo volontario. Quando un utilizzatore si concede il tempo di scrivere ad una delle mailing lists del progetto o di mettere un archivio un rapporto di bug, ha già indicato se stesso come possessore di un maggiore potenziale per un coinvolgimento rispetto alla maggior parte degli utilizzatori (dai quali il progetto non avrà mai notizie). Seguite questo potenziale: se egli ha descritto un bug, ringraziatelo per il report e chiedetegli de vuole correggerlo. Se egli ha scritto per dire che che una domanda importante manca nelle FAQ, o che la documentazione del programma è insufficiente in qualche modo, allora ammettete apertamente l'esistenza del problema (ammesso che esso esista realmente), e chiedetegli se è interessato a scrivere il materiale mancante. Naturalmente molte volte l'utilizzatore farà obiezione. Ma non costa molto chiedere, e ogni volta che lo fate, ciò ricorda agli altri ascoltatori in quel forum che essere coinvolti nel progetto è qualcosa che ognuno può fare.

Non limitate i vostri obiettivi ad acquisire nuovi sviluppatori e gente che scriva la documentazione. Per esempio, anche l'addestramento della gente a scrivere buoni rapporti di bugs dà buoni risultati a lungo andare, se non impiegate *troppo* tempo per persona, e se essi procedono ad inviare più report di bugs in futuro—cosa a cui sono molto propensi se ottengono una costruttiva reazione al loro primo rapporto. Una costruttiva reazione non è necessario che sia una correzione del bug, sebbene questo sia sempre l'ideale; essa può essere anche un sollecitazione per maggiori informazioni, o solamente la conferma che quel comportamento *è* un bug. La gente vuole essere ascoltata. Secondariamente essi vogliono che il loro bug sia corretto. Voi potrete non essere in grado di dare loro la seconda cosa in modo opportuno, ma (o il progetto intero) potete dare loro la prima cosa.

[2]Notare che lì non ci sarebbe bisogno di convertire tutti i tests esistenti alla nuova struttura; le due cose potevano esistere felicemente una a fianco a l'altra, con i vecchi tests convertiti solo se c'era bisogno che fossero cambiati.

Un corollario di questo è che gli sviluppatori non dovrebbero esprimere collera a persone che archiviano ben comprensibili ma vaghi report di bug. Questo è uno dei miei fastidi ricorrenti; vedo farlo su varie mailing lists open source, è il male che fa è palpabile. Alcuni sventurati principianti scriveranno un inuutile report:

> non mi riesce di prendere una Scanley da far girare. Ogni volta che parte, mi dà errore. Sta qualcun altro riscontrando lo stesso problema?

che hanno visto questo tipo di report centinaia di volte, e che non si sono trattenuti dal pensare che il principiante non li hanno visti centinaia di volte risponderanno all'incirca così:

> Cosa avete supposto di fare con così poca informazione? E' frustrante. Dateci almeno qualche dettaglio, come la versione di Scanley, il vostro sistema operativo e l'errore.

Lo sviluppatore ha mancato di vedere le cose dal punto di vista dell'utilizzatore, ed ha mancato anche di considerare l'effetto che una tale reazione potrebbe avere sul tutte le *altre* altre persone che stanno osservando lo scambio. Naturalmente un utilizzatore che non ha una esperienza di report di bugs, non saprà come scrivere un report di bug. Quale è il modo giusto di trattare questa persona? Educatelo! E fatelo in modo che egli ritorni per maggiori ragguagli:

> Mi dispiace che state avendo problemi. Abbiamo bisogno di maggiori informazioni per renderci conto di ciò che sta succedendo qui. Prego diteci la versione di Scanley, il vostro sistema operativo, e il testo esatto dell'errore. La miglior cosa che possiate fare è quella di inviare uno scritto che mostri gli esatti comandi che avete dato, e l'uscita che hanno prodotto. Vedere http://www.scanley.org/how_to_report_a_bug.html per maggiori ragguagli.

Questo modo di rispondere è molto più efficace per ottenere le necessarie informazione dagli utilizzatori, perché sono scritte dal punto di vista dell'utilizzatore. Primo, esso esprime simpatia: *Voi avevate un problema; non sentivamo il vostro disappunto.*. (Ciò non è necessario in ogni risposta a ogni rapporto di bug; dipende dalla serietà del problema e da quanto è sembrato sconvolto l'utilizzatore. Secondo, invece di disprezzarlo perché no sa come riportare un bug, gli dice come, e abbastanza in dettaglio per essere veramente utile per esempio molti utilizzatori non si rendono conto che “mostraci l'errore” significa “mostraci l'esatto testo dell'errore, senza omissioni o riassunti.” La prima volta che lavorate con tale utilizzatore, bisogna che siate precisi su questo. Alla fine ciò offre un puntatore a molte più dettagliate e complete istruzioni per l'invio dei rapporti sui bugs. Se avete stabilito con successo un contatto con l'utilizzatore, egli si prenderà spesso il tempo di leggere quel documento e fare ciò che dice. Ciò vuol dire che voi, certamente, dovete aver preparato quel documento in anticipo. Esso dovrebbe dare chiare istruzioni su che tipo di informazioni vuole vedere in ogni rapporto il vostro team di sviluppo. Idealmente, esso dovrebbe evolversi nel tempo in risposta a ai tipi pericolari di omissioni e di cattivi report che l'utilizzatore tende a a fare per il vostro progetto.

Le istruzioni sui report di Subversion sono un esempio standard alla lettera della forma (vedere Appendice D, *Istruzioni di Esempio per Segnalare un Bug*). Notate come essi rispondono ad un invito a fornire una patch per correggere il bug. Ciò non avviene perché un tale invito porterà a un rapporto patch/report più grande la maggior parte degli utilizzatori sono in grado di correggere bugs di cui già sanno che una patch sarebbe la benvenuta, e non hanno bisogno che gli venga detto. Il reale proposito di un invito è quello di porre l'accento per tutti gli utilizzatori, specialmente quelli nuovi nel progetto, o nuovi al software libero in generale, sul fatto che che il progetto va avanti con contributi di volontari. In un certo senso, gli attuali sviluppatori del progetto non sono più responsabili della correzione dei bugs di quanto non lo siano quelli che li hanno segnalati nei report. Questo è un punto importante con il quale molti nuovi utilizzatori non avranno familiarità. Una volta che se ne rendono conto essi probabilmente contribuiranno a fa si che la correzione al bug avvenga, se non contribuendo col codice allora fornendo un più completa ricetta per la riproduzione, o offrendosi per tests di bugs che le altre persone postano. L'obiettivo è far si che ogni utilizzatore si renda conto che non c'è una *innata* differenza fra se stessi e la gente che lavora al progetto è una questione di quanto tempo uno ci mette dentro, non di chi uno sia.

L'ammonizione a non rispondere in modo irritato non vale per gli utilizzatori sgarbati. Occasionalmente gli utilizzatori postano reports o rimostranze che, indipendentemente dal loro contenuto di informazione, mostrano un disprezzo con derisione nei confronti di alcune manchevolezze del progetto. Spesso tali persone alternativamente insultano e fanno complimenti, come la persona che postò alla mailing list di Subversion:

> Com'è che sono quasi 6 giorni che non ci sono ancora binari postati per la piattaforma di windows?!? E' la stessa storia ogni volta, ed è molto frustrante. Perché queste cose non sono automatizzate in modo che possano essere disponibili immediatamente?!?. Quando voi postate un allestimento “RC”, io penso che l'idea sia quella ch volete che gli utilizzatori testino l'allestimento, ma tuttavia non provvedete in nessun modo a fare così. Perché anche avete un lungo periodo testing se non fornite i mezzi per testare??

La risposta iniziale a questa infiammatorio post fu sorprendentemente contenuta: la gente fece notare che il progetto aveva un politica esplicita di non fornire binari, e disse, con vari gradi di irritazione, che sarebbe stato compito dei volontari di produrli essi stessi se erano tanto importanti per loro. Ci credete o no, il suo post successivo partiva con queste righe:

> Prima di tutto, lasciatemi dire che Subversion è fantastico, e io apprezzo veramente l'impegno di chiunque vi sia coinvolto. [...]

...e quindi ritornò a redarguire il progetto *di nuovo* perché non forniva i binari, mentre non ancora si offriva volontario per fare qualcosa a proposito. Dopodiché, circa 50 persone gli saltarono addosso, e non so dire se veramente ci feci caso. La politica di “zero tolleranza” verso la maleducazione sostenuta in sezione chiamata «Stroncate sul Nascere la Scortesia» in Capitolo 2, *Partenza* si applica a persone con cui il progetto ha una prolungata interazione (o vorrebbe averla). Ma quando qualcuno rende chiaro dall'inizio che sta diventando una fontana di bile, non vale la pena di farlo sentire il benvenuto.

Tali situazioni sono fortunatamente piuttosto rare, e sono notevolmente rare in progetti e che fanno uno sforzo per ingaggiare utilizzatori con cortesia e costruttivamente sin dalla loro prima interazione.

Suddividete i Compiti di Management e i Compiti Tecnici

Suddividete i Compiti di Management e i Compiti Tecnici. Suddividere il carico del management così come il carico tecnico del mandare avanti il progetto. Nella misura in cui il progetto diventa più complesso, sempre più il lavoro è quello di amministrare la gente e il flusso delle informazioni. Non c'è motivo per non suddividere quel carico, e la suddivisione non richiede una gerarchia dall'alto in basso —ciò che avviene in pratica tende ad essere più tipico della topologia di una rete peer to peer che della struttura di un comando stile militare.

A volte i ruoli del management sono formalizzati, e a volte si verificano spontaneamente. Nel progetto di Subversion, noi avevamo un manager delle patch, un manager delle traduzioni, un manager della documentazione, un manager dei problemi (sebbene non ufficiale) e un manager di release. Per dare avvio ad alcuni di questi ruoli non prendemmo una decisione consapevole, per altri avvenne che i ruoli avessero inizio da sé; nella misura in cui il progetto cresce, mi aspetto che si aggiungeranno altri ruoli. Qui di seguito esamineremo alcuni questi ruoli, e una coppia di altri, in dettaglio (eccetto il manager di release che è stato trattato già in sezione chiamata «Il manager di release» e sezione chiamata «Dittatura Da Parte del Proprietario Della Release» precedentemente in questo capitolo).

Quando leggete la descrizione del ruolo, notate che nessuno di essi richiede il controllo esclusivo sul dominio in questione. Il manager dei problemi non impedisce ad altre persone di fare cambiamenti nel database dei problemi, il manager delle FAQ non insiste sul fatto di essere la sola persona che redige le FAQ, e così via. Questi ruoli consistono tutti nella responsabilità senza il monopolio. Una parte

importante del lavoro del manager di ciascun dominio è quella di prender nota quando altre persone stanno lavorando in quel dominio, e trascinare loro a fare le cose nel modo in cui le fa il manager, in modo che gli sforzi multipli si rafforzino piuttosto che andare in conflitto. I managers di dominio dovrebbero anche documentare i processi con i quali essi fanno il loro lavoro, così che quando uno lascia qualcun altro possa colmare la mancanza subito.

A volte c'è un conflitto: due o più persone vogliono lo stesso ruolo. Non c'è una via giusta per gestire questa cosa. Potreste suggerire ad ogni volontario di postare una proposta (una “applicazione”) e ottenere che ogni persona che fa l'invio voti su chi è il migliore. Ma ciò è scomodo e potenzialmente pericoloso. Io trovo che una tecnica migliore sia quella di chiedere ai vari candidati di sistemare la cosa fra loro stessi. Essi, di solito, saranno più soddisfatti del risultato, che se la decisione fosse stata imposta dal di fuori.

Il Manager delle Patch

In un progetto si software libero che riceve un sacco di patch, tener traccia di quali patch sono arrivate, e cosa si è deciso su esse può essere un incubo, specialmente se lo si fa un modo decentralizzato. La maggior parte delle patch arrivano come posts alla mailing list di sviluppo (sebbene alcune possano apparire nel tracciatore di bug, o su siti esterni), e quindi ci sono un numero di differenti itinerari che la patch può percorrere dopo l'arrivo.

A volte qualcuno revisiona le patch, trova i problemi, e li rimanda all'autore originario per la ripulita. Ciò, di solito, porta a un processo iterativo tutto visibile sulla mailing list in cui l'autore originario posta le versioni revisionate della patch fino a quando il revisore non ha nient'altro da criticare. Non è sempre facile dire quando il processo è terminato: se il revisore fa l'invio della patch, allora chiaramente il ciclo è completo. Ma se non lo fa, potrebbe essere semplicemente perché non ha tempo, o perché non ha l'accesso all'invio e non potrebbe unirsi alla cordata degli altri sviluppatori nel farlo.

Un'altra frequente risposta a una patch è una discussione a ruota libera, non necessariamente sulla patch stessa, ma sul fatto se il concetto che sta dietro la patch è buono. Per esempio, la patch può correggere un bug, ma il progetto preferisce correggere quel bug in un altro modo, come parte della risoluzione di una classe più generale di problemi. Spesso questa non è nota in anticipo, ed è la patch che ne stimola la scoperta.

Occasionalmente, una patch postata è accolta con assoluto silenzio. Ciò, di solito, è dovuto al fatto che *al momento* nessuno sviluppatore ha il tempo di revisionare la patch. Poiché non c'è un limite particolare per quanto riguarda il tempo che ogni persona aspetta che qualcun altro raccolga la palla, e nel frattempo altre priorità stanno arrivando, è molto facile che una patch sfugga tra le crepe senza che una singola persona abbia intenzione che avvenga. Il progetto potrebbe perdere una utile patch in questo modo, e ci sono anche altri dannosi effetti collaterali anche: ciò è scoraggiante per l'autore, che ha impiegato lavoro per la patch, e fa apparire il progetto nell'insieme come se abbia perso i contatti, specialmente agli altri che stanno prendendo in considerazione la scrittura di patch.

Il lavoro del manager di patch è quello di assicurare che le patch non “scivolino fra le crepe”. Ciò si ottiene seguendo ogni patch attraverso una sorta di stato stabile. Il manager di patch esamina ogni discussione della mailing list che consegua ad un post di patch. Se essa finisce con un invio, egli non fa niente. Se essa va in un' iterazione revisione/correzione, che termina con una versione finale della patch senza che ci sia l'invio, egli archivia un problema che punti alla versione finale, e alla mailing lista che tratta di esso, di modo che ci sia una registrazione permanente che gli sviluppatori possono seguire in seguito. Se la patch si indirizza ad un problema esistente, egli annota il problema con ricche informazioni, invece di aprire un nuovo problema.

Quando una patch non riscuote per niente una reazione, il manager di patch aspetta pochi giorni, quindi dà seguito alla cosa chiedendo se qualcuno sta per revisionarla. Questo, di solito, riceve una reazione: uno sviluppatore può spiegare che non pensa che la patch debba essere applicata, e ne dà le ragioni,

o può revisionarla, nel qual caso viene fatto uno dei precedenti percorsi. Se non c'è ancora risposta, il manager di patch può o non può archiviare un problema per la patch, a sua discrezione, ma almeno chi originariamente ha fatto l'invio ha ricevuto *qualche* reaction.

L'avere un manager di patch ha salvato il team di sviluppo di Subversion un sacco di volte, ed fatto risparmiare energie mentali. Senza una persona designata che si prenda la responsabilità, ogni sviluppatore avrebbe da preoccuparsi continuamente “Se non ho il tempo di rispondere a questa patch subito, posso contare sul fatto che qualche altro lo faccia? Dovrei cercare di dargli un'occhiata. Ma se altre persone stanno anche tenendola d'occhio, per le stesse ragioni, noi avremmo inutilmente duplicato lo sforzo.” Il manager di patch rimuove la seconda congettura dalla situazione. Ciascuno sviluppatore può prendere la decisione giusta per lui dal primo momento che vede la patch. Se vuole dargli seguito con una revisione, può farlo il manager di patch adatterà il suo comportamento di conseguenza. Se vuole ignorare la patch completamente, andrà anche bene; il manager di patch dovrà assicurarsi che essa non sia dimenticata.

Siccome il sistema funziona solo se la gente può far conto sul fatto che il manager di patch sia là senza errore, il ruolo dovrebbe essere detenuto formalmente. In Subversion noi facemmo richiesta per esso mediate annuncio pubblicitario sulla mailing list dello sviluppo e degli utilizzatori, raccogliemmo molti volontari, e prendemmo il primo che ci rispose. Quando quella persona dovette ritirarsi (vedere sezione chiamata «Gli avvicendamenti» più avanti in questo capitolo), facemmo di nuovo la stessa cosa. Non cercammo mai di avere più persone ad detenere in comune il ruolo a causa delle informazioni addizionali che sarebbero state richiesta fra loro, ma forse a un volume molto alto di invii di patch, potrebbe aver senso un manager di patch a più teste.

Il Manager delle Traduzioni

Nei progetti di software “traduzione” può riferirsi a due cose molto differenti. Può significare tradurre la documentazione del software in altre lingue, o può significare tradurre il software stesso cioè ottenere le segnalazioni dei errore o dei messaggi di aiuto in altre lingue preferite dall'utente. Ambedue sono complesse operazioni, ma una volta che la giusta infrastruttura è allestita, esse sono largamente separabili dall'altro sviluppo. Poiché le operazioni sono simili in qualche modo, ha senso (a seconda del vostro progetto), avere un solo manager delle traduzioni che le gestisca ambedue, o può essere meglio avere due differenti manager.

Nel progetto di Subversion noi avevamo un manager di traduzione che gestiva ambedue le cose. Egli non deve scrivere le traduzioni egli stesso, certo, egli può dare una mano a uno o due, ma mentre questo scrive, egli dovrebbe aver bisogno di parlare dieci lingue (venti contando i dialetti) per lavorare a tutte le traduzioni. Invece, egli gestisce dei team di traduttori volontari: egli li aiuta a coordinarsi fra loro, e coordina i team fra loro e il resto del progetto.

Una parte delle ragioni per cui il manager delle traduzioni è necessario è che i traduttori sono una entità demografica differente da quella degli sviluppatori. Essi a volte hanno qualche o nessuna esperienza nel lavorare con deposito del controllo di versione, o certamente proprio nel lavorare come parte del team di volontari distribuito. Ma sotto altri aspetti essi sono spesso la miglior specie di volontari: persone proprio con una specifica conoscenza del dominio che videro una necessità e scelsero di essere coinvolti. Essi di solito sono desiderosi di imparare, ed entusiasti di mettersi al lavoro. Tutto ciò di cui hanno bisogno è uno che gli dica come. Il manager delle traduzioni assicura che la traduzione avvenga in modo da non interferire senza necessità col regolare sviluppo. Egli anche funziona come come sorta di rappresentanza dei traduttori come corpo unificato, ogni volta che gli sviluppatori devono essere informati di cambiamenti tecnici richiesti per supportare lo sforzo di traduzione.

Così, le abilità più importanti della posizione sono diplomatiche, non tecniche. Per esempio, in Subversion noi avevamo la politica che tutte le traduzioni dovevano avere almeno due persone a lavorarvi, perché altrimenti non c'è modo di revisionare il testo. Quando un nuovo volontario si offre per tradurre Subversion in, diciamo, Malgascio, il manager delle traduzioni deve agganciargli qualcuno che

postò sei mesi prima esprimendo interesse a tradurre in Malgascio, o anche politicamente chiedere al volontario di andare a trovare *un altro* traduttore in Malgascio per lavorare come suo partner. Una volta che abbastanza persone sono disponibili, il manager li sistema per un proprio tipo di accesso all'invio, li informa delle convenzioni del progetto (come per esempio scrivere i messaggi di log), e quindi tiene un occhio ad assicurarsi che essi aderiscano a queste convenzioni.

Le conversazioni fra il manager delle traduzioni e gli sviluppatori, o fra il manager delle traduzioni e i team delle traduzioni, sono di solito tenute nel linguaggio originale del progetto cioè la lingua da cui tutte le traduzioni sono fatte. Per molti progetti di software libero, questa è l'inglese, ma non importa quale sia fino a quando il progetto è d'accordo su ciò. (L'inglese comunque è probabilmente il meglio per progetti che vogliono attrarre una larga comunità internazionale di sviluppatori).

Le conversazioni *all'interno* di un particolare team di traduzione avvengono nella loro lingua comune, comunque, è uno dei compiti del manager delle traduzioni quello di mettere su una mailing list dedicata per ogni team. In questo modo i traduttori possono discutere il loro lavoro liberamente, senza distrarre la gente su altre liste principali, la maggior parte delle quali non sarebbero in grado di capire il linguaggio di traduzione, comunque.

Internazionalizzazione Verso Localizzazione

Internazionalizzazione (*I18N*) and *localizzazione* (*L10N*) si riferiscono ambedue all'adattare il programma in modo che funzioni in ambienti linguistici e culturali diversi da quello in cui il programma fu originariamente scritto. I termini sono spesso trattati come interscambiabili, ma in realtà essi non sono per nulla la stessa cosa. As http://en.wikipedia.org/wiki/G11n writes:

> La distinzione fra di esse è sottile ma importante. L'internazionalizzazione è l'adattamento per il suo *possibile* uso virtualmente ovunque, mentre la localizzazione è l'aggiunta di funzionalità per l'uso in un ambiente *specifico*

Per esempio, cambiare il vostro software in un sistema di compressione senza perdite di codifica testo Unicode (http://en.wikipedia.org/wiki/Unicode) è uno spostamento di internazionalizzazione, perché non riguarda una particolare lingua, ma piuttosto l'accettazione di testo da un qualsiasi numero di lingue. D'altra parte, facendo si che il vostro software stampi tutti i messaggi d'errore in Sloveno, quando esso accerta che sta girando in ambiente sloveno, la sua localizzazione si sposta.

Così il compito del manager delle traduzioni riguarda principalmente la localizzazione, non l'internazionalizzazione.

Il Manager della Documentazione

Il tenere aggiornata la documentazione è un compito senza fine. Anche nuove funzionalità o miglioramenti che entrano nel codice hanno la possibilità di causare un cambiamento nella documentazione. Anche, una volta che la documentazione del progetto raggiunge un certo livello di completezza, voi troverete che un sacco di patch che la gente manda sono per la documentazione, non per il codice. Questo perché ci sono molte più persone competenti a correggere i bug nella prosa più che nel codice: tutti gli utilizzatori sono lettori, ma solo pochi sono programmatori.

Le patch sulla documentazione sono di solito molto più facili da revisionare e da applicare che le patch sul codice. C'è poco testing o nessun testing da fare, e la qualità dal cambiamento può essere valutata rapidamente giusto con una revisione. Poiché la quantità è alta, ma il carico di revisione abbastanza basso il rapporto informazioni addizionali amministrative-lavoro produttivo è più grande per le patch di documentazione di quello delle patch di codice. Inoltre, la maggior parte delle patch avranno probabilmente la necessità di qualche ritocco per mantenere una coerenza di voce d'autore nella

documentazione. In molti casi le patch si sovrapporranno o intaccheranno altre patch, e necessiteranno di essere ritoccate una rispetto all'altra prima di essere inviate alla mailing list a e al deposito.

Date le esigenze di gestire le patch sulla documentazione, e il fatto che il codice base ha bisogno di essere monitorato, in modo che la documentazione sia aggiornata, ha senso avere una persona, o un piccolo team, dedicato al compito. Essi possono tenere una registrazione di come e dove esattamente la documentazione resta indietro al software, ed possono avere delle procedure collaudate per gestire grandi quantità di patch in un modo integrato.

Certo, questo non preclude ad altra gente nel progetto di applicare patch di documentazione al volo, specialmente quelle piccole, quando il tempo lo permette. E lo stesso manager di patch (vedere sezione chiamata «Il Manager delle Patch» prima in questo capitolo) può tener traccia sia di patch del codice si di patch di documentazione, archiviandole dove i team della documentazione e il team di sviluppo rispettivamente vogliono. (Se la quantità totale di patch supera la capacità di una persona di tenerne traccia, tuttavia, passare a separati manager di codice e documentazione, è probabilmente un buon primo passo. Il punto del team della documentazione è avere persone che si ritengano responsabili del mantenere la documentazione organizzata, aggiornata, e coerente. In pratica, ciò significa conoscere la documentazione profondamente con l'osservare in codice base, osservare i cambiamenti che *gli altri* inviano alla documentazione, osservare le patch di documentazione che arrivano, e usare tutte queste sorgenti di informazioni per mantenere in salute la documentazione.

Il Manager di Problemi

Il numero dei problemi nel tracciatore di bug del progetto cresce in proporzione al numero di persone che usano il software. Quindi, anche se correggete i bug e sistemate un programma sempre più robusto, vi dovreste aspettare che tuttavia che il numero di problemi aperti cresca essenzialmente senza limiti. La frequenza di problemi duplicati anche cresce, come crescerà la frequenza di problemi descritti in modo incompleto e con pochi particolari.

I manager dei problemi sono di aiuto nell'alleviare questi problemi con l'osservazione di ciò che va nel database, facendovi periodicamente la rivista per vedere se ci sono problemi specifici. Il loro atto più comune probabilmente è correggere i problemi che arrivano, o perché chi ha fatto il report non riempì correttamente alcuni campi del form, o perché il problema è un duplicato di uno esistente nel database. Ovviamente, più un manager di problemi è familiare con il database dei bug del progetto, con più efficienza sarà capace di trovare i problemi duplicati. Questo è uno dei principali vantaggi di avere poche persone che si specializzino nel database dei bug, invece che chiunque cerchi di farlo *ad hoc*. Quando il gruppo cerca di farlo in maniera decentralizzata, nessun singolo acquisisce una profonda esperienza nel contenuto del database.

I manager di problemi possono servire da mappa fra i problemi e i singoli sviluppatori. Quando ci sono un sacco di reports di bug che arrivano, non tutti gli sviluppatori possono leggere le mailing lists di notifica dei problemi con uguale attenzione. Comunque, se qualcuno che conosce il team degli sviluppatori sta tenendo d'occhio tutti i problemi che arrivano, allora può con discrezione dirigere l'attenzione di certi sviluppatori verso specifici bugs quando è opportuno. Certo, questo deve essere fatto con sensibilità verso chiunque altro vada avanti nello sviluppo, e ai desideri e al temperamento del destinatario. Quindi, è spesso la miglior cosa che il manager di problemi sia uno sviluppatore egli stesso.

A seconda di come il vostro progetto usa il tracciatore di bug, il manager di problemi può anche modellare il database in modo da riflettere le priorità del progetto. Per esempio, in Subversion, noi programmavamo i problemi in relaeses future specifiche, in modo che quando qualcuno chiedeva “Quando sarà corretto il bug X?, noi eravamo in grado si rispondere “Fra due releases”, anche se non gli potevamo dare la data esatta. Le releases sono rappresentate nel tracciatore di problemi come pietre miliari obiettivo, un campo disponibile in IssueZilla[3] As a rule, every Subversion release has one major

[3]IssueZilla è il tracciatore di bug che usiamo noi; esso è un discendente di BugZilla.

new feature and a list of specific bug fixes. We assign the appropriate target milestone to all the issues planned for that release (including the new feature—it gets an issue too), so that people can view the bug database through the lens of release scheduling. These targets rarely remain static, however. As new bugs come in, priorities sometimes get shifted around, and issues must be moved from one milestone to another so that each release remains manageable. This, again, is best done by people who have an overall sense of what's in the database, and how various issues relate to each other.

Un'altra cosa che i manager di problemi fanno è segnalare quando i problemi diventano obsoleti. A volte un bug è corretto accidentalmente come parte di in cambiamento al software non correlato, o a volte il progetto cambia la sua mentalità su fatto che un certo comportamento sia un errore. Trovare i problemi obsoleti non è facile: il solo modo di farlo sistematicamente è di fare una spazzata a tutti i problemi nel database. Intere spazzate diventano sempre meno fattibili col tempo, nella misura in cui cresce il numero dei problemi. Dopo un certo limite, il solo modo di mantenere in salute il database è quello di usare un approccio dividi-e-conquista: classificare immediatamente i problemi all'arrivo e dirigerli all'attenzione dello sviluppatore o del team. Il destinatario allora si prende carico del problema per il resto della sua vita, custodendolo verso la risoluzione o verso l'oblio se necessario. Quando il database è così grande, il manager dei problemi diventa più che un coordinatore complessivo, spedendo meno tempo a guardare ad ogni problema da sé, e più tempo a metterlo nelle mani della persona giusta.

Il Manager delle FAQ

La manutenzione delle FAQ è sorprendentemente un problema difficile. Diversamente dalla maggior parte dei documenti in un progetto, il cui contenuto è pianificato in anticipo dagli autori, una FAQ è un documento del tutto reattivo (vedere Manutenzione di una sezione FAQ). Non importa quanto grosso diventi, voi tuttavia non sapete quale sarà la nuova aggiunta. E poiché esso e costruito pezzo per pezzo, è molto facile che il documento nella sua interezza diventi incoerente e disorganizzato, e contenente anche voci duplicate o semi duplicate. Anche quando non ha nessun ovvio problema come questi, ci sono spesso interdipendenze non notate fra le voci link che dovrebbero essere creati e non lo sono perché la relativa voce fu inserita un anno lontano.

Il ruolo del manager delle FAQ è duplice. Primo, egli mantiene la qualità complessiva delle FAQ in quanto rimanere familiare con almeno gli argomenti di tutte le domande in esse, in modo che quando la gente aggiunge nuove voci che sono un duplicato o sono correlate alle voci esistenti, possa essere fatto l'appropriato adattamento. Secondo, egli osserva la mailing list del progetto e gli altri forum per altri problemi o domande e per scrivere nuove voci delle FAQ basate su queste informazioni. Questo secondo compito può essere piuttosto complesso: uno deve essere capace di seguire un argomento, riconoscere le domande base sollevate in esso, postare una voce nuova nelle FAQ, incorporarvi i commenti da parte di altri (poiché è impossibile che il manager delle FAQ essere un esperto in ogni argomento trattato nelle FAQ), e avvertire quando il processo è finito in modo che la voce sia alla fine aggiunta.

Il manager delle FAQ diventa anche l'esperto naturale nel formattare le FAQ. Ci sono un sacco di piccoli dettagli coinvolti nel tenere una FAQ nella forma giusta (vedere sezione chiamata «Trattate Tutte le Risorse Come un Archivio» in Capitolo 6, *Comunicazione*); quando gente a caso modifica le FAQ, a volte dimentica alcuni di questi dettagli. Questo va bene finché il manager delle FAQ è li a pulire dopo di loro.

Sono disponibili vari software per essere di aiuto nella manutenzione delle FAQ. E' bene fare uso di essi, finché non compromettono la qualità delle FAQ, ma guardatevi dal sovra-automazione. Alcuni progetti cercano di automatizzare completamente il processo di manutenzione delle FAQ, permettendo a chiunque di contribuire e modificare le voci delle FAQ in modo simile alle wiki (vedere sezione chiamata «Wiki» in Capitolo 3, *L'Infrastruttura Tecnica*). Ho visto che questo accade particolarmente con le Faq-O-Matic (http://faqomatic.sourceforge.net/), sebbene può essere che che i casi che ho visto fossero semplici abusi che andavano oltre ciò per cui le Faq-O-Matic erano state concepite. In ogni caso, mentre la decentralizzazione completa della manutenzione delle FAQ riduce il carico di lavoro

per il progetto, ha come risultato delle FAQ più trasandate. Non c'è una persona con una larga vedute di tutte le FAQ, nessuno che avvisa quando certe voci necessitano di aggiornamento o diventano obsolete completamente, e nessuno che osservi le interdipendenze tra le voci. Il risultato sono delle FAQ che spesso non riescono a fornire agli utilizzatori quello che stanno cercando, e nei casi peggiori li ingannano. Usate qualunque strumento di cui avete bisogno per fare la manutenzione alle vostre FAQ, ma non permettete mai alla convenienza degli strumenti di sedurvi a compromettere la qualità delle FAQ.

Vedere l'articolo di Sean Michael Kerner, *Le FAQ sule FAQ*, a http://osdir.com/Article1722.phtml, per la descrizione e la valutazione degli strumenti di manutenzione delle FAQ.

Gli avvicendamenti

Di volta in volta un volontario la cui responsabilità cresce (per es. il manager delle patch, il manager delle traduzioni, ecc..) diventeranno incapaci di portare a termine gli obblighi della posizione. Ciò può avvenire perché il lavoro è diventato più pesante di quanto egli si fosse prefigurato, o può essere dovuto a fattori completamente esterni: il matrimonio, un nuovo figlio, un nuovo datore di lavoro, o qualunque altra cosa.

Quando un volontario è sommerso così, di solito non lo nota subito. Avviene per lenti gradi, e non c'è un punto nel quale egli si rende conto che non può più portare avanti gli obblighi del ruolo. Invece, il resto del progetto non sente proprio nulla da lui per qualche tempo. Allora ci può essere una ventata di attività improvvisa, mentre egli si sente colpevole per aver ignorato il progetto per così lungo tempo, e si mette da parte una notte per recuperare. Allora voi non sentirete di lui ancora per una altro tempo, e allora ci potrebbe o non potrebbe essere un'altra ventata. Ma c'è raramente una dimissione formalmente non sollecitata. Il volontario stava facendo il suo lavoro nel tempo libero, così le dimissioni significherebbero rendere noto apertamente a se stesso che il suo tempo libero si è permanentemente ridotto. Le persone sono spesso riluttanti a fare ciò.

Quindi, tocca voi e agli altri nel progetto notare quello che sta avvenendo—o piuttosto, non avvenendo e chiedere al volontario cosa stia succedendo. L'indagine dovrebbe essere amichevole e al 100% priva di senso di colpevolezza. Il vostro proposito è trovare un pezzo di informazione, non far sentir male la persona. Generalmente l'indagine dovrebbe essere visibile al resto del progetto, ma se voi conoscete un motivo specifico per cui dovrebbe essere meglio che sia privata, questo va anche bene. La principale ragione per farla pubblicamente è che se in volontario risponde dicendo che non sarà più in grado di fare il lavoro in futuro, ci sia un contesto stabilito per il vostro *prossimo* post pubblico: la richiesta di un nuovo volontario che ricopra tale ruolo.

A volte, un volontario non è capace di fare il lavoro che ha intrapreso, ma o è inconsapevole o non vuole ammettere il fatto. Certo, chiunque più avere problemi all'inizio, specialmente se la responsabilità è complessa. Comunque, se qualcuno non sta proprio lavorando al suo compito che ha intrapreso, anche se tutti gli altri gli hanno dato tutto l'aiuto e i suggerimenti che potevano, allora la sola soluzione è che si faccia da parte e permetta a qualcun altro di provare. E se la persona non vede questo da sé, c'è bisogno che glielo si dica. Queste sono le uniche vie di base di gestire questa cosa, ma è un processo dai molti passi e ogni passo è importante.

n primo luogo, assicuratevi di non essere pazzi. Privatamente parlate agli altri nel progetto per vedere se gli altri sono d'accordo che il problema sia serio come credete. Anche se siete già del parere che lo sia, ciò serve allo scopo di permettere agli altri di sapere che voi state pensando di chiedere alla persona di mettersi da parte, Di solito nessuno fa obiezione a ciò—essi saranno solo felici del fatto che voi stiate intraprendendo un compito scomodo, di modo che essi non abbiano a farlo!

Poi, *privately* privatamente contattate il volontario in questione e parlategli, gentilmente ma direttamente, del problema che vedete. Siate specifici, dando quanti più esempi potete. Assicuratevi di

evidenziare come la gente abbia cercato di essere di aiuto, ma che il problema ha continuato ad esistere, senza miglioramento. Vi dovreste aspettare di metterci molto tempo a scrivere questa email, ma con questo tipo di messaggio, se non tornate indietro su ciò che state dicendo, non dovreste dirlo punto e basta. Dite che vorreste trovare un nuovo volontario per ricoprire il ruolo, ma puntualizzare che ci sono molti altri modi per collaborare col progetto. A questo punto non dite che avete parlato cogli altri di questo; nessuno vuole che si dica che la gente sta cospirando a sua insaputa.

Ci sono pochi modi differenti in cui le cose possano andare dopo questo. La più probabile reazione è che egli sia d'accordo con voi, o in ogni caso non voglia discutere, e voglia dimettersi. In tal caso, suggerite che faccia l'annuncio lui stesso, e poi voi potete dar seguito con un post di ricerca di un sostituto.

Oppure egli può essere d'accordo che ci sono stati dei problemi, ma chiede ancora un po' di tempo (o un'altra chance, nel caso di compito non continuativi in ruoli tipo manager di release). Come reagire alla cosa è una chiamata in giudizio, ma comunque facciate, non siate d'accordo su questo solo perché vi sentite come se non potete rifiutare una tale ragionevole richiesta. Ciò prolungherebbe l'agonia, non la diminuirebbe. C'è spesso una ragione molto buona per rifiutare la richiesta, vale a dire, che c'è stata una abbondanza di chance, e che è così che le cose sono arrivate al punto in cui sono ora. Qui è come lo ho messo in una email a un tale che stava ricoprendo il ruolo di manager di release ma non era proprio portato per esso:

```
> Se vuoi sostituire me con qualcun altro, mi sarà gradito
> passare al ruolo che viene dopo. Ho una richiesta che
> spero non sia irragionevole. Mi piacerebbe tentare con una nuova
> release nel tentativo di mettermi alla prova.

Io capisco completamente il desiderio (sono stato lì io stesso), ma in quest

Questa non è la prima o la seconda release, è la sesta o la settima...E per
```

Nel peggiore dei casi, il volontario può essere in disaccordo apertamente. Allora voi dovete prendere atto del fatto che le cose stanno per diventare scomode e andare avanti comunque. Adesso è il momento di dire alle altre persone ciò che avete detto su questo (ma non dite chi finché non avete il suo permesso, perché queste conversazioni erano confidenziali), e che voi non pensate che è bene che il progetto continui così. Siate insistenti, ma mai minacciosi. Tenete in mente che con la maggior parte dei ruoli, l'avvicendamento avviene in realtà nel momento in cui qualcuno incomincia a fare il nuovo lavoro, *non* non nel momento in cui la vecchia persona smette di farlo. Per esempio, se il contenzioso è sul ruolo, diciamo, del manager di problemi, ad ogni momento voi e le altre influenti persone nel progetto potete far pressione per un nuovo manager di problemi. In realtà non è necessario che la persona che prima lo stava facendo smetta di farlo, purchè non saboti (deliberatamente o per altro) lo fatica del nuovo volontario.

Il che porta a un pensiero tentatore. Invece di chiedere alla persona di dimettersi, perché non incorniciarlo nel dargli qualche aiuto? Perché non avere due manager di problemi, o qualunque sia il ruolo?

Sebbene ciò sembri simpatico in teoria, non è generalmente una buona idea. Ciò che fa sì che il ruolo di manager funzioni ciò che lo rende utile, nei fatti—è la sua centralizzazione. Quelle cose che possono essere fatte in uno stile decentralizzato, sono già fatte in quel modo. L'avere due persone a ricoprire un ruolo manageriale introduce un sovraccarico di comunicazioni fra queste due persone, come altrettanto la possibilità di uno scivoloso spostamento delle responsabilità (“Io pensavo che avresti portato il kit di primo aiuto” “Io, no, io pensavo che *tu* tu averesti portato il kit di primo aiuto”). Certo, ci sono eccezioni. A volte due persone lavorano estremamente bene insieme, o la natura del ruolo è tale che può essere allargato a più persone. Ma questi probabilmente non possono essere di molto aiuto quando voi vedete qualcuno che si dimena in un ruolo per cui non è portato. Se egli in primo luogo si fosse

reso conto del problema, avremmo cercato tale aiuto prima di adesso. In ogni caso, sarebbe stato irrispettoso permettere che qualcuno perdesse tempo nel continuare a fare un lavoro a cui nessuno prestasse attenzione.

Il fattore più importante nel chiedere a qualcuno di dimettersi è la privacy: dargli lo spazio per prendere una decisione senza che si senta come se gli altri lo stiano osservando e aspettando. Io una volta feci l'errore un chiaro errore, in retrospettiva—di scrivere a tutte le tre parti in un sola volta per chiedere che il manager di release di Subversion si facesse da parte in favore di altri due volontari. Avevo già parlato alle altre due persone privatamente, e sapevo che essi volevano assumersene la responsabilità. Così pensai, naïsul velluto e in qualche modo intensamente che avevo risparmiato tempo e rottura inviando una email a tutti loro per iniziare l'avvicendamento. Davo per acquisito che il manager di release attuale fosse pienamente al corrente dei problemi e avrebbe visto immediatamente la ragionevolezza del mio punto di vista.

Mi sbagliavo. L'attuale release manager fu molto offeso, e anche giustamente. Una cosa è che ti venga chiesto di passare la palla. Un'altra cosa che te lo si venga chiesto *d'avanti* alla persona a cui passerai la palla. Quando mi resi conto del perché si era offeso, mi scusai. Egli alla fine si fece da parte con grazia, e continua ad essere coinvolto nel progetto oggi. Ma i suoi sentimenti furono feriti, e inutile a dirsi, questo non fu il più favorevole degli inizi per nuovi i volontari per di più.

Quelli Che Fanno gli Invii

Come sola classe formalmente distinta che si trova nei progetti open source, quelli che fanno gli invii meritano una attenzione speciale qui. Quelli che fanno gli invii sono una concessione inevitabile alla discriminazione in un sistema che diversamente è quanto meno discriminatorio possibile. Ma la “discriminazione” qui non è intesa come dispregiativo. La funzione che quelli che fanno gli invii svolgono è assolutamente necessaria, e io non penso che un progetto avrebbe successo senza di essa. Il controllo della qualità richiede, giusto, il controllo. Ci sono sempre molte persone che si ritengono competenti nel fare cambiamenti a un programma, e un qualche numero minore che effettivamente lo sono. Il progetto non può contare sul personale giudizio della gente; esso deve imporre gli standard e deve concedere l'invio solo a quelli che li soddisfano.[4]. D'altro canto, avere persone che possano inviare cambiamenti che funzionino direttamente fianco a fianco con persone che non lo possono introduce una ovvia dinamica di potere. La dinamica deve essere gestita in modo da non danneggiare il progetto.

In sezione chiamata «Chi Vota?» in Capitolo 4, *L'Infrastruttura Sociale e Politica*, noi già discutemmo i meccanismi del considerare i nuovi ammessi all'invio. Qui noi daremo un'occhiata agli standard con i quali i potenziali ammessi all'invio dovrebbero essere giudicati, e come questo procedimento debba essere presentato alla comunità allargata.

Scegliere Coloro che Faranno gli Invii

Nel progetto di Subversion noi sceglievamo coloro che dovevano far gli invio sulla base del principio Ippocratico: *primo non fare danni*. Il nostro criterio principale non è l'abilità tecnica oppure la conoscenza del codice, ma soltanto il fatto che chi fa gli invii mostri buon giudizio. Giudizio può significare semplicemente sapere cosa non intraprendere. Una persona potrebbe postare solo piccole patch, risolvendo onestamente piccoli problemi nel codice; ma se le patch si applicano un modo pulito, non contengono bug, e sono prevalentemente in accordo con i messaggi di log del progetto e col codice, con le convenzioni, e ci sono sufficienti patch a mostrare una chiara linea di comportamento, allora

[4]Notate che l'accesso all'invio significa qualcosa di differente in un sistema di versione decentralizzato, in cui ognuno può allestire un deposito che è collegato nel progetto, e dà a se stesso l'accesso all'invio a quel deposito. Nondimeno ancora si applica il *concetto* dell'accesso all'invio. L'”accesso all'invio” stenograficamente sta per “il diritto di fare cambiamenti al codice che saranno inoltrati alla prossima release del software del gruppo”. In un sistema di controllo di versione centralizzato, questo significa avere l'accesso diretto all'invio; in un sistema decentralizzato, significa avere che i cambiamenti di uno sono tirati dentro alla distribuzione automaticamente. E' la stessa idea in ogni caso; i meccanismi con cui son realizzati non sono terribilmente importanti.

uno che fa già gli invii di solito propone quella persona per l'accesso all'invio. Se almeno tre persone dicono si, e nessuno obietta, allora viene fatta la proposta. Giusto, voi non potreste avere l'evidenza che è capace di risolvere problemi complessi in tutte le aree del codice base, ma questo non ha importanza: la persona ha reso chiaro che almeno è capace di giudicare le sue abilità. Le capacità tecniche possono essere imparate (e insegnate), ma il giudizio, per la maggior parte, no. Quindi, è la sola cosa che volete assicurarvi la persona abbia, prima di dargli l'accesso all'invio.

Quando la proposta di un nuovo accesso all'invio provoca discussioni, non è riguardo alle capacità tecniche, ma riguardo al comportamento della persona nella mailing list o in IRC. A volte qualcuno mostra abilità tecniche e capacità di lavorare all'interno delle linee guida formali del progetto, eppure è sostanzialmente bellicoso e non collaborativo nei forum pubblici. Questa è una faccenda seria; se la persona non sembra darsi da fare col tempo, anche in risposata ai consigli, allora noi non lo aggiungeremo come persona che fa gli invii non importa quanto capace sia. In un gruppo di volontari, capacità di socializzazione, o l'abilità di “giocare bene in un ambiente di prova”, sono importanti quanto le pure abilità tecniche. Siccome ogni cosa è sotto il controllo di versione, la penalità per l'aggiunta di uno che fa gli invii che voi non dovreste avere non è è tanto quanti problemi potrebbe causare al codice (la revisione li individuerebbe ad ogni modo), ma il fatto che potrebbe alla fine obbligare il progetto a revocare l'accesso agli invii alla persona—un atto che non è mai piacevole e che potrebbe a volte dar luogo a polemiche.

Molti progetti insistono sul fatto che la persona che potenzialmente potrebbe fare gli invii dimostri un certo livello di esperienza tecnica e sia tenace, inviando un certo numero di patch non secondarie; cioè, non solo questi progetti vogliono sapere che la persona non procurerà danni, vogliono sapere se possibilmente è brava nel codice base. Ciò è bene, ma siate attenti a che ciò non incominci a cambiare l'appartenenza a quelli che fanno gli invii in appartenenza ad un club esclusivo. La domanda da tenere a mente dovrebbe essere ”Cosa porterà i migliori risultati per il codice?” non “Dobbiamo svalutare lo stato sociale associato alla facoltà di fare gli invii ammettendo questa persona?” Il punto dell'accesso agli invii non è rafforzare l'autostima della gente, è consentire che buoni cambiamenti entrino nel codice senza un minimo di agitazione. Se voi avete 100 persone che fanno gli invii, 10 dei quali fanno cambiamenti su base regolare, e 90 dei quali correggono solo degli errori di stampa o dei piccoli bugs poche volte all'anno, questo è ancora meglio che avere solo i 10.

Revocare l'Accesso all'Invio

La prima cosa da dire sulla revoca dell'accesso all'invio è: cercate di non essere in quella situazione in primo luogo. A seconda delle persone il cui accesso all'invio sta venendo revocato, le discussioni su una tale azione possono essere molto divisive. Anche quando non fossero divisive, esse sarebbero causa di perdita di tempo ai danni del lavoro produttivo.

Comunque, se dovete farlo, la discussione dovrebbe tenersi privatamente fra le stesse persone che sarebbero nella posizione di votare per *concedere* a quella persona qualunque gusto dell'accesso all'invio che correntemente ha. La persona stessa non dovrebbe essere inclusa. Questo contraddice la solita ingiunzione contro la segretezza, ma in questo caso è necessario. Primo, nessuno sarebbe in grado di parlare liberamente altrimenti. Secondo, se la mozione non va a termine, voi non necessariamente volete che la persona sappia che ciò sia stato mai preso in considerazione, perché ciò potrebbe aprire domande (“Chi era dalla mia parte? Chi contro di me?) che portano al peggior tipo di faziosità. In certe rare circostanze, il gruppo può volere che qualcuno sappia che la revoca dell'acceso all'invio è stata o sta venendo considerata, come un avviso, ma questa apertura dovrebbe essere una decisione che prende il gruppo. Nessuno dovrebbe mai, di sua iniziativa, rivelare informazioni su una discussione e su un voto che gli altri presumono che siano segreti.

Once someone's access is revoked, that fact is unavoidably public (see sezione chiamata «Evitare Misteri» later in this chapter), so try to be as tactful as you can in how it is presented to the outside world.

Accesso all'Invio Parziale

Alcuni progetti presentano gradazioni dell'accesso all'invio. Per esempio, ci possono essere collaboratori il cui accesso all'invio dà loro libere redini nella documentazione, ma essi non possono fare gli invii per quanto riguarda il codice stesso. Aree comuni di invio parziale sono la documentazione, le traduzioni, l'adattare il codice ad altri linguaggi di programmazione, i file di specificazione per i pacchetti (per es. i file di specificazione per RedHat RPM) ed altre aree in cui un errore non dà luogo a un problema per la base del progetto.

Poiché l'accesso all'invio non è solo accesso all'invio ma significa anche far parte di un elettorato (vedere sezione chiamata «Chi Vota?» in Capitolo 4, *L'Infrastruttura Sociale e Politica*), nasce naturalmente la domanda: su cosa possono votare quelli che hanno l'invio parziale? Qui non c'è una singola risposta giusta; dipende da che tipo di dominio di invio parziale ha il vostro progetto. In Subversion noi abbiamo mantenuto le cose abbastanza semplici: uno che ha l'accesso all'invio parziale, può votare solo su questioni confinate al dominio di colui che fa gli invii, e non su ogni altro dominio. Importante, noi abbiamo meccanismi per dare voto consultivo (essenzialmente colui che fa gli invii scrive "+0" o "+1 (non impegnativo) invece che solo “+1” sulla scheda. Non c'è motivo di silenziare completamente la gente solo perché il loro voto non è formalmente impegnativo.

Coloro che hanno il pieno invio possono votare su ogni cosa, e solo coloro che hanno il pieno invio possono votare sull'ammissione di nuove persone con diritto di voto di ogni specie. Nella pratica, tuttavia, la competenza ad ammettere nuove persone con l'invio parziale è di solito delegata: ogni persona con invio pieno può “sponsorizzare” una nuova persona con invio parziale, e quest'ultimo in un dominio può in affetti scegliere nuove persone con diritto di invio per quello stesso dominio (questo è particolarmente utile nel far si che il lavoro di traduzione funzioni regolarmente).

Il vostro progetto può aver bisogno di piccoli differenti aggiustamenti, a seconda della natura del lavoro, ma a ogni progetto si applicano gli stessi principi generali. Ogni persona che ha l'invio dovrebbe poter votare su questioni che cadono entro la portata del suo accesso all'invio, e non su questioni all'infuori di esso, e i voti su questioni procedurali dovrebbero essere concessi solo a quelli che hanno il pieno invio, a meno che non ci sia qualche ragione (come deciso da coloro che hanno il pieno invio) di allargare l'elettorato.

Riguardo all'applicazione dell'accesso parziale all'invio: è spesso meglio *non* avere che il sistema di controllo di versione faccia rispettare i domini dell'invio parziale, anche se lo può. Vedere sezione chiamata «Autorizzazione» in Capitolo 3, *L'Infrastruttura Tecnica* le ragioni del perché.

Persone che Hanno l'Accesso all'Invio Dormienti

Alcuni progetti rimuovono automaticamente le persone dall'accesso all'invio se essi per un certo tempo (diciamo, un anno) non fanno invii. Penso che questo non sia di aiuto e anche controproducente, per due ragioni.

Primo, ciò può tentare alcuni ad inviare cambiamenti accettabili ma non necessari, solo per impedire che il loro accesso all'invio si estingua. Secondo, non serve veramente a niente. Se il criterio principale per la concessione dell'accesso all'invio è il buon giudizio, allora perché presumere che il giudizio di qualcuno si deteriorerebbe solo perché egli è via dal progetto per un certo tempo? Anche se egli svanisce completamente per anni, non guardando il codice o non seguendo le discussioni dello sviluppo, quando riappare *saprà* quanto ha perso il contatto, ed agirà di conseguenza. Avete avuto fiducia nel suo giudizio prima, allora perché non averla sempre? Se i diplomi della scuola superiore non si estinguono, allora l'accesso all'invio certamente non lo dovrebbe.

A volte colui che ha un accesso all'invio può chiedere di essere rimosso, o di essere esplicitamente segnalato come dormiente nella liste di quelli che fanno gli invii (vedere sezione chiamata «Evitare

Misteri» sotto per maggiori dettagli su questa lista). In quei casi, il progetto dovrebbe acconsentire ai desideri della persona, certamente.

Evitare Misteri

Anche se la discussioni sull'ammissione di un nuova particolare persona all'invio deve essere confidenziale, i ruoli e le stesse procedure non c'è bisogno che siano segreti. Infatti, è meglio renderli pubblici, così la gente si rende conto che quelli che fanno gli invii non sono delle Camere da Star chiuse ai solo mortali, ma che chiunque può raggiungerli semplicemente postando delle buone patch e conoscendo come comportarsi nella comunità. Nel progetto di Subversion, noi mettiamo questa informazione giusto nel documento delle linee guida, dal momento la gente più adatta ad essere interessata a come l'accesso agli invii viene concesso è quella che pensa di contribuire codice al progetto.

Oltre a pubblicare le procedure, pubblicate *la lista* attuale di coloro che hanno accesso all'invio . Il posto tradizionale per fare questo è un file chiamato `MAINTAINERS` o `COMMITTERS` in cima all'albero del codice sorgente del progetto. Esso dovrebbe elencare prima tutti coloro che hanno l'invio pieno, seguiti dai vari domini con accesso parziale e dai membri di ciascun dominio. Ogni persona dovrebbe essere elencata col nome e con l'indirizzo email, sebbene l'email potrebbe essere codificata per evitare spam (vedere sezione chiamata «Nascondere gli indirizzi presenti negli archivi» in Capitolo 3, *L'Infrastruttura Tecnica*) se la persona preferisce così

Poichè la distinzione fra accesso all'invio pieno e all'invio parziale è chiara e ben definita, è proprio dell'elenco fare la distinzione anche. Oltre a ciò l'elenco non dovrebbe cercare di indicare le distinzioni non formali che inevitabilmente si presentano in un progetto, come chi è influente e come. E' una registrazione pubblica, non un file di riconoscimenti. Elencate quelli che hanno accesso all'invio in ordine alfabetico, o nell'ordine in cui sono arrivati.

Riconoscimenti

I riconoscimenti sono la moneta principale nel mondo del software libero. Qualunque cosa una persona possa dire sulle sue motivazioni della partecipazione a un progetto, non conosco sviluppatori che sarebbero felici di fare tutto il loro lavoro anonimamente, o sotto il nome di qualcun altro. Ci sono ragioni tangibili per questo: la reputazione di uno nel progetto approssimativamente determina quanta influenza ha, e la partecipazione a un progetto open source può anche avere indirettamente un valore monetario, perché alcuni datori di lavoro guardano a questo nel curriculum. Ci sono anche ragioni non tangibili, magari anche più forti: la gente semplicemente vuole essere apprezzata, e istintivamente guarda a segni che il loro lavoro è stato riconosciuto dagli altri. La promessa di riconoscimenti è quindi è una delle migliori motivazioni che il progetto abbia. Quando sono riconosciuti piccoli contributi, la gente ritorna per fare di più.

Una delle più importanti caratteristiche dello sviluppo in collaborazione (vedere Capitolo 3, *L'Infrastruttura Tecnica*) è quella che esso tiene accurate registrazioni di chi fece cosa, quando. Dovunque è possibile, usate questi meccanismi esistenti per assicurarvi che i riconoscimenti siano distribuiti accuratamente, e siano specifici sulla natura del contributo. Non scrivete solo “Grazie a J. Random <jrandom@example.com>" se invece potete scrivere “Grazie a J. Casuale <jrandom@example.com> per il rapporto di bug e per la ricetta per la riproduzione" in un messaggio di log.

In Subversion, noi abbiamo la informale ma costante politica di riconoscere a chi fa un rapporto di bug o nel problema archiviato, se ce n'è uno, o nel messaggio di log dell'invio che corregge il bug, se no. Un rapido controllo dei logs di invio fino all'invio 14525 mostra che circa il 10% degli invii dà il riconoscimento a qualcuno per nome ed email, di solito la persona che ha analizzato e ha riportato il bug corretto da quell'invio. Notate che questa persona è differente dallo sviluppatore che in realtà

fece l'invio, il cui nome è registrato automaticamente dal sistema di controllo della versione. Delle 80 e rotte persone che hanno il pieno o parziale invio in Subversion, 50 avevano i riconoscimenti nei log di invio (di solito molte volte) prima di diventare persone con l'invio essi stessi. Ciò, certo, non prova che l'essere riconosciuto è un fattore del loro coinvolgimento continuato, ma ciò almeno stabilisce una atmosfera in cui la gente può contare sul fatto che i suoi contributi vengono riconosciuti.

Bisogna distinguere fra riconoscimenti di routine e ringraziamenti speciali. Quando si discute un pezzo particolare di codice, o qualche altro contributo che qualcuno ha dato, è bene riconoscere il suo lavoro. Per esempio, dicendo “I recenti cambiamenti di Daniel al codice delta significano che ora noi possiamo implementare la funzionalità X” contemporaneamente aiuta la gente a identificare di quali cambiamenti state parlando e riconosce il lavoro di Daniel. D'altra parte, postare solamente per ringraziare Daniel per i cambiamenti al codice delta non serve ad uno scopo immediato. Non aggiunge nessuna informazione, perché il sistemi di controllo della versione e altri meccanismi hanno già registrato il fatto che egli ha fatto dei cambiamenti. Ringraziare ciascuno per ogni cosa può distrarre e in ultimo può essere senza informazione, perché i ringraziamenti sono in gran parte effettivi, nela misura in cui stanno fuori dal comune, livello di base di un commento favorevole che avvenga tutte le volte. Questo non significa, certo, che voi non dovreste ringraziare la gente. Solo assicuratevi di farlo in modo che non tenda a portare ad una inflazione di riconoscimenti. Seguire queste linee guida aiuterà:

- Più è effimero il forum, più vi sentirete liberi di esprimere i ringraziamenti lì. Per esempio, ringraziando qualcuno per la correzione di un bug incidentalmente durante una conversazione in IRC è cosa buona, come lo è una divagazione in una email dedicata principalmente ad altri argomenti. Ma non postate una email solo per ringraziare qualcuno, a meno che ciò non sia per una impresa eroica non solita. Nello stesso modo, non intasate le pagine web del progetto con espressioni di gratitudine. Una volta che iniziate a farlo, non sarà mai chiaro quando e dove fermarsi. E non mettete *mai* ingraziamenti nei commenti al codice; ciò sarebbe solo una distrazione dal proposito principale dei commenti, che è aiutare il lettore a capire il codice.

- Meno qualcuno è coinvolto nel progetto, più è appropriato ringrazialo per qualcosa che ha fatto. Ciò potrebbe suonare contrario all'intuito, ma è in linea con l'opinione che l'esprimere ringraziamenti è qualcosa che voi fate quando qualcuno contribuisce ancora di più di quanto voi pensavate che facesse. Così, ringraziare costantemente i collaboratori regolari perché fanno ciò che regolarmente fanno significherebbe esprimere una minore aspettativa su di loro di quanto essi la hanno di se stessi. Se mai, voi volete puntare all'effetto opposto!

  Ci sono eccezioni occasionali a questa regola. E' accettabile ringraziare qualcuno perché occupa soddisfacentemente il suo ruolo previsto quando quel ruolo comporta fatiche temporanee intense di volta in volta. L'esempio canonico è il manager di release che va a velocità più elevata per quanto riguarda il tempo di ciascuna release, ma altrimenti rimane dormiente (dormiente come manager di release in ogni caso egli può essere anche un attivo sviluppatore, ma questa è un'altra questione).

- Come per il criticismo e per i riconoscimenti, la gratitudine dovrebbe essere specifica. Non ringraziate la gente solo perché è grande, anche se lo è. Ringraziatela per qualcosa che ha fatto che era fuori dell'ordinario e per il buon punteggio, dite esattamente perché ciò che fecero era così grande.

n generale c'è sempre una tensione fra l'assicurarsi che i contributi individuali della gente siano riconosciuti, e l'assicurarsi che che il progetto sia una fatica pubblica invece che un insieme di glorie individuali. Giusto rimanete al corrente di questa tensione e cercate di sbagliare dalla parte del gruppo, e le cose non vi scapperanno di mano.

Le Diramazioni

In sezione chiamata «La Possibilità di Diramazione» in Capitolo 4, *L'Infrastruttura Sociale e Politica*, noi vedemmo come *la possibilità* di una diramazione abbia un effetto importante su come il progetto viene amministrato. Ma cosa accede quando una diramazione si verifica veramente? Come dovreste

gestire la cosa? E quali effetti dovete aspettarvi che la cosa abbia? All'inverso, quando dovreste *iniziare* una diramazione?

La risposta dipende da che tipo di diramazione è? Alcune diramazioni si hanno per un amichevole e inconciliabile disaccordo sulla direzione del progetto; forse la maggior parte sono dovute a disaccordi tecnici e a conflitti interpersonali. Certo, non è sempre possibile dire le differenza fra le due cose, in quanto elementi tecnici possono coinvolgere elementi personali. La cosa che tutte le diramazioni hanno in comune è che un gruppo di sviluppatori (o talvolta anche uno solo degli sviluppatori) ha deciso che i costi del lavorare con qualcuno o tutti gli altri ora non prevalgono sui benefici.

Una volta che il progetto si biforca, non c'è una risposta definitiva alla domanda su quale diramazione è il “vero” o “originale” progetto. La gente parlerà in modo colloquiale della diramazione F che viene dal progetto P, come se P sta continuando per la sua normale traiettoria senza una marcia inferiore, mentre F diverge in nuovo territorio, ma questa è, in effetti, una dichiarazione di come quel particolare osservatore avverte la cosa. E' fondamentalmente una questione di percezione: quando una larga percentuale di osservatori è d'accordo, l'affermazione incomincia a diventare vera. Non è il caso che ci sia una verità oggettiva sin dall'inizio, una che noi siamo solo imperfettamente capaci di percepire all'inizio. Piuttosto, le percezioni *sono* la verità oggettiva, poichè in ultima analisi una diramazione—o un progetto—sono una entità che esiste solo nella mente della gente comunque.

Se quelli che stanno iniziando una diramazione ritengono di star facendo crescere un nuovo ramo fuori dal progetto principale, la questione della percezione è risolta immediatamente e facilmente. Ognuno, sviluppatori e utilizzatori, tratterà la diramazione come un nuovo progetto, con un nome (magari basato sul vecchio nome, ma facilmente distinguibile da esso), un sito separato, e una filosofia separata per quanto riguarda gli obbiettivi. Le cose riusciranno disordinate, comunque, quando ambedue le parti riterranno di essere i guardiani legittimi del progetto originale e quindi di avere il diritto di continuare ad usare in nome originale. Se c'è qualche organizzazione con diritto di marchio sul nome, o con un controllo legale sulle pagine web o sul dominio, ciò di solito risolve il problema per decreto: quella organizzazione decide chi è nel progetto e chi nella diramazione, perché detiene tutte le carte in una guerra di pubbliche relazioni. Naturalmente, raramente le cose arrivano fino a questo punto: dal momento che ciascuno conosce quali sono le dinamiche del potere, eviterà di combattere una battaglia il cui esito è noto in partenza, e giusto salta dritto alla fine.

Fortunatamente, nella maggior parte dei casi, c'è un piccolo dubbio su quale sia il progetto e quale la diramazione perchè una diramazione è, in essenza, un voto di fiducia. Se più della metà degli sviluppatori sono favorevoli a qualunque corso si propone di prendere la diramazione, di solito non c'è bisogno della diramazione il progetto può andare per quella strada da sé, a meno che non diventi una dittatura con un dittatore particolarmente testardo. D'altra parte, se meno della metà degli sviluppatori sono favorevoli, la diramazione è proprio una ribellione di una minoranza, e la cortesia e il senso comune indicano che essa dovrebbe ritenersi un ramo divergente piuttosto che la linea principale.

Gestire Una Diramazione

Se qualcuno minaccia una diramazione nel vostro progetto, mantenete la calma e ricordate gli obiettivi a lungo termine. La sola *esistenza* di una diramazione non è ciò che fa male a un progetto; piuttosto è la perdita di sviluppatori e utilizzatori. Il vostro vero scopo non è schiacciare la diramazione, ma minimizzare i suoi effetti dannosi. Potete essere furiosi, potete ritenere che la diramazione fu ingiusta e non necessaria, ma esprimere questo pubblicamente può solo alienare gli sviluppatori indecisi. Invece non obbligate la gente a fare scelte esclusive, e siate collaborativi e pratici con la diramazione. Per iniziare, non togliete l'accesso all'invio a qualcuno nel vostro progetto solo perché ha deciso di lavorare alla diramazione. Lavorare sulla diramazione non significa che la persona ha improvvisamente perso la sua competenza a lavorare al progetto originale; coloro che fanno gli invii prima dovrebbero rimanere quelli che fanno gli invii dopo. Oltre a ciò voi dovreste esprimere il desiderio di restare quanto più compatibili possibile con la diramazione, e dire che sperate che gli sviluppatori trasferiranno i cambiamenti fra il progetto e la diramazione quando è opportuno. Se avete l'accesso amministrativo ai

servers, offrite a quelli della diramazione l'aiuto delle sovrastrutture al momento dell'avvio. Per esempio, offrite loro una copia completa con la storia profonda del deposito del controllo di versione, se essi non hanno altro modo di ottenerla, cosicché essi non abbiano a partire senza dati storici (ciò può essere non necessario a seconda del sistema di controllo della versione). Chiedete loro se c'è qualcos'altro di cui abbiano bisogno, e fornitelo se potete. Chinatevi all'indietro per mostrare che non state fra i piedi e per mostrare che la diramazione avrà successo o fallirà per i suoi meriti e nient'altro.

Il motivo per fare tutto ciò e farlo pubblicamente non è in verità quello di aiutare la diramazione, ma convincere gli sviluppatori che la vostra parte è una scommessa sul sicuro, apparendo quanto meno vendicativi possibile. In guerra talvolta ha senso (senso strategico, non umano) costringere la gente a scegliere una parte, ma nel software libero quasi sempre non lo ha. Infatti, dopo una diramazione, alcuni sviluppatori spesso lavorano apertamente ad ambedue i progetti e fanno del loro meglio per tenere le due cose compatibili. Questi sviluppatori aiutano a tenere aperte le linee di comunicazione dopo la diramazione. Essi permettono al vostro progetto di beneficiare di nuove interessanti funzionalità della diramazione (si, la diramazione può avere cose che volete), e anche accrescere le possibilità di una fusione per strada.

A volte una diramazione ha così successo, che anche se era considerata dai suoi stessi istigatori come diramazione dall'inizio, diventa la versione che ognuno preferisce, e alla fine soppianta l'originale per richiesta popolare. Un esempio famoso di ciò fu la diramazione GCC/EGCS. La *GNU Compiler Collection* (*GCC*, prima la *GNU C Compiler*) è il codice nativo di compiler open source più popolare, e anche uno dei più portabili compilatori del mondo. Dovuto al disaccordo fra i manutentori ufficiali del GCC e la Cygnus Software.,[5] uno dei gruppi di sviluppatori più attivi, Cygnus creò una diramazione di GCC chiamata *EGCS*. La diramazione era deliberatamente non antagonistica: gli sviluppatori EGCS , in qualunque momento, non cercavano di descrivere la loro versione di GCC come una nuova versione. Invece si concentrarono sul fatto di rendere EGCS quanto migliore possibile, incorporando patch ad un ritmo superiore a quello dei manutentori dello GCC ufficiale. EGCS guadagnò in popolarità, e alla fine alcuni principali distributori di sistemi operativi decisero di impacchettare EGCS come loro compilatore di default invece di GCC. A questo punto, diventò chiaro ai manutentori di GCC che insistendo sulla denominazione "GCC" mentre ognuno passava alla diramazione EGCS avrebbe caricato ognuno con un inutile cambiamento di nome, ma non avrebbe fatto niente per impedire la migrazione. Così GCC adottò il codice base di EGCS, e c'è ancora un solo GCC, ma grandemente migliorato con la diramazione.

Questo esempio mostra perché non potete sempre considerare una diramazione come una assoluta cosa brutta. Una diramazione può essere dolorosa e non benvenuta al momento, ma voi non potete sapere se avrà successo. Quindi, voi e il resto del progetto dovete tenerci un occhio su, ed essere preparati non solo ad assorbirne il codice e le funzionalità quando possibile, ma nei casi più estremi, anche ad unirvi alla diramazione se essa raggiunge le dimensioni di popolarità del progetto. Certo, sarete spesso in grado di predire la probabilità di successo di una diramazione guardando chi si unisce ad essa. Se la diramazione è partita dai più grandi lamentosi del progetto e ad essa si è aggiunta una manciata di sviluppatori scontenti che non si stavano comportando costruttivamente comunque, essi hanno sostanzialmente risolto il problema con voi con la diramazione, e voi probabilmente non dovete preoccuparvi della diramazione portando via slancio al progetto originale. Ma se vedete sviluppatori influenti ed rispettati che sostengono la diramazione, ve ne dovrete chiedere il perché. Magari il progetto stava essendo eccessivamente restrittivo, e la migliore soluzione è adottare nella linea principale del progetto alcuni o tutti gli atti contemplati dalla diramazione—in sostanza evitare che la diramazione diventi tale.

Iniziare una Diramazione

Tutto il consiglio qui è nell'ipotesi che stiate facendo una diramazione come ultima risorsa. Esaurite tutte le altre possibilità prima di iniziare una diramazione. Fare una diramazione significa quasi sempre

[5]Adesso parte di RedHat (http://www.redhat.com/).

perdere sviluppatori, con solo una incerta promessa di acquisirne di nuovi dopo. Essa significa anche iniziare una competizione per l'attenzione degli utilizzatori: chiunque stia per scaricare ha da chiedere a se stesso: “Hmm, voglio questo o l'altro?” Qualunque dei due voi siate, la situazione è sporca, perché è stata introdotta una domanda che non c'era prima. Alcune persone sostengono che la diramazione è salutare per l'ecosistema del software nella sua interezza, per il classico argomento della selezione naturale: sopravviverà il più adatto, che significa che, alla fine, ognuno ottiene il software migliore. Questo può essere vero dal punto di vista degli ecosistemi, ma non è vero dal punto di vista di un singolo progetto. La maggior parte delle diramazioni non riescono, e la maggior parte dei progetti non sono contenti di essere biforcati.

Un corollario è che non dovreste usare la minaccia di una diramazione come tecnica estremistica di una discussione—“Fate le cose come dico io o io biforcherò il progetto!”—poiché chiunque è al corrente che un diramazione ch non riesce ad attrarre sviluppatori dal progetto originale è improbabile che viva a lungo. Tutti gli osservatori—non solo gli sviluppatori, ma utilizzatori e impacchettatori del sistema operativo anche—si faranno la loro idea su che parte scegliere. Voi dunque dovreste apparire estremamente riluttanti a un diramazione, così se alla fine la fate, possiate reclamare che era l'ultima via rimasta.

Non trascurate di tenere in conto *tutti* i fattori nel valutare il potenziale successo della vostra diramazione. Per esempio, se molti degli sviluppatori in un progetto hanno lo stesso datore di lavoro, allora anche se sono scontenti e privatamente favorevoli alla diramazione, è improbabile che lo dicano così a voce alta se il loro datore di lavoro è contro di essa. Molti programmatori di software libero amano pensare che avere una licenza libera sul codice significa che nessuna compagnia può dominare lo sviluppo. E vero che la licenza, in un senso definitivo, è un garante di libertà—se gli altri vogliono abbastanza fortemente biforcare il progetto, ed hanno le risorse per farlo, essi possono. Ma in pratica, alcuni team di progetti furono in gran parte finanziati da una entità, e non c'è motivo di pretendere che non interessi quel supporto dell'entità. Se essa si oppone alla diramazione, è improbabile che i suoi sviluppatori vi prendano parte, anche se segretamente lo vogliono.

Se tuttavia concludete che dovete fare la diramazione, allineatevi privatamente col supporto prima, quindi annunciate la diramazione in un modo non ostile. Anche se siete arrabbiati, o in disappunto, con i proprietari correnti, non dite questo nel messaggio. Giusto spassionatamente dichiarate ciò che vi ha portato alla decisione di una diramazione, e che non volete significare nessuna intenzione malevola verso il progetto da cui vi diramate. Dando per ipotesi che voi la considerate una diramazione (come opposta a una conservazione di emergenza del progetto originale), mettete l'accento sul fatto che non state diramando il nome ma il codice, e scegliete un nome che no vada in conflitto col nome del progetto. Potete usare un nome che contiene o si riferisce al nome originale, finché ciò non apre la porta a una confusione di identità. Certo è bene spiegare in modo prominente sulla pagina web della diramazione che essa discende dal programma originale, e anche che essa spera di soppiantarlo. Solo non fate si che la vita degli utilizzatori sia più difficile obbligandoli a sbrogliare una disputa di identità.

Infine, potete ottenere che le cose partano con il piede giusto concedendo automaticamente a tutti coloro che avevano l'invio nel progetto originale l'accesso all'invio nella diramazione, inclusi quelli che erano apertamente in disaccordo con la necessità di una diramazione. Anche se essi non usano mai l'accesso all'invio, il vostro messaggio è chiaro: ci sono disaccordi qui, ma non nemici, e date il benvenuto ai contributi di codice provenienti da ogni origine competente.

Capitolo 9. Licenze, Diritti d'Autore e Brevetti

La licenza che scegliete probabilmente non avrà grande impatto sull'adozione del vostro progetto, finché è open source. Gli utilizzatori scelgono generalmente software basato su qualità e funzionalità, non sui dettagli della licenza. Nondimeno, avete bisogno di una comprensione dei problemi delle licenze del software libero, compresa l'assicurazione che la licenza sia compatibile con i suoi obiettivi, e che siate capaci di discutere le decisioni sulla licenza con altre persone. Prego notate, che io non sono un legale, e che niente in questo capitolo dovrebbe essere costituito come un consiglio legale. Per quello avrete bisogno di impiegare un un legale o di essere un legale.

La Terminologia

In una discussione su licenze open source, la prima cosa che balza all'evidenza è che sembrano esserci varie parole per la medesima cosa: *software libero*, *open source*, *FOSS*, *F/OSS*, and *FLOSS*. Partite con l'ordinarle, insieme ad altri pochi termini

software libero
: Il software può essere liberamente condiviso e modificato, includendolo in qualche forma di codice. Il termine fu coniato per prima da Richard Stallman, che lo codificò nella GNU General Public License (GPL), e fondò la Free Software Foundation (http://www.fsf.org/) per promuoverne il concetto.

  Sebbene il “software libero” copra quasi esattamente la stessa estensione dell'”open source”, la FSF, fra gli altri, preferisce il primo termine perché esso enfatizza l'idea di libertà e di software redistribuibile liberamente soprattutto come movimento sociale piuttosto che tecnico. La FSF riconosce che il termine è ambiguo—esso significherebbe “libero” nel senso di “a costo zero”, invece che “libero nel senso di “in libertà” ma ritiene che esso sia ancora il miglior termine, tutto considerato, e che le altre possibilità in inglese hanno le loro ambiguità. (In questo libro “free” è usato nel senso di “in libertà” non nel senso di “a costo zero”).

software open source
: Software libero sotto altro nome. Ma il nome differente riflette una importante differenza filosofica: open source fu coniato dall'Open Source Initiative (http://www.opensource.org/) come una voluta alternativa al “software libero”, per rendere tale software una scelta più gradita per le grandi imprese, presentandola come una metodologia di sviluppo, piuttosto che un movimento politico. Essi anche avevano voluto smontare un altro marchio: quello che ogni cosa “libera” debba essere di bassa qualità.

  Mentre una licenza che sia libera è anche open source, e viceversa, (con piccole trascurabili eccezioni), la gente tende a raccogliere un termine e incollarlo ad essa. In generale quelli che preferiscono “software libero”, molto verosimilmente sono per avere un atteggiamento morale verso il problema, mentre coloro che preferiscono “open source”, o non la vedono come una questione di libertà, o non sono interessati a metter in mostra il fatto che lo fanno. Vedere sezione chiamata «"Free" e "open source" a confronto» in Capitolo 1, *Introduzione* per una storia più dettagliata dello scisma.

  La Free Software Foundation fa una eccellente—assolutamente non oggettiva, ma sottile e completamente corretta—esegesi dei due termini, a http://www.fsf.org/licensing/essays/free-software-for-freedom.html. L'iniziativa presa dall' Open Source Initiative su questa

cosa è (o era nel 2002) divulgata in due pagine: http://www.opensource.org/advocacy/case_for_hackers.php#marketing and http://www.opensource.org/advocacy/free-notfree.php.

FOSS, F/OSS, FLOSS
: Dove ce ne sono due di ogni cosa, lì ce ne saranno presto tre, e questo è esattamente ciò che sta avvenendo con i termini per il software libero. Il mondo accademico, forse volendo la precisione e la comprensione al di sopra dell'eleganza, sembra aver deciso per FOSS o talvolta per F/OSS che sta per "Free / Open Source Software". Un'altra variante che sta guadagnando slancio è FLOSS che sta per "Free / Libre Open Source Software" (*libre* è familiare in molte lingue e non soffre dell'ambiguità di “free”; vedere http://en.wikipedia.org/wiki/FLOSS per sapere di più).

  Tutti questi termini significano essenzialmente la stessa cosa: software che può essere modificato e redistribuito da chiunque, a volte ma non sempre col requisito che i lavori derivati siano liberamente redistribuibili sotto gli stessi termini.

DFSG-compliant
: Conforme elle linee guida della Debian Free Software (http://www.debian.org/social_contract#guidelines). Questo è il testo largamente usato per indicare se una data licenza è veramente open source (free, *libre*, etc.). La missione del Progetto Debian è quella di mantenere un sistema operativo completamente libero, dimodochè non ci sia bisogno per uno che lo installa di avere il dubbio se abbia il diritto di modificare o redistribuire in parte o del tutto il sistema. Le linee guida Debian Free Software sono il requisito che la licenza di un pacchetto di software deve avere per essere incluso in Debian. Poichè il progetto Debian spese una buona quantità di tempo a pensare come costruire questo testo, le linee guida cui si pervenne si dimostrarono molto robuste (vedere http://en.wikipedia.org/wiki/DFSG), e da quanto mi risulta, nessuna seria obiezione è stata sollevata su di esse sia dalla Free Software Foundation, sia dalla Open Source Initiative. Se sapete che una licenza è DFSG-conforme, sapete che essa garantisce tutte le importanti libertà (come l'autirizzazione a iniziare un nuovo progetto partendo dal progetto sorgente, anche contro i desideri dell'autore originale) richiesta per sostenere le dinamiche di un progetto open source. Tutte le licenze discusse in questo capitolo sono DFSG-conformi.

OSI-approved
: Approvata dall'Open Source Initiative. Questo è un altro testo largamente usato per dire se una liacenza permette tutte le necessarie libertà. La definizione di software open source si basa sulle linee guida del Debian Free Software, e una licenza che ha una definizione quasi sempre ha l'altra. Ci sono state poche eccezioni negli anni, ma riguardanti solo particolari licenze e nessuna di qualche rilevanza qui. Diversamente dal progetto Debian, l'OSI conserva un elenco di tutte le licenze che ha approvato, a http://www.opensource.org/licenses/, cosicché "approvata-OSI" è uno stato non ambiguo: una licenza c'è o non c'è nella lista.

  Anche la Free Software Foundation tiene aggiornata una lista delle licenza a La FSF classifica le licenza a http://www.fsf.org/licensing/licenses/license-list.html. La FSF classifica le licenza non solo in base al fatto se sono libere, ma se sono compatibili con la GNU General Public License. La compatibilità GPL è un importante argomento, trattato in sezione chiamata «La GPL e la compatibilità di Licenza» più avanti in questo capitolo.

proprietario, closed-source
: L'opposto di “free” e “open source”. Ciò significa distribuito sotto i termini delle licenze tradizionali, basate sul costo, dove gli utilizzatori pagano per una copia, o sotto altri termini restrittivi sufficienti per impedire alle dinamiche open source di operare. Anche il software distribuito gratis può essere proprietario, se la sua licenza non permette la libera redistribuzione e modifica.

  Generalmente “proprietario” e “closed source” sono sinonimi. Comunque in più “closed source” implica che il codice sorgente non può persino essere visto: poiché il codice sorgente non può

essere visto nella maggior parte dei software proprietari, questa è normalmente una differenza senza distinzioni. Comunque, a volte, qualcuno rilascia del software proprietario sotto una licenza che permette ad altri di vedere il codice sorgente. Con confusione essi lo chiamano “open source” o “quasi open source”, ecc.. , ma ciò è ingannevole. La *visibilità* del codice sorgente non è il problema; la domanda importante è cosa potete fare con esso. Così la differenza fra proprietario e closed source è in gran parte irrilevante e i due termini possono essere trattati come sinonimi.

A volte *commerciale* è usato come sinonimo di “proprietario”, ma, parlando appropriatamente, i due termini non sono la stessa cosa. Il software libero può essere software commerciale. Dopotutto il software libero può essere venduto, fin tanto che agli acquirenti non è impedito di dar via copie essi stessi. Esso può essere commercializzato in altre maniere, per esempio vendendo assistenza, servizi, e certificazione. Ci sono compagnie miliardarie in dollari costruite sul software libero oggi, cosicché esso non è né in modo innato anti-commerciale né anti-compagnia. D'altra parte esso *è* anti-proprietario per natura, e questa è la chiave per cui differisce dai modelli di licenza per-copia

di pubblico dominio
Non avente un intestatario di copyright, nel senso che non c'è nessuno che abbia i diritto di limitare la copia dell'opera. Essere di pubblico dominio non è la stessa cosa di non avere un autore, e, anche se l'autore o gli autori dell'opera hanno deciso di metterla in pubblico dominio, questo non cambia il fatto che essi la hanno scritta.

Quando un'opera è di pubblico dominio, del materiale facente parte di essa può essere incorporato in un'opera protetta da diritto d'autore, e quindi *quella copia* di materiale è coperta da diritto d'autore come l'intera opera. Ma ciò non cambia la disponibilità del lavoro originale, che rimane di pubblico dominio. Quindi il rilasciare qualcosa come di pubblico dominio è tecnicamente un modo per renderla “libera”, in accordo con la maggior parte delle organizzazioni che certificano software libero. Comunque usualmente ci sono buone ragioni per usare una licenza invece di rendere di pubblico dominio: anche con il software libero certe limitazioni possono essere utili non solo all'intestatario del copyright, ma anche ai destinatari, come chiarisce la prossima sezione.

copyleft
Una licenza che usa la legge sul copyright per ottenere un risultato opposto al copyright tradizionale. A seconda di quello che chiedete, questo significa sia licenze che permettono le libertà in discussione qui, sia, più strettamente, licenze che non solo permettono quelle libertà, ma che *le obbligano*, con lo stipulare che le libertà devono viaggiare con l'opera. La Free Software Foundation usa esclusivamente la seconda definizione; altrove è uguale: la maggior parte delle persone usano il termine allo stesso modo della Free Software Foundation;—ma altre, inclusi coloro ohe scrivono per i media prevalenti, tendono ad usare la prima definizione. Non è chiaro che chiunque usi il termine sia cosciente che bisogna fare la distinzione.

L'esempio canonico di più stretta e decisa definizione è la GNU General Public License che stabilisce che ogni lavoro derivato deve essere rilasciato sotto la GPL; vedere sezione chiamata «La GPL e la compatibilità di Licenza» più avanti in questo capitolo per maggiori ragguagli.

Aspetti Delle Licenze

Sebbene ci siano molte licenze di software libero disponibili, nei punti più importati esse dicono la stessa cosa: che chiunque può modificare il codice, che chiunque può redistribuirlo sia nella forma originale che in quella modificata, e che i detentori del copyright non forniscono alcuna garanzia (evitare responsabilità dato che le persone potrebbero far girare versioni modificate senza conoscerle). La differenza fra le licenze si riassume in due spesso-ricorrenti questioni:

compatibilità con licenze proprietarie
Alcune licenze libere permettono al codice da esse coperto di essere usato in programmi proprietari. Ciò non intacca i termini della licenza del software proprietario: è sempre proprietaria, succede solo

che contiene del software non proprietario. La licenza Apache, la licenza X Consortium, la licenza stile-BSD e la licenza stile MIT sono tutte licenze proprietarie-compatibili.

compatibilità con altre licenze libere
: La maggior parte delle licenze libere sono compatibili l'una con l'altra, nel senso che il codice sotto una licenza può essere combinato con il codice sotto un'altra licenza, e il risultato distribuito sotto un'altra licenza senza violare i termini delle altre. La principale eccezione a queste è la GNU General Public License che richiede che ogni opera che usa un codice rilasciato sotto la GPL sia distribuito sotto la GPL e senza che si aggiunga ulteriore restrizione oltre quello che la GPL richiede. La GPL è compatibile con alcune licenze libere, ma non con altre. Ciò è discusso con maggiori dettagli in sezione chiamata «La GPL e la compatibilità di Licenza» più avanti in questo capitolo.

imposizione del riconoscimento
: Alcune licenze libere stabiliscono che un uso del codice protetto sia accompagnato da una nota, la cui posizione e presentazione sono usualmente specificate, che dà il riconoscimento agli autori o ai detentori del copyright. Queste licenze sono tuttavia proprietario-compatibili: esse non richiedono che il lavoro derivato sia libero, solamente che sia dato il riconoscimento al codice libero.

Protezione del marchio
: Una variante dell'obbligo del riconoscimento. Le licenze a-protezione-del-marchio specificano che il nome del software originale(il suo detentore di copyright, o la loro istituzione, ecc..) può non essere usata dai lavori derivati senza previa autorizzazione scritta. Sebbene l'obbligo del riconoscimento insista sul fatto che siano usati certi nomi, e la protezione del marchio sul fatto che non siano usati, essi sono ambedue espressione dello stesso desiderio: che la reputazione del codice originale sia tutelata e trasmessa, senza essere offuscata dall'associazione.

Protezione dell'"integrità artistica"
: Certe licenze (La Licenza Artistica, usata nella maggior parte dell'implementazione del linguaggio Perl e la licenza TeX di Donald Knut, per esempio) richiedono che la modifica e la redistribuzione siano fatte in modo da distinguere chiaramente fra la versione originaria del codice e qualunque modificazione. Esse permettono essenzialmente alcune libertà ma impongono certi requisiti che permettono facilmente di verificare l'integrità del codice originale. Queste licenze non si sono diffuse molto oltre gli specifici programmi per i quali erano state create e non saranno trattate in questo capitolo; sono menzionate qui per ragioni di completezza.

La maggior parte di questi contratti non sono mutuamente esclusivi e alcune licenze ne includono diverse. L'argomento comune a esse è che esse pongono richieste al destinatario in cambio del diritto del destinatario di usare e/o distribuire il codice. Per esempio, alcuni progetti vogliono che il proprio nome e reputazione si trasmettano con il codice, e questo ha il valore di imporre il carico extra di un riconoscimento o una clausola di marchio; a seconda della sua onerosità, questo carico può far nascere nell'utilizzatore un scelta di una licenza che chieda meno.

La GPL e la compatibilità di Licenza

La linea divisoria di gran lunga più netta in fatto di licenze è quella fra proprietarie-compatibili e proprietarie-incompatibili, cioè fra la GNU General Public License e tutte le altre. Poiché il principale obiettivo dei creatori della GPL è la promozione del software libero, essi hanno deliberatamente creato la licenza per rendere impossibile mischiare il codice sotto licenza GPL con programmi proprietari. Specificatamente, fra i requisiti della GPL (vedere http://www.fsf.org/licensing/licenses/gpl.html per il suo testo completo) ci sono questi due:

1. Ogni lavoro derivato— cioè ogni lavoro che contenga una quantità di codice non facile sotto licenza GPL deve essere distribuito sotto licenza GPL.

2. Non ulteriori restrizioni devono essere poste sulla redistribuzione sia del codice originale sia di quello derivato. (L'espressione esatta è: "Non potete imporre ulteriori restrizioni sull'eserciziodei diritti concessi o affermati sotto questa Licenza")

Con queste condizioni la GPL riesce a rendere la libertà contagiosa. Una volta che il programma sia protetto dalla GPL, i suoi termini di redistribuzione sono *virali*—essi passano avanti a ogni altra cosa in cui il codice è incorporato, rendendo praticamente impossibile usare il codice sotto GPL in programmi closed-source. Comunque le stesse clausole rendono la GPL incompatibile con altre licenze libere. Il modo solito in cui ciò avviene è quello in cui altre licenze impongono un requisito—per esempio una clausola di riconoscimento che richiede che l'autore originale sia menzionato in qualche modo— che è incompatibile con il “Non potete imporre interiori restrizioni..” della GPL. Dal punto di vista della Free Software Fondativo, queste conseguenze di second'ordine son auspicabili, o almeno non spiacevoli. La GPL non solo mantiene il suo software libero, ma in pratica fa in modo che il vostro software sia un agente che spinge *altri* software a far valere la libertà.

La domanda se sia questo un buon modo di promuovere il software libero è una delle più continue guerre sante in Internet (vedere sezione chiamata «Evitare le Guerre Sante» in Capitolo 6, *Comunicazione*), non vogliamo andare a fondo su questo. La cosa più importante per i nostri propositi è che la compatibilità GPL è un importante problema quando si sceglie una licenza. La GPL è di gran lunga la più importante licenza open source; a http://freshmeat.net/stats/#license, È al 68% e la successiva licenza più importante è al 6%. Se volete che il vostro codice sia mescolato liberamete con codice sotto GPL— e c'è una gran quantità di codice sotto GPL là—allora potete scegliere una licenza GPL compatibile. La gran parte delle licenze GPL compatibili sono anche proprietarie compatibili: cioè codice sotto una tale licenza può può essere usato sotto una licenza GPL, e può essere usata in un programma proprietario. Certamente, il *risultato* di questo mixing non sarebbe compatibile con qualsiasi altra licenza, poiché uno sarebbe sotto la GPL e un altro sarebbe sotto una licenza closed source. Ma quello che interessa è che si applica solo ai lavori derivati e non al codice che voi che distribuite per cominciare.

Fortunatamente la Free Software Foundation tiene aggiornata una lista delle licenze compatibili con la GPL e di quelle non compatibili, a http://www.gnu.org/licenses/license-list.html. Tutte le licenze discusse in questo capitolo sono in quella lista, da una parte o dall'altra.

Scegliere una Licenza

Quando scegliete una licenza per il vostro progetto, per quanto possibile, usate una licenza esistente invece di crearne un'altra. Ci sono due ragioni per cui le licenze esistenti sono le migliori:

- La familiarità. Se usate una delle tre o quattro licenze più popolari, la gente non avrà la sensazione di leggere roba giuridica per usare il vostro codice, perché lo ha già fatto per quella licenza da tempo.

- La qualità. A meno che non abbiate un team di legali a disposizione, è improbabile che voi spuntiate con una licenza legalmente solida. Le licenze menzionate sono il frutto di tanto pensare e di tanta esperienza; a meno che il vostro progetto non abbia bisogno di cose speciali, è improbabile che voi fareste di meglio.

Per applicare una di queste licenze al vostro progetto, vedete sezione chiamata «Come Applicare Una Licenza Al Vostro Software» in Capitolo 2, *Partenza* .

La licenza MIT / X Window System

Se il vostro obiettivo è quello che il vostro codice sia accessibile dal più grande numero di sviluppatori e di lavori derivati e non date importanza al fatto che sia usato in programmi proprietari, scegliete la

licenza MIT / X Window System (così chiamata perché è la licenza sotto la quale il Massachusetts Institute of Technology rilasciò il codice del sistema X Window). Il messaggio fondamentale di questa licenza è “Siete liberi di usare questo codice in qualunque modo vogliate”. Essa è compatibile con la GNU GPL, ed è breve, semplice e facile da capire:

```
Copyright (c) <anno> <detentori del copyright>

Il permesso è così garantito, gratuito, a qualunque persona che ottenga
una copia di questo software e i file associati di documentazione (il
“Software”), per commerciare col Software senza restrizioni, inclusi
i diritti illimitati all'uso, alla copia, alla modifica, all'unione, alla
alle sotto licenze e/o a vendere copie del Software, e a
permettere a persone alle quali il Software è fornito a fare lo stesso,
soggetto alle seguenti condizioni:

L'avviso di copyright summenzionato e questo avviso di permesso devono ess
in tutte le copie o porzioni del Software sottostante.

IL SOFTWARE E' FORNITO “COSI' COM'E'”, SENZA GARANZIA DI ALCUN GENERE,
ESPRESSA O IMPLICITA, INCLUDENDO E SENZA ESSERE LIMITATO ALLE GARANZIE
DI POTER ESSERE VENDUTO, L'IDONEITA' A UN PARTICOLARE FINE E
LA NON VIOLAZIONE DEI DIRITTI ALTRUI. IN NESSUN CASO GLI AUTORI O I DETENT
SARANNO RESPONSABILI PER QUALSIASI RECLAMO, DANNO O ALTRA RESPONSABILITA',
DEL CONTRATTO, IN SITUAZIONE NON CONTEMPLATE NEL CONTRATTO O ALTRO, CHE NA
O IN RIFERIMENTO AL SOFTWARE O ALL'USO O AD ALTRE QUESTIONI RIGUARDANTI IL
```

(Presa da http://www.opensource.org/licenses/mit-license.php.)

La GNU General Public License

Se preferite che il codice del vostro progetto non sia usato in programmi proprietari o se almeno non vi interessa se venga usato o no in programmi proprietari, scegliete la GNU General Public License (http://www.fsf.org/licensing/licenses/gpl.html). La GPL è probabilmente la licenza più largamente usata nel mondo oggi. Questo fatto di essere riconosciuta all'istante è di per se stesso uno dei più importanti vantaggi della GPL.

Quando scrivete una libreria con l'intendimento che sia usata come parte di altri programmi, considerate attentamente se le restrizioni imposte dalla GPL siano in linea con gli obiettivi del vostro progetto. In alcuni casi—per esempio, quando state cercando di sostituire una libreria proprietaria concorrente che fa le stesse cose—può essere molto più strategico licenziare il vostro codice in modo che possa essere mescolato con programmi proprietari, anche se invece non vorreste ciò. La Free Software Foundation ha anche foggiato una alternativa alla GPL in queste circostanze: la *GNU GPL per librerie*, più avanti rinominata *GNU GPL Minore* (la maggior parte della gente usa l'acronimo *LGPL*, in ogni caso). La LGPL ha meno stringenti restrizioni della GPL, e può essere mescolata facilmente con codice non libero. Comunque, è un po' più complessa e ha bisogno di qualche tempo per essere compresa, cosicché se vi state orientando per usare la GPL, vi consiglio di usare la licenza tipo MIT/X.

La GNU affero GPL: Una Versione della GNU GPL per codice Lato Server

Nel 2007 la Free Software Foundation rilasciò una variante della GPL chiamata *GNU Affero GPL* (http://www.fsf.org/licensing/licenses/agpl.html)[1]. Il suo proposito era imporre la condivisione di clausole tipo GNU al crescente numero di compagnie che offrivano servizi hostati—software che girava su loro servers, con cui gli utilizzatori interagiscono solo sul network, e che quindi non era mai distribuito come eseguibile o codice sorgente. Molti di tali servizi avevano usato software sotto GPL, spesso con modifiche, ma non avevano da mettere in comune i loro cambiamenti col mondo perché non distribuivano nessun codice.

La soluzione GNU AGPLv3's a ciò era prendere la normale GPL e aggiungervi la clausola “Interazione del Network Remoto”, che stabiliva: "*...se modificate il programma, la vostra versione modificata deve in modo preminente offrire a tutti gli utilizzatori l'interazione con esso da remoto attraverso una rete di computers ...una opportunità di ricevere il Corrispondente Sorgente della vostra versione ...gratuitamente, attraverso qualche mezzo standard o personalizzato per facilitare la copia del software.*" This expanded the GPL's enforcement powers into the new world of application service providers. The Free Software Foundation recommends that the GNU AGPLv3 be used for any software that will commonly be run over a network.

Ciò allargò il potere di impegnare da parte della GPL nel nuovo mondo dei provider di servizi applicazione. La Free Software Foundation raccomanda che la GNU AGPLv3 venga usata per ogni software che sarà fatto girare su un network. Notate che la AGPLv3 non è direttamente compatibile con la GPLv2 (mentre è compatibile con la GPLv3, certo). Comunque la maggior parte dei software sotto licenza GPLv2 contengono la clausola “o ogni altra versione successiva”, così voi potete tramutarla nella GPLv3, se e quando avete bisogno di mescolarli con codice AGPLv3. Comunque, se avete bisogno di mescolarli con programmi strettamente sotto licenza GPLv2 (cioè senza la clausola “o ogni altra versione successiva”), allora la AGPLv3 non funzionerà.

Sebbene la storia della AGPLv3 sia un po' complicata, la licenza in se stessa è semplice: è giusto la GPLv3 con una clausola extra sulla interazione col network. L'articolo della Wikipedia sulla AGPLv3 è eccellente: http://en.wikipedia.org/wiki/Affero_General_Public_License

La GPL è libera o non libera?

Una conseguenza della scelta della GPL è la possibilità—piccola ma non infinitamente piccola che voi stessi siate invischiati nella disputa se la GPL sia “libera” o meno, dato che essa pone delle restrizioni su ciò che potete fare con il codice—cioè la restrizione che il codice non può essere redistribuito sotto ogni altra licenza. Per qualcuno, l'esistenza di questa restrizione significa che la GPL è “meno libera” di licenze più permissive come la licenza MIT/X. Dove questo argomento funziona, certamente, è che, poiché “più libero” è meglio che “meno libero” (dopotutto chi non è a favore della libertà?), conseguentemente queste licenze sono migliori della GPL.

Questo dibattito è un'altra popolare guerra santa (vedere sezione chiamata «Evitare le Guerre Sante» in Capitolo 6, *Comunicazione*). Evitate di parteciparvi, almeno nei forum di progetto. Non tentate di provare che la GPL è meno libera, altrettanto libera, o più libera di altre licenze. Piuttosto mettete l'accento sulle ragioni per cui il vostro progetto ha scelto la GPL. Se il fatto di essere riconosciuta per una licenza è una ragione, ditelo. Se l'obbligo di una licenza libera per lavori derivati è anche un ragione, dite anche questo, ma rifiutatevi di essere coinvolti sulla discussione se ciò rende il codice più o meno

[1] La storia della licenza e del suo nome è un po' complicata. La prima versione della licenza fu rilasciata da Affero, Inc, che la basò sulla GNU GPL versione 2. A questa si fece comunemente riferimento come AGPL. Più tardi la Free Software Foundation decise di adottarne l'idea, ma fino ad allora essa aveva rilasciato la versione 3 della sua GNU GPL, così basò la sua nuova licenza Afferoizzata su quella e la chiamò "GNU AGPL". La vecchia licenza Affero è più o meno deprecata adesso. Se volete clausole tipo Affero, dovreste usare la versione GNU. Per evitare ambiguità, chiamatela "AGPLv3”, "GNU AGPL", o con la massima precisione, "GNU AGPLv3".

"libero". La libertà è un argomento complesso e serve a poco parlare di esso se la terminologia sta per essere usata come pretesto al posto della sostanza.

Poiché questo è un libro e non argomento di mailing list, comunque ammetterò di non aver mai capito l'argomento “La GPL non è libera”. La sola restrizione che essa impone è che la gente non imponga *ulteriori* restrizioni. Dire che ciò dà luogo a minore libertà a me è sempre sembrato come dire che bandire la schiavitù riduce la libertà, perché impedisce a certa gente di possedere schiavi.

(Oh, se siete coinvolti in un a discussione su questo, non rilanciate facendo analogie incendiarie.)

Cosa sulla Licenza BSD?

Una discreta quantità di software open source è distribuito sotto *la licenza BSD* (o, a volte una *licenza stile BSD*). La licenza originale BSD fu usata dalla Berkeley Software Distribution, con essa l'Università della California rilasciò importanti parti dell'implementazione Unix. La licenza (il testo esatto può essere visto nella sezione 2.2.2. di http://www.xfree86.org/3.3.6/COPYRIGHT2.html#6) era simile nello spirito alla licenza MIT/X, eccetto che in una clausola:

> *Tutti i materiali pubblicitari devono mostrare il seguente riconoscimento: Questo prodotto include software sviluppato dall'Università della California, Laboratorio Lawrence Berkeley.*

La presenza di quella clausola non solo rese la licenza originale BSD incompatibile con la GPL, ma stabilì anche un pericoloso precedente: che altre organizzazioni mettono simili clausole pubblicitarie nel *loro* software libero—mettendo il nome della propria organizzazione al posto della "l'Università della California, Lawrence Berkeley Laboratory"—e i ridistributori di software furono investiti da un sempre crescente carico su quello che a loro era richiesto di esibire. Fortunatamente molti dei progetti che usarono quella licenza diventarono consci del problema, e semplicemente eliminarono la clausola pubblicitaria. Nel 1999 anche l'Università della California fece lo stesso.

Il risultato fu una licenza BSD rivisitata, che è semplicemente una licenza originale BSD con la clausola pubblicitaria rimossa. Comunque, questa storia rende la frase “licenza BSD” un po' ambigua: si riferisce alla licenza revisionata o alla originale? Questo è il motivo per i quale io preferisco la licenza MIT/X, che è essenzialmente equivalente, e che non soffre di nessuna ambiguità. Comunque c'è forse un motivo per preferire la licenza BSD alla MIT/X, ed è quello che la licenza BSD include questa clausola:

> *Né il nome dell' <ORGANIZZAZIONE> né il nome dei suoi collaboratori può essere usato per sottoscrivere o promuovere prodotti derivati da questo software senza precedente autorizzazione scritta.*

Non è chiaro il fatto che senza una tale clausola un destinatario avrebbe avuto i diritti di usare il nome del licenziatario in qualche modo, ma la clausola rimuove ogni possibile dubbio. Per organizzazioni preoccupate del controllo del marchio, quindi, la licenza BSD può essere di poco preferibile alla MIT/X. In generale, comunque, una licenza liberale di copyright non implica che il destinatario abbia il diritto di usare o indebolire il vostro marchio— la legge sul copyright e quella sul marchio sono due bestie differenti.

Se volete usare un licenza BSD revisionata, un modello è disponibile a http://www.opensource.org/licenses/bsd-license.php.

L'Assegnazione del Copyright e la Proprietà

Ci sono tre modi per maneggiare la proprietà del copyright per il codice libero e la documentazione che sono stati forniti da molta gente. Il primo è ignorare la questione del copyright completamente (io non lo raccomando). Il secondo è raccogliere un *contratto legale di collaborazione* (*CLA*) da ogni persona

che lavora al progetto, che garantisce esplicitamente al progetto il diritto di usare quel contributo della persona. Questo e usualmente sufficiente per la maggior parte dei progetti, e la cosa simpatica è che in qualche giurisdizione i CLA possono essere inviati per email. Il terzo è acquisire le vere cessioni dei diritti dai collaboratori, di modo che il progetto (cioè qualche entità legale, nonprofit) sia il possessore legale del copyright per ogni cosa. Questo è il modo legalmente più inattaccabile, ma è anche il più gravoso per i collaboratori; solo pochi progetti insistono su di esso.

Notare che anche anche sotto la proprietà centralizzata del copyright il codice[2] rimane libero, perché le licenze open source non danno al detentore del copyright il diritto di rendere retroattivamente proprietarie tutte le copie del codice. Così anche se il progetto, come entità legale, facesse improvvisamente il dietro front e partisse con il distribuire tutto il codice sotto una licenza restrittiva, ciò non causerebbe un problema per la comunità pubblica. Gli altri sviluppatori partirebbero con una diramazione basata sull'ultima copia libera di codice, e continuerebbe come se nulla fosse successo. Poiché essi sanno che possono farlo, la maggior parte dei collaboratori cooperano quando loro viene chiesto di sottoscrivere un CLA o una assegnazione di copyright.

Non far Nulla

La maggior parte dei progetti non raccolgono CLA o assegnazione di copyright dai loro collaboratori. Invece, accettano codice, ogni qualvolta sembra ragionevolmente chiaro che il collaboratore abbia inteso che esso incorporato nel progetto.

In circostanze normali ciò è regolare. Ma ogni tanto, qualcuno può decidere per la citazione in giudizio per infrazione di copyright, adducendo il fatto che essi sono veri proprietari del codice in questione, e che non hanno mai convenuto che esso fosse distribuito dal progetto sotto una licenza open source. Per esempio il gruppo SCO fece qualcosa di simile per il progetto Linux, vedere http://en.wikipedia.org/wiki/SCO-Linux_controversies per i dettagli. Quando ciò avvenisse, il progetto non avrà la documentazione che mostri che il collaboratore ha formalmente dato il diritto di usare il codice, cosa che potrebbe rendere più difficile una difesa legale.

Gli Accordi di Licenza per i Collaboratori

I CLA probabilmente offrono il miglior compromesso fra sicurezza e convenienza. Un CLA è un modulo elettronico che il collaboratore compila e invia al progetto. In molte giurisdizioni l'invio di una email è sufficiente. Una firma digitale sicura può o non può essere richiesta; consultate un legale per trovare quale metodo sarebbe migliore per il vostro progetto.

La maggior parte dei progetti usano due tipi di CLA leggermente differenti, uno per collaboratori individuali e uno per collaboratori associati. Ma in ambedue i tipi il linguaggio base è lo stesso: il collaboratore concede al progetto *"...una licenza di copyright perpetua, per tutto il mondo, non esclusiva, a costo zero, senza l'onere di pagare un compenso al titolare del copyright, l'irrevocabile licenza di copyright a riprodurre, preparare lavori derivati, mostrare pubblicamente, eseguire pubblicamente, creare una sotto licenza, e distribuirne [i] Contributi come lavori derivati."* In più, potreste ottenere che un legale ratifichi ogni CLA, ma se voi metteste tutti questi aggettivi in essi, potreste probabilmente far bene.

Quando richiedete i CLA dai collaboratori, assicuratevi di puntualizzare che *non* state richiedendo una assegnazione di copyright. In effetti, molti CLA partono col ricordare a chi legge questo:

> *Questo è solo un contratto di licenza; non trasferisce la proprietà del copyright e non cambia i vostri diritti di usare i vostri Contributi per ogni altro proposito.*

Qui ci sono alcuni esempi:

[2]Userò “codice” per riferirmi sia al codice che alla documentazione, d'ora in avanti

- CLA per collaboratori individuali:
  - http://apache.org/licenses/icla.txt
  - http://code.google.com/legal/individual-cla-v1.0.html
- CLA per collaboratori associati:
  - http://apache.org/licenses/cla-corporate.txt
  - http://code.google.com/legal/corporate-cla-v1.0.html

Trasferimento del Copyright

Trasferimento del copyright significa che il collaboratore dà al progetto la proprietà del copyright sul suo contributo. Idealmente ciò viene fatto su carta e anche mediante fax o con invio per posta normale al progetto.

Alcuni progetti insistono sulla piena assegnazione, perché avendo una unica entità legale propria il copyright sull'intero codice base può essere utile se i termini della licenza open source hanno bisogno di essere fatti valere in una corte. Se nessuna unica entità ha il diritto di farlo, tutti i collaboratori potrebbero dover collaborare, ma qualcuno potrebbe non aver tempo o essere rintracciabile quando nasca il problema.

Differenti organizzazioni applicano differenti quantità di rigore nel raccogliere le assegnazioni. Alcune si procurano una dichiarazione informale da parte di un collaboratore su una mailing list di liste pubbliche—qualcosa dall'effetto di “Io in tal modo assegno il copyright in questo codice al progetto, di essere rilasciato in licenza sotto gli stessi termini del restante codice” Almeno un legale con cui ho parlato dice che ciò è in effetti sufficiente, presumibilmente perché avviene in un contesto in cui la concessione del copyright è normale prevista comunque, e perché rappresenta un tentativo *bona fide* da parte del progetto di accertare le vere intenzioni dello sviluppatore. D'altra parte la Free Software Foundation va dalla parte opposta: richiede ai collaboratori di sottoscrivere e mandare per posta un pezzo di carta contenente una formale dichiarazione, a volte per un solo contributo, a volte per il contributo corrente e futuro. Se il collaboratore è un impiegato la FSF richiede che anche il collaboratore lo sottoscriva.

La paranoia dell'FSF è incomprensibile. Se qualcuno viola i termini della GPL incorporando qualcuno del loro software in un programma proprietario, la FSF avrà bisogno di litigare su questo davanti al tribunale, e vuole che i suoi copyrights siano il più inattaccabili possibile quando ciò avviene. Siccome la FSF è la detentrice del copyright di una gran quantità di software popolari, vede ciò come una possibilità reale. Se la vostra organizzazione ha la necessità di essere scrupolosa, è un fatto che solo voi potete decidere, consultandovi con dei legali. In generale, a meno che non ci sia un motivo specifico per cui il vostro progetto abbia bisogno di una piena assegnazione del copyright, andate avanti con i CLA; essi sono più facili per tutti.

Gli Schemi a Doppia Licenza

alcuni progetti tentano di fondare se stessi usando uno schema a doppia licenza, nel quale i lavori derivati proprietari possono pagare il detentore del copyright per i diritti di usare il codice, ma il codice rimane ancora libero per l'uso da parte dei progetti open source. Ciò tende a finanziare bene sia per le librerie di codice sia per applicazioni per computer indipendenti dalla rete, naturalmente. I termini esatti differiscono da caso a caso. Spesso la licenza per i lato libero è la GNU GPL, poiché essa impedisce ad altri di incorporare il codice coperto nobel loro prodotto proprietario senza il permesso del detentore del copyright, ma a volte essa è una licenza personalizzata che ha lo steso effetto. Un esempio della prima è la licenza MySQL, descritta a http://www.mysql.com/company/legal/licensing/; un esempio

della seconda è la strategia di licenza della Sleepycat Software, descritta a http://www.sleepycat.com/download/licensinginfo.shtml.

Potreste starvi domandando: come può un detentore del copyright offrire una concessione di licenza proprietaria per un costo aggiuntivo se i termini della GNU GPL stabiliscono che il codice deve essere disponibile sotto termini meno restrittivi? La risposta è che i termini della GPL sono qualcosa che il detentore del copyright impone a ogni altro; il proprietario è quindi libero di decidere di *non* applicare quei termini a se stesso. Un buon modo di pensare a ciò è immaginare che che il possessore proprietario abbia un numero infinito di copie ammucchiate in un secchio. Ogni volta che lui prende una copia dal secchio per inviarle nel mondo, può decider quale licenza mietervi dentro: GPL, proprietaria o qualche altra. Il suo diritto di fare ciò non è intaccato dalla GPL o da qualche altra licenza open source; è semplicemente un potere garantito dalla legge sul copyright.

L'attrattiva della licenza doppia sta nel fatto che, al meglio delle sue possibilità, fornisce a un progetto di software libero un modo per ottenere una fonte di profitto sicuro. Sfortunatamente ciò può anche interferire con le dinamiche normali di un progetto open source. Il problema è che ogni volontario che dà un contributo sta collaborando con due distinte entità: la versione libera del codice e quella proprietaria. Mentre il collaboratore sarà soddisfatto di collaborare alla versione free, dacché quella è la norma nei progetti open source, egli può sentirsi buffo nel collaborare al flusso di entrata semi proprietario dei qualche altro. L'imbarazzo è esacerbato dal fatto che che nelle licenze doppie, il detentore del copyright ha veramente bisogno di raccogliere copyright formali sottoscritti da tutti i collaboratori, per potersi proteggere da un collaboratore scontento che in un secondo momento reclami una percentuale dei diritti d'autore dalle entrate proprietarie. Il processo di raccolte di queste assegnazioni con un pezzo si carta significa che i collaboratori si confrontano duramente col fatto che essi stanno facendo un lavoro che crea soldi per qualcun altro.

Non tutti i volontari saranno seccati da ciò; dopotutto il loro contributo va all'edizione open source, e in ciò può essere che stia il loro interesse. Tuttavia la licenza doppia è una richiesta da parte del detentore del copyright di assegnare a se stesso uno speciale diritto che gli altri non hanno. Ed è così che spinge al sorgere di tensioni a un certo punto, almeno fra alcuni volontari.

Ciò che sembra avvenire nella pratica è che le compagnie che si basano sul software a licenza doppia non hanno comunità di sviluppo veramente egualitario. Esse conseguono una correzione dei bugs su piccola scala e patch di ripulita da fonti esterne, ma finiscono col fare una gran parte di duro lavoro con le risorse interne. Per esempio, Zack Urlocker, vice presidente del marketing alla MySQL, mi disse che la compagnia finisce col prendere in affitto i più attivi volontari comunque. Così, anche se il prorotto di per se stesso è open source, sotto licenza GPL, il suo sviluppo è più o meno controllato dalla compagnia, sebbene con la (estremamente improbabile) possibilità che qualcuno veramente insoddisfatto dell' uso del software da parte della compagnia potrebbe fare una diramazione dal progetto. In quale grado questo minacci in modo preëstimolante le politiche della compagnia non so, ma ad ogni modo, MySQL non sembra avere problemi di consenso sia nel mondo dell'open source sia oltre.

I Brevetti

I brevetti sono il rilascio parafulmine del momento nel software libero. Perché pongono la sola minaccia reale contro la quale la comunità del software libero non può difendersi. I problemi di copyright e di marchio ci si può sempre fare l'esperienza. Se parte del vostro codice sembra poter usurpare il copyright di qualche altro, voi potete riscrivere quella parte. Se vien fuori che qualcuno che ha il marchio sul nome del vostro progetto, nel peggior caso potete cambiar nome al vostro progetto. Sebbene il cambiamento del nome potrebbe essere un inconveniente temporaneo, non sarebbe un problema nei tempi lunghi, poiché il codice stesso farebbe ancora ciò che ha sempre fatto.

Ma una partente è una completa ingiunzione contro l'implementazione di una certa idea. Non importa chi scrive il codice, né quale linguaggio di programmazione è usato. Una volta che uno ha accusato un

progetto di software libero di infrangere un brevetto, il progetto o deve smettere di implementare una data funzionalità, o trovarsi di fronte a una causa dispendiosa e che assorbe tempo. Poiché gli istigatori di tali cause sono usualmente compagnie con tasche profonde—cioè quelle che hanno le risorse e l'inclinazione ad acquistare i brevetti all'inizio—la maggior parte dei progetti di software libero non può permettersi l'ultima possibilità, e deve capitolare immediatamente anche se si pensa che è molto verosimile che il brevetto sarebbe non sacettibile di tutela giudiziaria nella corte. Per evitare di trovarsi in una tale situazione in primo luogo, i progetti di software libero, stanno incominciando a scrivere codice in modo difensivo, evitando in anticipo algoritmi patentati anche quando essi sono la migliore o l'unica soluzione disponibile per i problemi di programmazione.[3]

Osservazione ed evidenze aneddotiche mostrano che non solo la vasta maggioranza del programmatori open source, ma una maggioranza di *tutti* i programmatori pensano che i brevetti dovrebbero esser abolite completamente.[4] I programmatori open source tendono rendersi conto in maniera particolarmente forte di ciò, e possono rifiutarsi di lavorare a progetti che siano molto associati alla raccolta o all'imposizione dei brevetti. Se la vostra organizzazione raccoglie brevetti di software, allora chiarite, in modo pubblico irrevocabile, che i brevetti non sarebbero mai applicate ai progetti open source, e che sono solo da usare come difesa nel caso che altre parti inizino una procedura legale di infrazione contro la vostra organizzazione. Questa non solo è la cosa giusta da fare, è anche una buona pubblica relazione open source. Per esempio RedHat ha promesso che i progetti open source sono al sicuro dai suoi brevetti, vedere http://www.redhat.com/legal/patent_policy.html.

Sfortunatamente, la raccolta dei brevetti a scopo difensivo è un atto razionale. Il corrente sistema dei brevetti, almeno negli Stati Uniti, è per sua natura un confronto armato: se i vostri concorrenti hanno acquistato un sacco di brevetti, allora la vostra miglior difesa è acquistare un sacco di brevetti a vostra volta, così se siete colpiti da una azione legale di infrazione di un brevetto potete rispondere con una minaccia simile—allora le due parti si siedono ad un tavolo per un accordo commerciale di licenze incrociate in modo che nessuna di esse abbia da pagare qualcosa, eccetto che i legali della loro proprietà intellettuale, ovviamente.

Il danno fatto al software libero dai brevetti è più insidioso che la minaccia diretta allo sviluppo del codice, comunque. I brevetti di software incoraggiano una atmosfera di segretezza fra i progettisti di firmware, che giustamente temono che pubblicando i dettagli della loro interfaccia staranno dando un aiuto ai concorrenti che stanno cercando di dar loro uno schiaffo con azioni legali di infrazione dei brevetti. Questo non è un danno teorico; è avvenuto in modo evidente per lungo tempo nell'industria delle schede video, per esempio. Molti costruttori di schede video sono riluttanti a rilasciare le specifiche di programmazione necessarie per produrre drivers open source ad alte prestazioni per le loro schede, rendendo così impossibile ai sistemi operativi liberi di supportare quelle schede al pieno delle loro potenzialità. Perché i costruttori farebbero ciò? Non ha senso per loro lavorare *contro* il supporto al software; dopotutto la compatibilità con più sistemi operativi può significare solo più vendita di schede. Ma vien fuori che, dietro le porte della stanze di progettazione, questi negozi stanno tutti violando i brevetti l'uno degli altri, a volte con cognizione di fatto e altre inconsapevolmente. I brevetti sono così imprevedibili e così potenzialmente di larga portata che nessun costruttore di schede può mai essere certo do essere al sicuro, anche dopo avere fatto una ricerca del brevetto. Così, i costruttori non osano pubblicare le specifiche delle loro interfacce, perché ciò renderebbe molto più facile per i concorrenti capire se un brevetto è stata violato. (Certamente la natura di questa situazione è tale che non troverete una ammissione scritta da parte di una fonte primaria che ciò sta avvenendo; io lo ho appreso da una personale informazione).

Alcune licenze di software hanno delle clausole per combattere, o almeno scoraggiare, i brevetti di software. La GNU GPL, per esempio, contiene questa espressione:

[3] La Sun Microsystems e l'IBM hanno almeno fatto un gesto nei confronti del problema dall'altra direzione, liberando un gran numero di brevetti, 1600 e 500 rispettivamente—per l'uso da parte delle comunità di software libero. Non sono un legale e perciò non posso valutare la reale utilità di queste concessioni, ma anche se esse sono dei brevetti importanti, e i termini delle concessioni le rendono realmente libere per l'uso da parte dei progetti open source, ciò sarebbe tuttavia solo una goccia nel secchio.

[4] Vedere http://groups.csail.mit.edu/mac/projects/lpf/Whatsnew/survey.html per un tale esame.

> 7. Se, in conseguenza di un giudizio della corte o di una dichiarazione
>   di violazione di un brevetto o per qualche altra ragione (non limitata
>   al problema dei brevetti) sono imposte a voi condizioni (per decisione
>   della corte, accordo o altro) che vadano contro le condizioni di questa
>   licenza, esse non vi esonerano dalle condizioni di questa licenza. Se voi
>   non potete distribuire in modo da soddisfare contemporaneamente i vostri
>   obblighi con questa licenza e ogni altro obbligo pertinente, quindi di
>   conseguenza non potete distribuire punto il Programma. Per esempio, se
>   una licenza brevetto non permettesse la redistribuzione senza compenso
>   del Programma da parte di tutti quelli che ne ricevono copie direttamente
>   o indirettamente tramite voi, allora l'unico modo per voi di soddisfare s
>   essa che questa licenza sarebbe astenersi del tutto dalla distribuzione d
>
> [...]
>
> Non è proposito di questa sezione indurvi a violare alcun
> brevetto o altri giusti diritti di proprietà o a contestare la validità
> di alcuni di questi diritti; questa sezione ha il solo proposito di
> protegger il sistema di distribuzione del software libero che è
> implementato dalla prassi delle licenze pubbliche. Molte persone hanno dato
> generosi contributi alla larga estensione di software distribuito
> con questo sistema confidando nella applicazione coerente di quel
> sistema; dipende dall'autore/donatore decidere se vuol
> distribuire il software con altri sistemi e la licenza non può
> imporre quella scelta.

anche La Licenza Apache, Versione 2.0 (http://www.apache.org/licenses/LICENSE-2.0) ontiene requisiti anti-brevetto. Primo, stabilisce che chiunque distribuisca codice sotto la licenza debba implicitamente includervi un brevetto che liberi dal pagamento di qualsiasi copia per ogni brevetto che potrebbe contenere ciò che potrebbe applicarsi al codice. Secondo, e con moltissimo ingegno, essa punisce chiunque inizi un reclamo di violazione del brevetto sul lavoro coperto, ponendo termine automaticamente alla sua licenza brevetto nel momento in cui tale reclamo è fatto:

> 3. Concessione di Licenza Brevetto. Soggetto ai termini e alle condizioni
>   di questa Licenza ogni collaboratore concede con questo mezzo a Voi una l
>   brevetto perpetua, per tutto il mondo, non esclusiva, gratuita, che liber
>   dal pagamento di qualsiasi copia, irrevocabile (eccetto quanto stabilito
>   questa sezione) a fare, aver fatto, usare, offrire per vendita, vendere,
>   e in altro modo trasferire il Lavoro, dove tale licenza si applica solo a
>   diritti di brevetto che danno facoltà di concedere diritti da parte di ta
>   Collaboratore e che sono necessariamente violati dal suo Contributo(i) da
>   o in combinazione con il suo Contributo(i) al Lavoro al quale tale Contri
>   fu inviato. Se voi create una controversia con qualche entità (incluso un
>   reclamo contro la parte che sta al vostro fianco o un contro reclamo in u
>   asserendo che il Lavoro o il Contributo incorporato nel Lavoro costituisc
>   una violazione del brevetto diretta o derivante dal contributo, allora og
>   licenza brevetto concessa a Voi sotto questa Licenza per quel Lavoro sarà
>   terminata dalla data in cui tale controversia è depositata.

Sebbene ciò sia utile, sia da un punto di vista legale che come politica, per costruire la difesa dei brevetti nelle licenze di software libero a questo modo, alla fine questi passi non saranno sufficienti a dissipare

l'effetto scoraggiante che la minaccia di dibattimento in tribunale ha sul software libero. Ciò creerà solo dei cambiamenti nella sostanza o nell'interpersonale della legge sui brevetti. Per apprendere sul problema e come sta venendo combattuto, andare a http://www.nosoftwarepatents.com/. L'articolo della Wikipedia http://en.wikipedia.org/wiki/Software_patent ha anche un sacco di utili informazioni sui brevetti di software. Io ho scritto anche un post in un blog che sintetizza gli argomenti contro i brevetti sul software a Questo capitolo è stato solo una introduzione ai problemi di licenza sul software libero, a http://www.rants.org/2007/05/01/how-to-tell-that-software-patents-are-a-bad-idea/.

Ulteriori Risorse

Questo capitolo è stato solo una ontroduzione ai problemi delle licenze di software libero. Sebbene io penso che esso abbia sufficienti informazioni per permettervi di partire col vostro progetto, ogni seria ricerca sui problemi delle licenze sviscererà rapidamente ciò che questo libro può fornire. Qui c'è una ulteriore lista di risorse sulle licenze open source:

- *Comprendere l'Open Source e le Licenze del Software Libero* di Andrew M. St. Laurent. Edito da O'Reilly Media, prima edizione Agosto 2004, ISBN: 0-596-00581-4.

  Questo è un libro completo sulle licenze open source in tutta la loro complessità, che include molto argomenti omessi in questo capitolo. Vedere http://www.oreilly.com/catalog/osfreesoft/ per i dettagli.

- *Rendete il Vostro Software Open Source GPL Compatibile. Oppure.* di David A. Wheeler, a http://www.dwheeler.com/essays/gpl-compatible.html.

  Questo capitolo tocca anche altre numerose questioni di licenza, e ha una alta densità di links ecellenti.

- http://creativecommons.org/

  La Creative Commons che promuove una larga estensione di copyrights più flessibili e liberali di quanto la pratica del copyright incoraggi. Essa non offre copyright per il software , ma per il testo, l'arte e la musica, tutte accessibili con un selezionatore di di licenze di facile uso; alcune licenze sono copyleft, alcune non copyleft, ma tuttavia libere, altre sono tradizionali copyrights, ma rilasciate con qualche restrizione. Il sito della Creative Commons dà spiegazioni estremamente chiare su ciò cui si riferisce. Se io avessi da scegliere un sito per dimostrare le più larghe implicazioni filosofiche del movimento del software libero, esso sarebbe questo.

Appendice A. Sistemi di Controllo di Versione Liberi

Questi sono tutti i sistemi di controllo di versione di cui ero a conoscenza alla metà del 2007. L'unico che uso regolarmente è Subversion. Ho poca o nessuna esperienza con la maggior parte di questi sistemi, tranne Subversion e CVS; le informazioni qui sono prese dai loro siti web. Vedi anche http://en.wikipedia.org/wiki/List_of_revision_control_software.

CVS — http://www.nongnu.org/cvs/

CVS è in giro da molto tempo, ed è già familiare a molti sviluppatori. Nei suoi giorni fu rivoluzionario: fu il primo sistema di controllo di versione open source con accesso da Internet per gli sviluppatori (per quanto ne so), e il primo ad offrire checkout anonimi di sola lettura, il che diede ai nuovi sviluppatori un facile modo per venire coinvolti nei progetti. CVS tiene le versioni solo dei file, non delle directory; offre la possibilità di fare branch, mettere tag, e una buona prestazione sul lato client, ma non gestisce molto bene file grandi o binari. Non supporto commit atomici.*[Disclaimer: sono stato attivo nello sviluppo di CVS per circa cinque anni, prima di aiutare ad iniziare il progetto Subversion per rimpiazzarlo.]*

Subversion — http://subversion.tigris.org/

Subversion fu scritto inizialmente e principalmente per essere un rimpiazzo per CVS—cioè, per affrontare il controllo di versione circa allo stesso modo in cui CVS lo fa, ma senza i problemi e la mancanza di feature che la maggior parte delle volte annoiavano gli utenti di CVS. Uno degli obbiettivi di CVS è far percepire indolore la transizione a Subversion alla gente già abituata a CVS. Non c'è spazio qui per andare nel dettaglio delle funzionalità di Subversion; vedi il suo sito per maggiori informazioni. *[Disclaimer: sono coinvolto nello sviluppo di Subversion, ed è l'unico di questi sistemi che uso regolarmente.]*

SVK — http://svk.elixus.org/

Anche se costruito basandosi su Subversion, SVK probabilmente assomiglia a qualcuno dei sistemi decentralizzati sotto più di quanto lo faccia Subversion. SVK supporta lo sviluppo distribuito, commit locali, una sofisticata fusione dei cambiamenti, e l'abilità di rispecchiare alberi da altri sisetmi di controllo di versione non-SVK. Vedi il sito per i dettagli.

Mercurial — http://www.selenic.com/mercurial/

Mercurial è un sistema di controllo di versione distribuito che offre, tra le altre cose, "completo cross_indexing di file e modifiche; protocolli di sincronizzazione HTTP e SSH efficienti per CPU e larghezza di banda; unione di branch arbitraria; interfaccia web stand-alone integrata; [portabilità verso] UNIX,MacOs X and Windows" e altro (la precedente lista di funzionalità è stata parafrasata dal sito web di Mercurial).

GIT — http://git.or.cz/

GIT è un progetto iniziato da Linus Torvalds per gestire l'albero del codice del kernel di Linux. All'inizio GIT era piuttosto strettamente focalizzato sui bisogni dello sviluppo kernel, ma si è espanso oltre ed è ora usato da progetti diversi dal kernel Linux. La sua home page dice che è ".. disegnato per

gestire progetti molto vasti con velocità ed efficienza; è usato principalmente per vari progetti open source, tra cui si nota il kernel Linux. Git cade nella categoria degli strumenti di gestione distribuita del codice sorgente, simile ad esempio a GNU Arch o Monotone (o BitKeeper nel mondo proprietario). Ogni directory di lavoro di Git è un repository completo con tutte le funzionalità di tracciamento di revisione, non dipendente dall'accesso alla rete o ad un server centrale."

Bazaar — http://bazaar.canonical.com/

Bazaar è ancora in sviluppo. Sarà un'implementazione del protocollo GNU Arch, manterrà la compatibilità con il protocollo GNU Arch quando evolve, e lavora con la comunità GNU Arch per ogni cambiamento di protocollo che potrebbe essere richiesto per la facilità dell'utente.

Bazaar-NG — http://bazaar-ng.org/

Bazaar-NG (or bzr) è al momento sotto sviluppo da parte di Canonical (http://canonical.com/). Offre una scelta tra lavoro centralizzato o decentralizzato all'interno di un singolo progetto. Per esempio, quando sei in ufficio, puoi lavorare su di un branch centrale condiviso; per cambiamenti sperimentali o lavoro offline, puoi creare un branch sul tuo laptop e fare il merge più avanti.

Darcs — http://abridgegame.org/darcs/

"David's Advanced Revision Control System è l'ennesimo rimpiazzo di CVS. E' scritto in Haskell, ed è usato su Linux, MacOS X, FreeBSD, OpenBSD and Microsoft Windows. Darcs include uno script cgi, che può essere usato per vedere i contenuti del vostro repository."

Arch — http://www.gnu.org/software/gnu-arch/

GNU Arch supporta lo sviluppo sia distribuito che centralizzato. Gli sviluppatori committano le loro modifiche ad un "archivio", che può essere locale, e le modifiche possono essere mandate e prese da altri archivi quando i gestori di questi archivi vedono che vanno bene. Così come una tale metodologia implica, Arch ha un supporto ai merge più sofisticato di CVS. Arch permette anche di creare facilmente branch di archivi a cui non si ha l'accesso al commit. Questo è solo un breve riassunto; vedi le pagine web di Arch per i dettagli.

monotone — http://www.venge.net/monotone/

"monotone è un sistema di controllo di versione libero. fornisce un deposito di versione semplice e transazionale sul file singolo, con un protocollo di operazione completamente disconnesso e di efficiente sincronizzazione peer-to-peer. capisce i merge sensibili alla storicizzazione, branch leggeri, revisione del codice integrata e test di terze parti. usa una nominazione di versione crittografica e certificati RSA lato client. ha un buon supporto per l'internazionalizzazione, non ha dipendenze esterne, va su linux, solaris, osx e windows, ed ha la licenza GNU GPL."

Codeville — http://codeville.org/

"Perchè l'ennesimo sistema di controllo di versione? Tutti gli altri sistemi di controllo di versione richiedono che teniate con attenzione traccia delle relazioni tra i branch così da non dover ripetutamente fare il merge degli stessi conflitti. Codeville è molto più anarchico. Vi permette di aggiornare dal vostro commit ad ogni repository in ogni momento senza inutili nuovi merge."

"Codeville funziona creando un identificativo per ogni cambiamento che viene fatto, e ricordando la lista di tutti i cambiamenti che sono stati applicati ad ogni file e l'ultimo cambiamento che ha modificato

ogni riga in ogni file. Quando c'è un conflitto, controlla per vedere se uno dei due è già stato applicato all'altro, e se così fa vincere l'altro automaticamente. Quando c'è un conflitto di versione non risolvibile automaticamente, Codeville si comporta praticamente nello stesso modo di CVS."

Vesta — http://www.vestasys.org/

"Vesta è un sistema SCM [Software Configuration Management] portabile orientato al supporto di sviluppo di sistemi software di praticamente ogni peso, da abbastanza piccoli (meno di 10,000 righe di codice) a molto grandi (10,000,000 righe di codice)."

"Vesta è un sistema maturo. E' il risultato di oltre 10 anni di ricerca e sviluppo al Compaq/Digital Systems Research Center, ed è stato usato in produzione dal gruppo di microprocessori Compaq's Alpha per oltre due anni e mezzo. Il gruppo Alpha ha avuto oltre 150 sviluppatori attivi in due siti distanti migliaia di miglia, sulle coste est e ovest degli Stati Uniti. Il gruppo ha usato Vesta per gestire i build con 130 MB di codice, ognuno generante 1.5 GB di dati derivati. I build fatti al sito ad est in un giorno medio producevano circa 10-15 GB di dati derivati, tutti gestiti da Vesta. Anche se Vesta è stato progettato con in mente lo sviluppo software, il gruppo Alpha dimostrò la flessibilità del sistema usandolo per lo sviluppo di hardware, controllando i loro file di linguaggio di descrizione hardware nella funzionalità di controllo di codice sorgente di Vesta e costruendo simulatori e altri oggetti derivati nel builder di Vesta. I membri del vecchio gruppo Alpha, ora parte di Intel, stanno continuando ad usare Vesta oggi in un progetto di microprocessore."

Aegis — http://aegis.sourceforge.net/

"Aegis è un sistema di gestione di configurazione software basato su transazione. Fornisce un framework in cui un gruppo di sviluppatori può lavorare su molte modifiche ad un programma in maniera indipendente, ed Aegis coordina l'integrazione di questi cambiamenti nel codice principale del programma, con le minori discrepanze possibili."

CVSNT — http://cvsnt.org/

"CVSNT è un avanzato sistema di controllo di versione multipiattaforma. Compatibile con il protocollo standard industriale CSV ora supporta molte nuove funzionalità. ... CVSNT è Open Source, software libero con licenza GNU General Public License." La sua lista di funzionalità include autenticazione attraverso tutti i protocolli standard CVS, più gli specifici di Windows SSPI e Active Directory; supporto al trasporto sicuro, via sserver o SSPI criptato; cross-platform (va in ambienti Windows e Unix); la versione NT è completamente integrata con il sistema Win32; il processamento MergePoint significa mai più tag di cui fare il merge; sotto sviluppo attivo.

META-CVS — http://users.footprints.net/~kaz/mcvs.html

"Meta-CVS è un sistema di controllo di versione costruito attorno a CVS. Anche se mantiene molte delle funzionalità di CVS, incluso l'intero supporto alla reta, è più capiente di CVS, e più facile da usare." Le funzionalità elencate sul sito web di META-CVS include: versionamento della struttura delle directory, gestione dei tipi di file migliorata, fare merge e branch in modo più facile per l'utente, supporto ai link simbolici, liste di proprietà attaccate ai dati versionati, importazione di dati di terze parti migliorata, e facile processo di upgrade da CVS.

OpenCM — http://www.opencm.org/

"OpenCM è progettato come un rimpiazzamento sicuro, ad alta integrità per CVS. Una lista di funzionalità chiave può essere trovata sulla pagina delle funzionalità. Pur non essendo 'ricco di

funzionalità' come CVS, supporta alcune utili cose che mancano in CVS. In breve, OpenCM fornisce un supporto di prima scelta per rinomine e configurazione, autenticazione crittografica e controllo degli accessi, e branch nativi."

Stellation — http://www.eclipse.org/stellation/

"Stellation è un avanzato ed estensibile sistema di gestione di configurazione di software, originariamente sviluppato alla IBM Research. Mentre Stellation fornisce tutte le funzionalità standard disponibili in ogni sistema SCM [Source Code Management, gestione di codice sorgente, ndt], spicca per un numero di funzionalità avanzate, come la gestione del cambiamento orientato all'attività (task oriented), versionamento di progetto consistente e branch facile, inteso per facilitare lo sviluppo dei sistemi software da parte di ampi gruppi di sviluppatori scarsamente coordinati."

PRCS — http://prcs.sourceforge.net/

"PRCS, Project Revision Control System (sistema di controllo di revisione di progetto), è la facciata di un insieme di strumenti che (come CVS) forniscono un modo per gestire insiemi di file e directory come un'entità, preservando le versioni coerenti dell'intero insieme. ... Il suo proposito è simile a quello di SCCS, RCS, e CVS, ma (secondo i suoi autori perlomeno), è molto più semplice di ognuno di questi sistemi."

ArX — http://www.nongnu.org/arx/

ArX è un sistema di controllo di versione che offre funzionalità di branche e merge, verifica crittografica dell'integrità dei dati, e la possibilità di pubblicare gli archivi facilmente su ogni server HTTP.

SourceJammer — http://sourcejammer.org/

"SourceJammer sistema di controllo del codice e versionamento scritto in Java. Consiste in una componente lato server che mantiene i file e la storia delle versioni, e tratta i check-in e i check-out, eccetera e altri comandi; e una componente lato client che fa le richieste al servere e gestisce i file sul lato client del file system."

FastCST — http://www.zedshaw.com/projects/fastcst/index.html

"Un sistema 'moderno' che usa insiemi di modifiche invece delle revisioni dei file e operazioni distribuite piuttosto che controllo centralizzato. Finchè avete un account di mail potete usare FastCST. Per grandi distribuzioni avete solo bisogno di un server FTP e/o un server HTTP o usare il comando precostruito 'serve' per fornire direttamente il vostro materiale. Tutti gli insiemi di modifiche sono universalmente unici e hanno tonnellate di meta-data così potete rifiutare tutto ciò che non volete prima di provarlo. Il merge è fatto confrontando l'insieme di cambiamenti con i contenuti della directory corrente, piuttosto di provare a farne il merge con altro insieme di cambiamenti."

Superversion — http://www.superversion.org/

"Superversion è un sistema di controllo di versione distritibuito multi-utente basato su insiemi di modifiche. Vuole essere un'alternativa di robustezza industriale, open source, alle soluzioni commerciali che sia facile da usare (o anche più facile) e similmente potente. Infatti, una usabilità intuitiva ed efficiente è stata una delle principali priorità nello sviluppo di Superversion fin all'inizio."

Appendice B. Bug Tracker Liberi

Non importa quale bug tracker il progetto usi, ad alcuni sviluppatori piace sempre lamentarsene. Questo sembra essere più vero nei bug tracker che in ogni altro strumento di sviluppo standard. Pensa che sia perchè i bug tracker sono così visuali e così interattivi che è facile immaginare i miglioramenti che uno farebbe (se solo ne avesse il tempo), e di descrivere ad alta voce i miglioramenti. Prendete le inevitabili lamentele cum grano salis—molti dei bug tracker in seguito sono abbastanza buoni.

Lungo questo elenco, la parola "problema" [orig. "issue", ndt] è usata per riferirsi a cosa trova il tracker. Ma ricordate che ogni sistema può avere la sua peculiare terminologia. in cui il corrispondente termine potrebbe essere "artefatto"["artifact"] o "baco" ["bug"] o qualcos'altro.

Bugzilla — http://www.bugzilla.org/

Bugzilla è molto conosciuto, attivamente mantenuto, e sembra rendere gli utenti abbastanza felici. Ne ho usato una variante modificata nel mio lavoro per quattro anni ormai, e mi piace. Non è altamente personalizzabile, ma in un modo strano, che può essere una delle sue funzionalità: le installazioni di Bugzilla tendono a sembrare abbanstanza simili ovunque le si trovi, il che significa che molti sviluppatori sono già abituati alla sua interfaccia e si sentiranno di essere in territorio familiare.

GNATS — http://www.gnu.org/software/gnats/

GNU GNATS è uno dei più vecchi bug tracker open source, ed è ampiamente usato. I suoi punti di forza sono la diversità di interfaccia (può essere usato non solo attraverso un browser, ma anche in strumenti email e linea di comando), e memorizzazione dei problemi come plaintext. Il fatto che tutti i dati dei problemi sono memorizzati in file di testo su disk rende più facile scrivere strumenti personalizzati per pescare e gestire i dati (per esempio, per generare rapporti statistici). GNATS può anche assorbire le email in modo automatico in vari modi, e può aggiungerle ai problemi appropriati basandosi su pattern nell'intestazione della email, che rende la memorizzazione delle conversazioni utente/sviluppatore molto facile.

RequestTracker (RT) — http://www.bestpractical.com/rt/

Il sito web di RT dice "RT è un sistema di segnalazione di livello enterprise che permette ad un gruppo di persone di gestire compiti, problemi, e richieste sottomesse da una comunità di utenti in modo intelligente ed efficiente," e questo riassume abbastanza. RT ha una interfaccia web abbastanza pulita, e sembra avere un base di installazione abbastanza vasta. L'interfaccia è un po' troppo visualmente complessa, ma diventa meno distraente quando vi ci abituate. RT ha licenza GNU GPL (per qualche ragione, il loro sito web non lo dice chiaramente).

Trac — http://trac.edgewall.com/

Trac è un po' di più di un bug tracker: è veramente un sistema con wiki integrata e bug tracker. Usa i collegamenti tra wiki per mettere in relazione problemi, file, insiemi di modifiche di controllo di versione, e pagine wiki normali. E' abbastanza semplici da tirare su, e si integra con Subversion (vedi Appendice A, *Sistemi di Controllo di Versione Liberi*).

Roundup — http://roundup.sourceforge.net/

Roundup è molto facile da installare (è richiesto solo Python 2.1 o superiore) e facile da usare. Ha interfacce web, email e linea di comando. I template dei dati dei problemi e l'interfaccia web sono personalizzabili, così come una parte della sua logica di transizione di stati.

Mantis — http://www.mantisbt.org/

Mantis è un sistema di bug tracker basato su web, scritto in PHP, che usa il database MySQL per la memorizzazione. Ha le funzionalità che vi aspettereste. Personalmente, trovo l'interfaccia web pulita, intuitiva, e gradevole agli occhi.

Flyspray — http://www.flyspray.org/

Flyspray è un sistema di bug tracker basato su web scritto in PHP. Le sue pagine web lo descrivono come "non complicato", e la lista di funzionalità include: supporto a diversi database (al momento (MySQL e PGSQL); progetti multipli; attività di monitoraggio, con notifica di cambiamenti (via email o Jabber); storia completa delle attività; temi CSS, allegati ai file, funzionalità avanzate di ricerca (anche facili da usare); feed RSS/Atom; wiki o semplice testo come input; votazioni; grafi di dipendenza.

Scarab — http://scarab.tigris.org/

Scarab è concepito per essere un bug tracker altamente personalizzabile, completamente accessoriato, che offre più o meno la somma delle funzionalità offerte dagli altri bug tracker: acquisizione dati, interrogazioni, rapporti, notifiche alle parti interessate, accumulo collaborativo dei commenti, e tracciamento delle dipendenze.

E' personalizzabile attraverso le pagine web di amministrazione. Potete avere "moduli" (progetti) multipli attivi in una singola installazione di Scarab. All'interno di un dato modulo, potete creare nuovi tipi di problemi (difetti, soluzioni, attività, richieste di supporto, eccetera), e aggiungere attributi arbitrari, per regolare il tracker alle specifiche necessità del vostro progetto.

Nel tardo 2004, Scarab stava arrivando vicino alla sua release 1.0.

Debian Bug Tracking System (DBTS) — http://www.chiark.greenend.org.uk/~ian/debbugs/

Il sistema di tracciamento di bug di Debian è inusuale nel fatto che tutti gli input e le manipolazioni dei problemi sono fatti via email: ogni problema ha il suo indirizzo email dedicato. Il DBTS scala molto bene: http://bugs.debian.org/ ha 277,741 problemi, per esempio.

Dato che l'interazione avviene attraverso normali client email, un ambiente che è familiare e facilmente accessibile alla maggior parte della gente, il DBTS va bene per trattare grandi volumi di segnalazioni in arrivo che hanno bisogno di una veloce classificazione e risposta. Ci sono anche degli svantaggi,certo. Gli sviluppatori devono investire il tempo necessario ad imparare il sistema di comandi email, e gli utenti devono scrivere le loro segnalazioni di bug senza un modello web a guidarli nella scelta dell'informazione da scrivere. Sono disponibili strumenti per aiutare gli utenti a mandare segnalazioni di bug migliori, come il programma da riga di comando **reportbug** o il pacchetto per Emacs `debbugs-el`. Ma la maggior parte delle persone non useranno questi strumenti; semplicemente scriveranno le email a mano, e potranno seguire o meno le linee guida per la segnalazione di bug pubblicate dal vostro progetto.

The DBTS ha una interfaccia web di sola lettura, per vedere e interrogare i problemi.

Tracker di problemi e segnalazioni

Questi sono più orientati verso il tracciamento di segnalazioni di help desk piuttosto che al tracciamento di bug nel software. Probabilmente farete di meglio con un normale bug tracker, ma questi sono elencati

per completezza, e perchè ci potrebbero essere progetti inusuali per cui un sistema di tracciamento di problemi e segnalazioni potrebbe essere più adeguato che un bug tracker tradizionale.

- **WebCall** — http://myrapid.com/webcall/
- **Teacup** — http://www.altara.org/teacup.html

  (Teacup non sembra essere più sotto attivo svilupo, ma i download sono tuttora disponibili. Da notare che ha interfaccia sia web che email.)

Bluetail Ticket Tracker (BTT) — http://btt.sourceforge.net/

BTT è qualcosa tra un tracker standard di problemi e un bug tracker. Offre funzionalità di privacy che sono qualcosa di inusuale tra i bug tracker open source: gli utenti del sistema sono classificati come Staff, Amico, Cliente, Anonimo, e più o meno dati sono disponibili a seconda della propria categoria. Offre un po' di integrazione email, una interfaccia a riga di comando, e meccanismi per convertire email in segnalazioni. Ha anche funzionalità per mantenere informazione non associata con nessuna segnalazione specifica, come la documentazione interna o le FAQ [Frequently Asked Questions, domande poste di frequente; ndt].

Appendice C. Perchè dovrebbe importarmi di che colore sia la rastrelliera?

Non dovrebbe; davvero non importa, e avete cose migliori su cui spendere il vostro tempo.

Il famoso messaggio "della rastrelliera" di Poul-Henning Kamp (di cui appare un estratto in Capitolo 6, *Comunicazione*) è una eloquente disquisizione su cosa tende ad andare male nelle discussioni di gruppo E' riportato qui con il suo permesso. La URL originale è http://www.freebsd.org/cgi/getmsg.cgi?fetch=506636+517178+/usr/local/www/db/text/1999/freebsd-hackers/19991003.freebsd-hackers.

```
Subject: Una rastrelliera per bici (ogni colore andrà bene) su di un erba più verd
From: Poul-Henning Kamp <phk@freebsd.org>
Date: Sat, 02 Oct 1999 16:14:10 +0200
Message-ID: <18238.938873650@critter.freebsd.dk>
Sender: phk@critter.freebsd.dk
Bcc: Blind Distribution List: ;
MIME-Version: 1.0


[bcc'ed to committers, hackers]

Il mio ultimo pamphlet era stato sufficientemente ben accolto che non ero preoccup
e oggi ho il tempo e la predisposizione a farlo.

Ho avuto un piccolo problema nel decidere la giusta distribuzione di questo tipo d
conoscenza nascosta ai contributori e agli hacker, che è probabilmente il meglio c
iscritto agli hacker ma a dopo i dettagli.

La cosa che mi ha scatenato questa volta è il thread "sleep(1) dovrebbe durare fra
le nostre vite per molti giorni ora,  probabilmente già alcune settimane, non poss

Per coloro che hanno perso questo particolare thread: Congratulazioni.

Era una proposta di rendere sleep(1) DTRT (Do The Right Thing, fai la cosa giusta)
un argomento non di tipo intero che spegnesse questo particolare fuoco di paglia.
nient'altro di questo, perchè è un argomento molto più piccolo di quanto ci si asp
del thread, e ha già ricevuto molta più attenzione di alcuni dei *problemi* che ab

La saga del sleep(1) è il più splendente esempio di una discussione della rastrell
mai avuto in FreeBSD. La proposta era ben pensata, avremmo guadagnato compatibilit
e ancora essere completamente compatibili con ogni programma che qualcuno abbia ma

Comunque sono state create e lanciate così tante obiezioni, proposte e cambiamenti
il cambiamento avrebbe chiuso tutti i buchi del formaggio svizzero o cambiato il g
o qualcosa di altrettanto serio.

"Cosa c'entra questa rastrelliera da bici?" Qualcuno di voi me lo ha chiesto.
```

E' una lunga storia, o meglio è una vecchia storia, ma in realtà
è abbastanza breve. C. Northcote Parkinson scrisse un libro nei primi anni 60, chi
"Parkinson's Law" ("Legge di Parkinson"), che contiene molti aspetti delle dinamic
gestione.

Potete trovarlo su Amazon, e magari anche nelle librerie dei vostri genitori,
vale il suo prezzo e il tempo di leggerlo in entrambi i casi, se vi piace Dilbert,

Qualcuno di recente mi ha detto che lo ha letto e ha trovato che solo circa il 50%
questi tempi. Questo è dannatamente buono direi, molti dei moderni libri di gestio
molto più bassi di così, e questo è vecchio di più di 35 anni.

Nello specifico esempio che coinvolge la rastrelliera delle biciclette,
l'altro componente vitale è una centrale atomica, penso che questo illustri l'età

Parkinson mostra come puoi andare nell'ufficio del direttore e ottenere
l'approvazione per costruire una centrale atomica da milioni o persino miliardi di
ma se volete costruire una rastrelliera per le biciclette sarete bloccati in discu
senza fine.

Parkinson spiega che questo accade perchè una centrale atomica è così vasta,
così costosa e così complicata che la gente non può percepirla, e piuttosto che pr
ricadono nell'assunzione che qualcun altro abbia controllato tutti i dettagli prim
così avanti. Richard P. Feynmann da alcuni esempi interessanti e molto pertinent
Los Alamos nei suoi libri.

Dall'altro lato una rastrelliera per bici. Chiunque può costruirne una in un fine
e ancora avere il tempo di guardare la partita in TV. Quindi non importa quanto be
ragionevole con la vostra proposta, qualcuno coglierà la possibilità di mostrare c
lavoro, che sta prestando attenzione, che è *qui*.

In Danimarca lo chiamiamo "lasciare l'impronta". Riguarda l'orgoglio personale e i
si tratta di essere in grado di indicare da qualche parte e dire "Qui! *io* l'ho f
E' un importante tratto nei politici, ma presente in molta gente se viene data l'o
nel cemento fresco.

Piego la testa in rispetto al proponitore originale perchè è rimasto attaccato all
pulizia delle noccioline dal tappeto, e il cambiamento ora è nel nostro albero. Av
andato via dopo meno di una manciata di messaggi in quel thread.

E questo mi porta, come promesso prima, al perchè mi sono tolto da -hackers:

Mi sono tolto da -hackers molti anni fa, perchè non potevo stare dietro al carico
altre mailing list per lo stesso identico motivo.

E ancora ricevo tante email. Molte vengono instradate su /dev/null dai filtri: Gen
così come commit a documenti in lingue che non capisco, commit su porte come quest
lo sappia.

Ma nonostante questi denti affilati sotto la casella di posta, ancora ricevo tropp

Questo è dove l'erba più verde entra nell'immagine:

Spero che potremmo ridurre la quantità di rumore nelle nostre mailing list e spero

che la gente costruisca rastrelliere per biciclette così spesso, e non mi interess

Il primo di questi desideri riguarda l'essere civili, sensibili e intelligenti nel

Se potessi definire in maniera concisa e precisa un insieme di criteri per quando
ad una email così che chiunque sarebbe d'accordo e si fermasse a quello, sarei un

Ma fatemi suggerire alcune finestre pop-up che mi piacerebbe vedere implementate n
invia o risponde ad email di mailing list a cui vorrebbero che mi iscrivessi:

```
+-------------------------------------------------------------+
| La tua email sta per essere mandata a diverse migliaia di   |
| persone, che dovranno usare almeno 10 secondi leggendola    |
| prima di decidere se è interessante. Almeno due             |
| settimane-uomo saranno usate nel leggere la tua email.      |
| Molti dei destinatari dovranno pagare per scaricare la      |
| tua email.                                                  |
|                                                             |
| Sei assolutamente certo che la tua email è di sufficiente   |
|  importanza per disturbare tutte queste persone?            |
|                                                             |
|                  [SI]  [RIVEDI]  [CANCELLA]                 |
+-------------------------------------------------------------+

+-------------------------------------------------------------+
| Attenzione: Non hai ancora letto tutte le email di questo   |
| thread.Qualcun altro potrebbe già aver detto cosa stai per  |
|  dire nella tua risposta. Per favore leggi l'intero thread  |
|  prima di rispondere ad una email ivi contenuta.            |
|                                                             |
|                      [CANCELLA]                             |
+-------------------------------------------------------------+

+-------------------------------------------------------------+
| Attenzione: Il tuo programma di posta non ti ha nemmeno     |
| ancora mostrato l'intero messaggio. Ne segue logicamente    |
| che non hai potuto leggerlo e capirolo tutto.               |
|                                                             |
| Non è educato rispondere ad una email fino a quando l'hai   |
| letta tutta e ci hai pensato.                               |
|                                                             |
| Un timer di raffreddamento ti impedirà di rispondere a      |
| tutte le email in questo thread per la prossima ora.        |
|                                                             |
|                       [Cancella]                            |
+-------------------------------------------------------------+

+-------------------------------------------------------------+
| Hai scritto questa email ad un tasso di più di N.NN cps.    |
| (caratteri per secondo, ndt)                                |
| Non è in generale possibile pensare e scrivere ad un tasso  |
| maggiore di A.AA cps, e quindi è possibile che la tua       |
| risposta sia incoerente, mal pensata e/o emotiva.           |
|                                                             |
```

```
      | Un timer di raffreddamento ti impedirà di mandare ogni     |
      | email per la prossima ora.                                 |
      |                                                            |
      |                       [Cancella]                           |
      +------------------------------------------------------------+
```

La seconda parte del mio desiderio è più emotiva. Ovviamente, le capacità che
abbiamo avuto nell'alimentare il fuoco non amichevole nel thread sleep(1), nonosta
i loro numerosi anni nel progetto, non si preoccuparono mai abbastanza per fare qu
farsi scaldare così tanto da qualcun altro che lo facesse?

Vorrei saperlo.

So che il ragionamento non avrà il potere di fermare questo "conservativismo reazi
Potrebbe essere che queste persone siano frustrate dalla loro tangibile mancanza d
o potrebbe essere un brutto caso di "siamo vecchi e scontrosi, NOI sappiamo come i

In entrambi i casi è molto improduttivo per il progetto, ma non ho suggerimenti su
che possa suggerire è di evitare di nutrire i mostri in agguato nella mailing list
dimenticate siano lì.

Spero che possiamo avere una base più forte e larga di contributori in FreeBSD, e
possiamo evitare che i vecchi uomini scontrosi e gli [omissis] del mondo li sbrani
prima che possano appoggiare un piede a terra.

Per la gente che spiava qua fuori, intimorita a partecipare dai gargoule: posso so
di provare comunque, non è questo il modo in cui vorrei che sia l'ambiente del pro

Poul-Henning

Appendice D. Istruzioni di Esempio per Segnalare un Bug

Questa è una versione alleggerita delle istruzioni online del progetto Subversion per i nuovi utenti su come segnalare bug. Vedi sezione chiamata «Trattate Ogni Utilizzatore Come un Potenziale Volontario» in Capitolo 8, *Gestire i Volontari* sul perchè è importante che un progetto abbia tali istruzioni. Il documento originale è qui http://svn.collab.net/repos/svn/trunk/www/bugs.html.

```
                        Segnalare Bug in Subversion

Questo documento dici come e dove segnalare i bug. (Non è na lista dei bug mportan

Dove Segnalare un Bug
---------------------

    * Se il bug è di Subversion stesso, manda una mail a
      users@subversion.tigris.org. Una volta che è stato confermato come bug,
      qualcuno, possibilmente tu, può entrare nel tracciatore dei problemi (O
      se sei abbastanza sicuro del bug, vai avanti e scrivi direttamente alla mail
      list di sviluppo, dev@subversion.tigris.org. Ma se non sei sicuro, è meglio
      a users@ ; lì qualcuno può dirti se il comportamento che hai riscontrato è a

    * Se il bug è nelle libreria APR library, per favore segnalalo ad entrambe que
      mailing lists: dev@apr.apache.org, dev@subversion.tigris.org.

    * Se il bug è nella libreria Neon HTTP, per favore segnalalo a:
      neon@webdav.org, dev@subversion.tigris.org.

    * Se il bug è in Apache HTTPD 2.0, per favore segnalalo ad entrambe queste
      mailing lists: dev@httpd.apache.org,
      dev@subversion.tigris.org. La mailing list di sviluppo di Apache httpd è mol
      quindi il tuo messaggio di segnalazione di bug può essere sovrastato. Puoi a
      http://httpd.apache.org/bug_report.html.

    * Se il bug è nel tuo orticello, per favore dagli un abbraccio e tienilo stret

Come Segnalare un Bug
---------------------

Primo, verificate che sia un bug. Se Subversion non si comporta nel modo atteso, g
e negli archivi di mailing list per una prova che si debba comportare nel modo che
di buon senso, tipo Subversion ha appena distrutto i tuoi dati e fatto uscire del
fidarti del tuo giudizio. Ma se non sei sicuro, vai avanti e chiedi agli utenti de
 users@subversion.tigris.org, o chiedi in IRC, irc.freenode.net, channel #svn.

Una volta che hai stabilito che è un bug, la cosa più importante che
puoi fare è tirare fuori una semplice descrizione e istruzioni di riproduzione. Pe
avevi trovato all'inizio, coinvolge cinque file su dieci commit, cerca di farlo su
un commit. Più facili le istruzioni di riproduzione, più facilmente uno sviluppato
```

Quando scrivi le istruzioni di riproduzione, non scrivere solo una descrizione in
il bug. Dai invece una trascrizione letterale dell'esatta serie di comandi che hai
dei file, e anche il loro contenuto se pensi che possa essere rilevante. La cosa v
la ricetta di riproduzione come uno script, il che ci aiuta molto.

*Veloce test di sanità mentale: *stai* usando la più recente versione di SUbversion
Magari il bug è già stato messo a posto; dovresti testare di nuovo la tua ricetta
versione più recente di Subversion.*

Oltre alla ricetta di riproduzione, avremo anche bisogno di una completa descrizio
dell'ambiente in cui hai riprodotto il bug. Questo significa:

* il tuo sistema operativo
* il rilascio e/o la revisione di Subversion
* il compilatore e le opzioni di configurazione con cui hai compilato Subversi
* ogni modifica privata che hai fatto al tuo Subversion
* la versione di Berkeley DB con cui stai facendo andare Subversion, se c'è
* ogni altra cosa che potrebbe essere rilevante. Sbaglia nel dare troppe infor

Una volta che hai tutto questo, sei pronto a scrivere il rapporto. Inizia con una
Cioè, di come di aspettavi che Subversion si comportasse, e confrontalo con come s
il bug ti sembra ovvio, potrebbe non essere così ovvio a qualcun altro. così è meg
A questo fai seguire la descrizione dell'ambiente e la ricetta di riproduzione. Se
e persino la patch per mettere a posto il bug, va benissimo - vedi
http://subversion.apache.org/docs/community-guide/#patches per le istruzioni su co

Manda tutte queste informazioni a dev@subversion.tigris.org, o se sei già stato lì
di segnalare un problema, allora vai al Issue Tracker e lì segui le istruzioni.

Grazie. Sappiamo che è un gran lavoro fare una segnalazione di bug efficace, ma un
segnalazione può far risparmiare ore del tempo di uno sviluppatore, e rendere più
che il bug venga messo a posto.

Appendice E. Copyright

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

   b. "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

   c. "Creative Commons Compatible License" means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

   d. "Distribute" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through

sale or other transfer of ownership.

e. "License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

f. "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

g. "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

h. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

i. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

j. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them;

to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

k. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights.

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant.

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties

through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative

Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or

endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted

here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights

are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at http://creativecommons.org/.