Free Software Project Management HOWTO

Benjamin "Mako" Hill

mako@debian.org

Diario delle Revisioni

Revisione v0.3.2	15 aprile 2002	Revisionato da: bch
Revisione v0.3.1	18 giugno 2001	Revisionato da: bch
Revisione v0.3	5 maggio 2001	Revisionato da: bch
Revisione v0.2.1	10 aprile 2001	Revisionato da: bch
Revisione v0.2	8 aprile 2001	Revisionato da: bch
Revisione v0.01	27 marzo 2001	Revisionato da: bch

Prima versione

Questo HOWTO è diretto a persone che hanno esperienza di programmazione, e qualche capacità nella gestione di un progetto software, ma che sono nuove all'esperienza del software libero. Questo documento vuole fungere da guida agli aspetti non-tecnici della gestione dei progetti di software libero, ed è stato scritto come corso accelerato per quelle capacità relazionali che non vengono insegnate ai programmatori commerciali, ma possono determinare il successo o il fallimento di un progetto di software libero. Traduzione a cura di Andrea Giancola. Revisione di Giulio Daprelà ed Elisabetta Galli.

1. Introduzione

Basta scorrere freshmeat.net per ricavare una montagna di ragioni per l'esistenza di questo HOWTO: Internet è disseminata di programmi utili e scritti in modo eccellente, che però si sono dissolti nell'universo dell'oblio del software libero. Questo triste scenario mi ha portato a chiedermi: "Perché?"

Il presente HOWTO cerca di fare molte cose (probabilmente troppe), ma non può rispondere a questa domanda, e non ci proverà. Ciò che l' HOWTO cercherà di fare è fornire al proprio progetto di software libero una chance da giocarsi, un vantaggio. Se si scrive una porcheria che non interessa a nessuno, si può leggere questo HOWTO fino a recitarlo a memoria nel sonno, e nonostante questo il progetto probabilmente fallirà. Non solo, si può scrivere un software bello, utile, e seguire tutte le istruzioni di questo HOWTO e il proprio software ancora potrebbe non farcela. Certe volte la vita è così. Comunque, mi sbilancerò sino a dire che se si scrive un bellissimo ed utile software ed si ignorano i consigli di questo HOWTO, probabilmente si fallirà *più spesso*.

Molte delle informazioni di questo HOWTO sono questioni di buon senso; naturalmente, come ogni dibattito sulle interfacce può provare, ciò che è buon senso per alcuni programmatori si dimostra completamente controintuitivo per altri. Dopo aver spiegato frammenti di questo HOWTO a sviluppatori di software libero in diverse occasioni, ho compreso che scrivere questo HOWTO potrebbe fornire una risorsa utile ed un punto d'incontro affinché i programmatori si scambino idee su ciò che ha o non ha funzionato per loro.

È necessaria una breve introduzione alla questione licenze: ne è cosciente chi si è trovato coinvolto in quella che sembra un'infinita lotta al diritto di proprietà intellettuale.

1.1. Copyright Information

1.2. Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your project (and potentially your system). Proceed with caution, and although this is highly unlikely, the author(s) does not take any responsibility for that.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

1.3. Nuove versioni

Questa versione è parte del terzo ciclo di pre-rilascio di questo HOWTO. È stato scritto per essere rilasciato agli sviluppatori per ricevere critiche e nuove idee. Si ricordi per favore che questa versione dell'HOWTO è ancora in uno stadio infantile e continuerà ad essere ampiamente revisionata.

L'ultimo numero di versione di questo documento dovrebbe sempre essere riportato sulla homepage del progetto (http://yukidoke.org/~mako/projects/howto) residente su yukidoke.org. (http://yukidoke.org)

La versione più recente di questo HOWTO sarà sempre resa disponibile sullo stesso sito web, in una varietà di formati:

- HTML (http://yukidoke.org/~mako/projects/howto/FreeSoftwareProjectManagement-HOWTO/t1.html).
- HTML (pagina singola) (http://yukidoke.org/~mako/projects/howto/FreeSoftwareProjectManagement-HOWTO.html).
- testo semplice (http://yukidoke.org/~mako/projects/howto/FreeSoftwareProjectManagement-HOWTO.txt).
- postscript compresso (http://yukidoke.org/~mako/projects/howto/FreeSoftwareProjectManagement-HOWTO.ps.gz).
- sorgente SGML compresso (http://yukidoke.org/~mako/projects/howto/FreeSoftwareProjectManagement-HOWTO.sgml.gz).

1.4. Riconoscimenti

In questa versione ho il piacere di ringraziare:

I colleghi sviluppatori Debian Martin Michlmayr e Vivek Venugopalan, che hanno inviato informazioni e collegamenti ad articoli estremamente interessanti. Entrambi sono stati aggiunti alla bibliografia, e sono state aggiunte all'HOWTO informazioni ricevute da ognuno. Grazie ad Andrew Shugg che ha segnalato diversi errori nel documento. Un grande grazie anche a Sung Wook Her (alias RedBaron) che sta facendo la prima traduzione dell'HOWTO in coreano. Sono stato felice di notare che la gente fino ad ora ha tratto giovamento e beneficio da questo HOWTO.

Ringraziamenti meno recenti, ma che non voglio eliminare, includono: Josh Crawford, Andy King e Jaime Davila, che hanno letto questo documento per intero e hanno dato dei feedback molto utili per correggerlo e migliorarlo. Non posso ringraziarvi abbastanza per il vostro aiuto. Un "grazie" extra va ad Andy King, che ha riletto il documento diverse volte, ed ha inviato delle correzioni che mi hanno reso la vita più facile.

Karl Fogel, l'autore di *Open Source Development with CVS*, pubblicato dalla Coriolis Open Press. Lunghi estratti del suo libro sono disponibili sul web (http://cvsbook.red-bean.com). 225 pagine del libro sono disponibili sotto licenza GPL, e costituiscono il miglior manuale introduttivo per CVS che abbia mai visto. Il resto del libro copre "le sfide e gli innati problemi di principio connessi alla conduzione di un progetto Open Source usando CVS." Il libro fa un buon lavoro nel coprire alcuni degli argomenti trattati in questo HOWTO, e molti altri. Il sito web del libro (http://cvsbook.red-bean.com) offre informazioni su come ordinare il libro, e fornisce diverse traduzioni dei capitoli su CVS. Se si è seriamente interessati alla conduzione di un progetto di software libero bisogna avere questo libro. Ho cercato di citare Fogel nelle sezioni di questo HOWTO in cui ero consapevole di stare attingendo direttamente dalle sue idee; se ne ho dimenticata qualcuna, chiedo venia, cercherò di correggerle nei rilasci futuri.

Karl Fogel può essere contattato presso `<kfogel (at) red-bean (dot) com>`

Anche Eric S. Raymond ha fornito materiale di supporto e ispirazione per questo HOWTO, con le sue discussioni prolifiche, coerenti e meticolose, e Lawrence Lessig mi ha ricordato l'importanza del software libero. Inoltre, voglio ringraziare tutti gli utenti e sviluppatori coinvolti nel progetto Debian (http://www.debian.org). Debian mi ha fornito una casa, un posto dove praticare il patrocinio del software libero, un posto dove ottenere dei risultati, un posto dove imparare da quelli che sono parte del movimento da molto più tempo di me. Debian mi ha anche fornito la testimonianza di un progetto di software libero che funziona davvero bene.

Soprattutto, voglio ringraziare *Richard Stallman*, per il suo lavoro alla Free Software Foundation, e per non essersi mai arreso. Stallman ha stabilito e articolato le basi filosofiche che mi spingono verso il software libero, e che mi portano a scrivere un documento per far sì che esso abbia successo. RMS può essere sempre raggiunto via email a <rms (at) gnu (dot) org>.

1.5. Il feedback

Ogni feedback su questo documento è sempre, senza fallo, il benvenuto. Senza le segnalazioni ed aggiunte dei lettori, questo documento non esisterebbe. Pensate che manchi qualcosa? Non esitate a contattarmi per farmi scrivere un capitolo, sezione o sottosezione, o per scriverne una voi stessi. Voglio che questo documento sia un prodotto di quel processo di sviluppo del software libero che mira a promuovere, e penso che il suo successo finale si fondi sulla sua capacità di esserlo davvero. Per favore inviate le vostre aggiunte, commenti e critiche al seguente indirizzo email: <mako@debian.org>.

1.6. Traduzioni

Non tutti parlano inglese. Le traduzioni sono una buona cosa, e sarebbe molto bello che questo HOWTO raggiungesse quella portata internazionale che la traduzione di un documento comporta.

Sono stato contattato da un lettore che promette una traduzione in coreano. Tuttavia, questo HOWTO è ancora giovane, ed oltre alla promessa del coreano, solamente l'inglese è disponibile; se voleste contribuire a, o eseguire per intero, una traduzione, otterreste il mio completo rispetto e ammirazione, e diventereste parte di un processo molto interessante. Se siete interessati, per favore non esitate a contattarmi a: `<mako@debian.org>`.

2. Iniziare un progetto

Senza dubbio l'inizio è il periodo più difficile per gestire con successo un progetto di software libero: la posa di fondamenta solide determinerà se il progetto prospererà o avvizzirà fino a morire. È anche l'argomento di più immediato interesse per chiunque legga questo documento come una guida.

Iniziare un progetto implica un dilemma che come programmatori bisogna affrontare: nessun potenziale utente del programma è interessato ad un programma che non funziona, ma d'altra parte il processo di sviluppo che si vuole impiegare ha come imperativo il coinvolgimento degli utenti.

È in questi pericolosi momenti iniziali che chi si sta dando da fare per iniziare un progetto di software libero deve cercare un equilibrio tra queste esigenze. Uno dei modi più importanti per farlo è stabilire una struttura solida per il processo di sviluppo attraverso alcuni dei suggerimenti delineati in questa sezione.

2.1. Scegliere un progetto

Se si sta leggendo questo documento, ci sono buone probabilità che si abbia già in mente un'idea per un progetto. Ci sono anche discrete probabilità che questa idea possa soddisfare una lacuna percepita anche da altri, facendo qualcosa che nessun altro progetto di software libero fa, o facendolo in un modo sufficientemente peculiare da necessitare di un nuovo software.

2.1.1. Identificare ed articolare l'idea

Eric S. Raymond scrive di come nascono i progetti software nel suo nuovo saggio, " The Cathedral and the Bazaar" (http://www.tuxedo.org/~esr/writings/cathedral-bazaar/), che è quasi una lettura obbligatoria per ogni sviluppatore di software libero. È disponibile online.

In "The Cathedral and the Bazaar" Raymond dice che: "ogni buon lavoro software comincia grattando il prurito di uno sviluppatore." L'ipotesi di Raymond, ora largamente accettata, è che i nuovi programmi di software libero siano scritti prima di tutto per risolvere un problema specifico presentatosi allo sviluppatore.

Se si ha in mente un'idea per un programma, ci sono buone probabilità che affronti un problema, o "prurito", che ci si vuole grattare: *questa idea è il progetto*. Esprimerla con chiarezza, scriverla, descrivere in dettaglio il problema che si vuole affrontare: il successo del progetto nel trattare un particolare problema sarà legato all'abilità nell'identificare quel problema con chiarezza fin dall'inizio. Si scopra cosa si vuole che il proprio progetto faccia esattamente.

Monty Manley esprime l'importanza di questo passo iniziale in un saggio, "Managing Projects the Open Source Way. (http://news.linuxprogramming.com/news_story.php3?ltsn=2000-10-31-001-05-CD)" Come mostrerà la prossima sezione, c'è *molto* lavoro da fare prima ancora che il software sia pronto per essere programmato. Manley dice: "Iniziare come si deve un progetto Open Source significa che uno sviluppatore deve prima di tutto evitare di scrivere codice troppo presto!"

2.1.2. Valutare l'idea

Nel valutare l'idea, bisogna per prima cosa porsi alcune domande. Questo dovrebbe avvenire prima di avanzare ulteriormente nella lettura del presente HOWTO. Ci si chieda: *il modello di sviluppo del software libero è davvero quello giusto per il proprio progetto?*

Ovviamente, dal momento che il programma gratta un proprio prurito, si è certamente interessati a vederlo implementato in codice. Ma, poiché un singolo hacker che programma in solitudine non si può qualificare come uno sforzo di sviluppo di software open source, ci si deve porre una seconda domanda: *qualcun altro potrebbe essere interessato?*

Certe volte la risposta è un semplice "no". Se si vuole scrivere un insieme di script per ordinare la *propria* raccolta di MP3 sulla *propria* macchina, *forse* il modello di sviluppo del software libero non è quello giusto. D'altro canto, se si vuole scrivere un insieme di script per ordinare gli MP3 di *chiunque*, un progetto di software libero potrebbe utilmente riempire un vuoto.

Fortunatamente Internet è un luogo così grande e variegato che è possibile che qualcuno, da qualche parte, condivida i propri interessi e provi lo stesso "prurito". E il fatto che ci siano così tante persone con bisogni e desideri così simili tra loro introduce la terza domanda principale: *qualcuno ha già avuto la propria idea, o un'idea abbastanza simile?*

2.1.2.1. Trovare progetti simili

Ci sono posti dove si può andare, sul web, per cercare di rispondere alla domanda di cui sopra. Se si ha esperienza con la comunità del software libero probabilmente si ha già familiarità con molti di questi siti. Tutte le risorse elencate sotto consentono di effettuare ricerche nei propri database:

freshmeat.net

> freshmeat.net (http://freshmeat.net) si autodefinisce come "il più grande indice sul Web per software Linux e Open Source" e la sua reputazione al riguardo è ineguagliabile e indiscussa. Se non si riesce a trovare qualcosa su freshmeat, è difficile che lo si possa trovare altrove.

Slashdot

Slashdot (http://slashdot.org) fornisce “Notizie per nerd. Roba che conta”, il che di solito comprende discussioni sul software libero, l’open source, la tecnologia, e notizie ed eventi sulla cultura geek. Non è insolito che un progetto di sviluppo particolarmente allettante venga annunciato qui, perciò vale indubbiamente la pena di controllare.

SourceForge

SourceForge (http://sourceforge.net) ospita e promuove un numero crescente di progetti open source e di software libero. Sta anche rapidamente diventando un punto di incontro ed una sosta obbligata per gli sviluppatori di software libero. La sua mappa dei software (http://sourceforge.net/softwaremap/trove_list.php) e le pagine dei nuovi rilasci (http://sourceforge.net/new/) dovrebbero essere soste obbligate prima di imbarcarsi in un nuovo progetto software. SourceForge fornisce anche una biblioteca di frammenti di codice (http://sourceforge.net/snippet/) che contiene utili pezzetti di codice riusabili, in tutta una gamma di linguaggi, che possono tornare utili in qualsiasi progetto.

Google e la Linux Search di Google

Google (http://www.google.com) e la Linux Search di Google (http://www.google.com/linux) offrono potenti strumenti di ricerca del web che possono far scoprire persone che lavorano su progetti simili. Non è un catalogo di software o informazioni come freshmeat o Slashdot, ma vale la pena di controllare, per assicurarsi di non dirigere i propri sforzi verso un progetto ridondante.

2.1.2.2. Decidere di procedere

Una volta che si è mappata con successo la zona delle operazioni, e si ha un’idea di quali progetti di software libero somiglianti esistono, ogni sviluppatore deve decidere se andare avanti con il proprio progetto. È raro che un nuovo progetto cerchi di raggiungere un obiettivo che non è affatto simile o collegato all’obiettivo di un altro progetto. Chiunque cominci un nuovo progetto deve chiedersi: “il nuovo progetto duplicherà il lavoro fatto da un altro progetto? il nuovo progetto si procaccerà sviluppatori ai danni di un progetto esistente? gli obiettivi del nuovo progetto possono essere raggiunti aggiungendo funzionalità ad un progetto esistente?”

Se la risposta ad almeno una di queste domande è “sì”, si provi a contattare lo sviluppatore del progetto (o dei progetti) esistente in questione per capire se lui o lei sarebbe disposto a collaborare.

Per molti sviluppatori questo è l’aspetto più ostico della gestione dei progetti di software libero, ma è un aspetto essenziale: è facile appassionarsi ad un’idea e lasciarsi prendere dall’impeto e dall’eccitazione di un nuovo progetto. Spesso è estremamente difficile da fare, ma è importante che ogni sviluppatore di software libero ricordi che l’interesse della comunità del software libero, e il modo più veloce per raggiungere gli obiettivi del proprio progetto e di progetti simili, spesso può consistere nel *non* dare il via ad un nuovo processo di sviluppo.

2.2. Dare un nome al progetto

Anche se ci sono un sacco di progetti che falliscono pur avendo nomi descrittivi, e un sacco che hanno successo pur senza averli, quando si dà un nome al progetto vale la pena di pensarci un po' sopra. Leslie Orchard affronta questo problema in un articolo pubblicato su Advogato (http://www.advogato.org/article/67.html). L'articolo è breve e sicuramente merita almeno un'occhiata.

Il riassunto è che Orchard raccomanda di scegliere un nome tale che, dopo averlo udito, molti utenti o sviluppatori:

- sapranno cosa fa il progetto
- se ne ricorderanno un domani

Curiosamente il progetto di Orchard, "Iajitsu," non fa nessuna delle due cose. Probabilmente non c'è alcuna correlazione col fatto che lo sviluppo di quel progetto si è arrestato da quando l'articolo è stato scritto.

Comunque l'argomento è convincente. Ci sono aziende il cui solo lavoro consiste nell'inventare nomi per dei software. Tirano su una quantità di denaro *incredibile* facendolo, ed è opinione comune che valgano tutto questo denaro. Anche se probabilmente non ci si può permettere un'azienda come questa, ci si può permettere di trarre un insegnamento dalla loro esistenza, e riflettere un momento sul nome che si sta dando al proprio progetto, perché *conta*.

Se c'è un nome che sta a cuore, ma che non rispetta i criteri di Orchard, si può continuare comunque. Penso che "gnubile" fosse uno dei migliori nomi che abbia mai sentito per un progetto di software libero, e ancora ne parlo, nonostante sia trascorso molto tempo da quando ho smesso di usare il programma. Comunque, se si è flessibili su questo argomento, si ascolti il consiglio di Orchard. Potrebbe aiutare.

2.3. Adottare una licenza per il programma

Ad un certo livello (piuttosto semplicistico), la differenza tra un software libero ed un software proprietario è la licenza. Una licenza aiuta come sviluppatori proteggendo i propri diritti legali ad ottenere la distribuzione del proprio software alle proprie condizioni, ed incoraggia coloro che vorrebbero aiutare il proprio progetto a partecipare.

2.3.1. Scegliere la licenza

Qualsiasi discussione sulle diverse licenze genera sicuramente almeno una piccola flame war, poiché ci sono forti convinzioni che alcune licenze di software libero siano migliori di altre. Questa discussione porta anche alla ribalta la questione del "software open source" e il dibattito sui termini "software open source" e "software libero". Ad ogni buon conto, poiché ho scritto un HOWTO sulla gestione dei progetti di software libero, e non un HOWTO sulla gestione dei progetti software open source, la mia scelta di campo è chiara.

Nel tentativo di raggiungere un diplomatico compromesso, senza sacrificare la mia personale filosofia, consiglio di scegliere una qualsiasi licenza che si conformi alle Debian Free Software Guidelines (http://www.debian.org/social_contract). Scritte originariamente dal progetto Debian sotto la guida di Bruce Perens, le DFSG costituiscono la prima versione della definizione di Open Source (http://www.opensource.org/docs/definition_plain.html). Esempi di licenze libere forniti dalle DFSG sono la GPL, la BSD, e la Licenza Artistica. Come accennato nell'HOWTO[ESRHOWTO] di ESR, se possibile evitare di scrivere una propria licenza. Le tre licenze citate hanno tutte lunghe tradizioni interpretative. Sono anche senza dubbio software libero (e possono quindi essere distribuite come parte di Debian, e in altri posti che consentono lo scambio di software libero).

Conformemente alla definizione di software libero fornita da Richard Stallman in "The Free Software Definition" (http://www.gnu.org/philosophy/free-sw.html), ognuna di queste licenze garantisce "la libertà per gli utenti di eseguire, copiare, distribuire, esaminare, modificare e migliorare il software". Ci sono un sacco di altre licenze che si conformano alle DFSG, ma rifarsi ad una licenza più diffusa offrirà il vantaggio di un immediato riconoscimento e comprensione. Molte persone scrivono tre o quattro frasi in un file COPYING e ritengono di aver scritto una licenza di software libero: come prova la mia lunga esperienza con la mailing list debian-legal molto spesso non è così.

Tentando un'analisi più approfondita, sono d'accordo con Karl Fogel che divide le licenze in due gruppi: le GPL, e quelle diverse dalla GPL.

Personalmente, licenzio tutto il mio software sotto la GPL. Creata e difesa dalla Free Software Foundation e dal GNU Project, la GPL è la licenza usata dal kernel Linux, da GNOME, da Emacs, e dalla gran parte del software GNU/Linux. È la scelta naturale, ed io penso che sia una buona scelta. Ogni fanatico BSD ci tiene a ricordare che la peculiare contagiosità della GPL impedisce la mescolanza di codice GPL con codice non-GPL; secondo molte persone (me compreso) questo è un beneficio, ma secondo alcuni è un grosso svantaggio.

Molte persone scrivono tre o quattro frasi in un file COPYING e ritengono di aver scritto una licenza di software libero: come prova la mia lunga esperienza con la mailing list debian-legal, molto spesso non è così. Può non proteggere l'autore, può non proteggere il software, e può rendere le cose molto difficili per chi voglia usare il software ma presta molta attenzione ai sottili dettagli legali delle licenze. Se ci si tiene molto ad una licenza fatta in casa, la si passi prima a qualcuno all'OSI (http://www.opensource.org) o alla mailing list debian-legal (mailto:debian-devel@lists.debian.org), prima di tutto per proteggersi da imprevisti effetti collaterali.

Le tre licenze più importanti si possono trovare ai seguenti siti:

- La GNU General Public License; (http://www.gnu.org/copyleft/gpl.html)
- la licenza BSD; (http://www.debian.org/misc/bsd.license)
- la licenza Artistica. (http://language.perl.com/misc/Artistic.html)

In ogni caso, si legga ogni licenza prima di usarla per rilasciare il proprio software. Come sviluppatori principali, non ci si possono permettere sorprese sulle licenze.

2.3.2. Il meccanismo delle licenze

Il testo della GPL offre una buona descrizione della meccanica di applicazione di una licenza (http://www.gnu.org/copyleft/gpl.html#SEC4) ad un nuovo software. La mia lista di controllo rapido per l'applicazione di una licenza comprende:

- rendere se stessi o la FSF il detentore del copyright per il lavoro. In qualche raro caso, si potrebbe desiderare che il detentore del copyright sia qualche organizzazione sponsorizzatrice (se è sufficientemente grande e potente). Fare questo è semplice, basta inserirne il nome nello spazio apposito quando si modifica la nota sul copyright riportata sotto. Contrariamente a quanto comunemente si crede, non c'è bisogno di affiliarsi ad alcuna organizzazione: la nota da sola è sufficiente per rivendicare il diritto d'autore sul proprio lavoro;
- se mai fosse possibile, attaccare e distribuire una copia completa della licenza sia con i sorgenti che con i binari, aggiungendo un file a se stante;
- in cima ad ogni file sorgente del proprio programma inserire una nota di copyright, ed includere informazioni su dove può essere reperita la licenza integrale. La GPL raccomanda che ogni file cominci con:

```
una riga per specificare il nome del programma e dare un'idea di cosa fa.
Copyright (C) yyyy  nome dell'autore

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

La GPL prosegue raccomandando di aggiungere indicazioni su come contattare l'autore, via email o posta;

- la GPL continua consigliando, se il programma viene eseguito in modalità interattiva, di far sì che ogni volta che entra in tale modalità stampi una nota che includa un messaggio come il seguente, che indirizza alle informazioni complete sulla licenza del programma:

```
Gnomovision version 69, Copyright (C) anno nome dell'autore
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

- infine, può essere utile includere una "copyright disclaimer" da parte del datore di lavoro o di una scuola, se si lavora come programmatori o se è verosimile che il proprio datore di lavoro, o scuola, possa in futuro accampare diritti di proprietà sul proprio codice. Non ce n'è bisogno frequentemente, ma ci sono molti sviluppatori di software libero che sono finiti nei guai, e vorrebbero avere chiesto una rinuncia di questo genere.

2.3.3. Avvertimento finale sulle licenze

Per favore, per favore, per favore, porre il proprio software sotto il riparo di *qualche* licenza. Può non sembrare importante, e per voi può non esserlo, ma le licenze *sono* importanti. Perché un pacchetto software sia incluso nella distribuzione GNU/Linux Debian, deve avere una licenza che soddisfi le linee guida per il software libero di Debian (http://www.debian.org/social_contract): se il proprio software non ha licenza, non potrà essere distribuito come pacchetto Debian fino a che non sarà ri-rilasciato sotto una licenza libera. Rilasciare la prima versione del proprio software sotto una licenza chiara risparmierà a tutti una seccatura.

2.4. Scegliere un metodo di numerazione delle versioni

La cosa più importante per un sistema di numerazione delle versioni è che ce ne sia uno. Può sembrare pedante enfatizzare questo punto, ma è sorprendente il numero di scripts e piccoli programmi che saltano fuori senza alcun numero di versione.

La seconda cosa più importante per un sistema di numerazione è che i numeri aumentino sempre. I sistemi di tracciamento automatico delle versioni, e la percezione comune di un ordine universale, finirebbero in pezzi se i numeri di versione non crescessero sempre. Non è *molto* importante che 2.1 sia un grande salto e 2.0.005 un piccolo salto, ma è importante che 2.1 sia più recente di 2.0.005.

Seguendo queste due semplici regole non si sbaglierà (troppo). Oltre a questo, la tecnica più comune sembra essere lo schema di numerazione delle versioni "livello principale", "livello secondario", "livello di patch". Che si sia familiari o meno con questo nome, ci si interagisce di continuo. Il primo numero è il numero principale, e indica importanti cambiamenti o riscritture; il secondo numero è il numero secondario, e rappresenta funzionalità aggiunte o ritoccate basate su una struttura in gran parte coerente; il terzo numero è il numero di patch, e di solito si riferisce solo a rilasci che correggono dei bug.

L'uso molto diffuso di questo schema è il motivo per cui si conoscono la natura e il grado relativo delle differenze tra un rilascio 2.4.12 del kernel Linux ed un 2.4.11, 2.2.12, e 1.2.12, pur senza conoscere nulla su nessuno di questi rilasci.

Si possono modificare o ignorare queste regole, e c'è chi lo fa. Ma attenzione, se si sceglie di farlo, qualcuno si arrabbierà, immaginerà che non le conosciate, e tenterà di insegnarvele, probabilmente non molto gentilmente. Io uso sempre questo metodo, e prego anche voi di farlo.

Ci sono diversi sistemi di numerazione delle versioni piuttosto conosciuti, e che potrebbe valere la pena di investigare prima di rilasciare la propria prima versione.

Numerazione delle versioni del kernel di Linux:

> Il kernel di Linux adotta un sistema di numerazione delle versioni in cui ogni numero di versione minore dispari si riferisce ad un rilascio di sviluppo o di collaudo, ed ogni numero di versione minore pari si riferisce ad una versione stabile. Pensandoci per un secondo, con questo sistema, i kernel 2.1 e 2.3 erano e saranno sempre kernel di sviluppo o di collaudo, mentre i kernel 2.0, 2.2 e 2.4 sono tutti codice di produzione, con maggior grado di stabilità e più collaudati.
>
> Sia che si preveda di avere un modello di sviluppo suddiviso (come descritto nel la Sezione 3.3), sia che si preveda di rilasciare una sola versione alla volta, la mia esperienza con diversi progetti di software libero e con il progetto Debian mi ha insegnato che l'uso del sistema di numerazione delle versioni di Linux merita di essere preso in considerazione. In Debian, *tutte* le versioni secondarie sono distribuzioni stabili (2.0, 2.1, etc). Ciononostante, molta gente crede che la versione 2.1 sia instabile o di sviluppo, e continua ad usare una versione più vecchia fino a che è così frustrata dalla mancanza di progressi nello sviluppo che protesta, e capisce il sistema. Se non si rilascia mai un numero di versione secondaria dispari ma solo quelli pari, nessuno viene danneggiato, e meno persone saranno confuse: è un'idea che merita di essere presa in considerazione.

Numerazione delle versioni di Wine:

> A causa della natura insolita dello sviluppo di Wine, per cui il non-emulatore migliora costantemente, ma non si dirige verso un obiettivo immediatamente raggiungibile, Wine è rilasciato ogni tre settimane. Questo viene fatto etichettando i rilasci in un formato "Anno Mese Giorno", in cui ogni rilascio sarà etichettato "wine-XXXXXXXX", cioè la versione del 4 Gennaio 2000 sarebbe "wine-20000104". Per certi progetti, il formato "Anno Mese Giorno" può avere molto senso.

Le pietre miliari di Mozilla:

> Se si considerano Netscape 6 e le sue versioni commerciali, si nota che la struttura degli sviluppi del progetto Mozilla è uno dei modelli di software libero più complicati in circolazione. La

numerazione delle versioni riflette la situazione, unica nel suo genere, in cui questo progetto viene sviluppato.

La struttura dei numeri di versione di Mozilla storicamente è stata composta di pietre miliari. Sin dall'inizio del progetto Mozilla, gli obiettivi del progetto, nell'ordine e nella misura in cui li si sarebbe dovuti raggiungere, erano tracciati su una serie di road map (http://www.mozilla.org/roadmap.html); i punti ed i risultati principali lungo queste road-map erano contrassegnati come pietre miliari. Quindi, anche se Mozilla era compilato e distribuito ogni sera come "nightly build", nel giorno in cui gli obiettivi di una pietra miliare erano stati raggiunti, quella particolare build era contrassegnata come "release milestone".

Anche se non ho visto impiegare questo metodo in nessun altro progetto sinora, l'idea mi piace, e penso che sia valida per i rami di collaudo o di sviluppo di una grande applicazione con una intensa attività di sviluppo.

2.5. Documentazione

Un numero enorme di applicazioni altrimenti fantastiche è avvizzito e morto perché il loro autore era la sola persona che sapesse usarle appieno. Anche se il proprio programma è scritto principalmente per un gruppo di utenti tecnologicamente preparati, la documentazione è di aiuto, e finanche necessaria, per la sopravvivenza del progetto. Si imparerà più avanti, nel la Sezione 4.3, che bisogna sempre rilasciare qualcosa di utilizzabile. *Un software privo di documentazione non è utilizzabile*.

La documentazione deve essere scritta per un pubblico molto vario, e ci sono molti modi per documentare un progetto. *L'importanza della documentazione all'interno del codice per facilitare lo sviluppo da parte di una comunità estesa è vitale*, ma esula dagli intenti di questo HOWTO. Stando così le cose, questa sezione tratta tecniche utili per la documentazione rivolta agli utenti finali.

Un sistema semi-ufficiale di documentazione, valido per la maggior parte dei progetti di software libero, e che vale la pena di seguire, è il risultato finale di una combinazione di tradizione e necessità. Sia gli utenti che gli sviluppatori si aspettano di poter ottenere documentazione in diversi modi, e se si vuole far decollare il progetto è essenziale fornire, in una forma leggibile, le informazioni che stanno cercando. La gente si aspetta di trovare:

2.5.1. Pagine di manuale

Gli utenti vorrano poter digitare "man nomedelprogetto" ed ottenere una pagina di manuale gradevolmente impaginata che illustri le basi dell'uso dell' applicazione. Ci si assicuri, prima di rilasciare il programma, di averci pensato.

Non è difficile scrivere le pagine di manuale. Eccellente documentazione sul processo di scrittura delle pagine di manuale è disponibile nel “Linux Man-Page-HOWTO”, accessibile tramite il progetto Linux Documentation (LDP) e scritto da Jens Schweikhardt. È reperibile dal suo sito (http://www.schweikhardt.net/man_page_howto.html) o presso (LDP) (http://www.linuxdoc.org/HOWTO/mini/Man-Page.html).

È anche possibile scrivere pagine man usando l’SGML di DocBook. Poiché le pagine man sono così semplici, e la metodologia DocBook relativamente nuova, non ho avuto modo di seguire l’argomento, ma sarei felice di ricevere aiuto da chiunque possa darmi ulteriori informazioni sulla cosa.

2.5.2. Documentazione accessibile a linea di comando

La maggior parte degli utenti si aspetta che una quantità minima di documentazione sia facilmente disponibile dalla linea di comando. Questo tipo di documentazione dovrebbe difficlmente superare una schermata (24 o 25 righe), ma dovrebbe coprire l’uso di base, una breve descrizione del programma (una o due frasi), una lista dei comandi con spiegazione, e tutte le opzioni principali (anche queste con spiegazione), più un riferimento a documentazione più approfondita per chi ne avesse bisogno. La documentazione a linea di comando per il programma Debian apt-get fornisce un ottimo esempio e un utile modello:

```
apt 0.3.19 for i386 compiled on May 12 2000  21:17:27
Usage: apt-get [options] command
       apt-get [options] install pkg1 [pkg2 ...]

apt-get is a simple command line interface for downloading and
installing packages. The most frequently used commands are update
and install.

Commands:
   update - Retrieve new lists of packages
   upgrade - Perform an upgrade
   install - Install new packages (pkg is libc6 not libc6.deb)
   remove - Remove packages
   source - Download source archives
   dist-upgrade - Distribution upgrade, see apt-get(8)
   dselect-upgrade - Follow dselect selections
   clean - Erase downloaded archive files
   autoclean - Erase old downloaded archive files
   check - Verify that there are no broken dependencies

Options:
  -h  This help text.
  -q  Loggable output - no progress indicator
  -qq No output except for errors
  -d  Download only - do NOT install or unpack archives
  -s  No-act. Perform ordering simulation
  -y  Assume Yes to all queries and do not prompt
  -f  Attempt to continue if the integrity check fails
  -m  Attempt to continue if archives are unlocatable
```

```
  -u  Show a list of upgraded packages as well
  -b  Build the source package after fetching it
  -c=? Read this configuration file
  -o=? Set an arbitary configuration option, eg -o dir::cache=/tmp
See the apt-get(8), sources.list(5) and apt.conf(5) manual
pages for more information and options.
```

È diventata una convenzione GNU quella di rendere accessibili queste informazioni con le opzioni "-h" e "--help". La maggior parte degli utenti GNU/Linux si aspetterà di poter recuperare in questo modo la documentazione di base, perciò se si sceglie di usare metodi diversi ci si prepari alle polemiche e alle conseguenze che ne possono derivare.

2.5.3. File che gli utenti si aspettano di trovare

In aggiunta alle pagine man e all'help a linea di comando, ci sono dei file che chi è in cerca di documentazione controllerà, specialmente nei pacchetti che contengono codice sorgente. In una distribuzione di sorgenti, la maggior parte di questi file può essere conservata nella directory radice della distribuzione, o in una sotto-directory chiamata "doc" o "Documentation". File comunemente in queste posizioni sono:

README o Readme

: Un documento che contiene tutte le istruzioni di base per l'installazione, la compilazione, ed anche l'uso: l'insieme minimo di informazioni necessarie per far funzionare il programma. Un README non è l'occasione giusta per essere prolissi, dovrebbe essere conciso ed efficace. Un README ideale è lungo almeno 30 righe, e non più di 250.

INSTALL o Install

: Il file INSTALL dovrebbe essere molto più breve del README, e dovrebbe descrivere brevemente e con rapidità come compilare e installare il programma. Di solito un file di INSTALL dice semplicemente all'utente di lanciare "./configure; make; make install", ed accenna a eventuali opzioni o azioni insolite che possono rendersi necessarie. Per la maggior parte delle procedure di installazione relativamente standard, e per la maggior parte dei programmi, i file INSTALL sono il più brevi possibile: raramente superano le 100 righe.

CHANGELOG, Changelog, ChangeLog, o changelog

: Un CHANGELOG è un file semplice che ogni progetto di software libero ben gestito dovrebbe includere. Un CHANGELOG non è altro che il file che, come suggerisce il nome, registra o documenta i cambiamenti che vengono fatti ad un programma. Il modo più semplice di tenere un CHANGELOG è semplicemente conservare tale file insieme al codice sorgente del programma, e a ogni rilascio aggiungervi in cima una sezione che descrive cosa è stato cambiato, corretto, o aggiunto, al programma. È una buona idea pubblicare il CHANGELOG anche sul sito web, perché può aiutare la gente a capire se vogliono o è necessario aggiornarsi ad una versione più recente, o se invece è meglio aspettare dei miglioramenti più significativi.

NEWS

Un file NEWS e un ChangeLog si assomigliano, ma diversamente da un CHANGELOG, un file NEWS non è solitamente aggiornato in occasione di nuove versioni: ogni qualvolta si aggiungono nuove funzionalità, lo sviluppatore responsabile lo annoterà sul file NEWS. I file NEWS non dovrebbero essere cambiati prima di un rilascio (dovrebbero essere mantenuti aggiornati di continuo), ma di solito è una buona idea controllare in ogni caso perché spesso gli sviluppatori semplicemente dimenticano di mantenerli aggiornati.

FAQ

Per i pochi che ancora non lo sanno, FAQ sta per Frequently Asked Questions (domande frequenti), e una FAQ per l'appunto è una loro raccolta. Non è difficile costruire un file di FAQ: semplicemente si stabilisca una prassi per cui se viene posta una domanda, o si legge una domanda su una mailing list, più di una volta, la domanda (e la relativa risposta) verrà aggiunta alle FAQ. Le FAQ sono meno indispensabili dei file elencati sopra, ma possono far risparmiare tempo, migliorare l'usabilità, e diminuire il mal di testa a tutti quanti.

2.5.4. Sito web

È solo indirettamente una questione di documentazione, ma un buon sito web sta rapidamente diventando una parte essenziale di ogni progetto di software libero. Il sito web dovrebbe fornire accesso alla documentazione (in HTML se possibile); dovrebbe anche includere una sezione per notizie ed eventi relativi al programma, ed una sezione che descriva in dettaglio come partecipare allo sviluppo o al collaudo, invitando esplicitamente alla collaborazione. Dovrebbe anche fornire collegamenti a mailing lists, ad altri siti pertinenti, e fornire un collegamento diretto a tutte le modalità possibili di scaricamento del software.

2.5.5. Altri suggerimenti sulla documentazione

- Tutta la documentazione dovrebbe essere in formato di testo semplice, o, nel caso in cui sia accessibile principalmente dal sito, in HTML: tutti sono in grado di visualizzare un file con cat, tutti sono in grado di visualizzare il testo una pagina per volta, (quasi) tutti sono in grado di leggere documenti HTML. *Se si vogliono distribuire informazioni in PDF, PostScript, RTF, o qualsiasi altro formato ampiamente usato, saranno ben accolte, ma queste informazioni devono essere disponibili anche in testo semplice o HTML, altrimenti c'è chi si arrabbierà molto.* Secondo me anche info ricade in questa categoria. C'è un sacco di fantastica documentazione GNU che qualcuno semplicemente non legge perché c'è solo sotto forma di info. E questo infastidisce *davvero* la gente. Non è una questione di formati migliori o peggiori; è una questione di accessibilità, e lo stato di cose attuale è la causa principale di questa intransigenza;
- non guasta distribuire tutta la documentazione presente sul sito web (FAQ eccetera) insieme al programma. Non si esiti a buttare tutto quanto nel tarball del programma: se la gente non ne avrà bisogno, lo cancellerà. Continuo a ripeterlo: *troppa documentazione non è peccato*;
- a meno che il software sia valido solo per una lingua diversa dall'inglese (un editor di lingua giapponese, per esempio), per favore lo si distribuisca con documentazione in lingua inglese. Se non si

parla l'inglese o comunque non si ha fiducia nel proprio livello di conoscenza, si chieda aiuto ad un amico. Che piaccia o no, giusto o ingiusto che sia, *l'inglese è la lingua del software libero*. Comunque, questo non vuol dire che la documentazione debba essere limitata all'inglese; chi parla un'altra lingua può distribuire traduzioni della documentazione insieme al software, se ha il tempo e l'energia per farlo. Sicuramente la cosa sarà utile a qualcuno;

- infine, *per favore controllare l'ortografia della propria documentazione*; gli errori ortografici nella documentazione sono dei bug. Io sono tra i primi colpevoli di questo errore, che è estremamente facile da commettere. Se l'inglese non è la propria lingua madre, si faccia controllare o correggere la documentazione e le pagine web da qualcuno di madrelingua inglese: ortografia o grammatica imprecise contribuiscono molto a far apparire il codice poco professionale. Nei commenti al codice queste cose sono meno importanti, ma nelle pagine di manuale e nelle pagine web errori del genere non sono accettabili.

2.6. Altri problemi di presentazione

Molti dei restanti problemi legati alla creazione di un nuovo programma di software libero ricadono sotto quello che molta gente chiama buon senso. Si dice spesso che l'ingegneria del software è al 90 per cento buon senso, combinato con il 10 per cento di conoscenze specialistiche; eppure, vale la pena di sottolineare questi problemi, nella speranza che possano far ricordare ad uno sviluppatore qualcosa che aveva dimenticato.

2.6.1. Nomi dei file di pacchetto

Sono d'accordo con Eric Steven Raymond quando dice che: "è utile a tutti che i file di archivio abbiano dei nomi strutturati come in GNU: una radice alfanumerica tutta minuscola, seguita da un punto, seguito da un numero di versione, da un'estensione, ed altri suffissi." Ci sono altre informazioni (inclusi molti esempi di cosa *non* fare) nel suo *Software Release Practices HOWTO*, che è incluso nella bibliografia di questo HOWTO e si può recuperare dal LDP.

2.6.2. Formati dei pacchetti

I formati dei pacchetti possono differenziarsi a seconda del sistema per cui si sta sviluppando: per software basato su Windows gli archivi zip (.zip) di solito sono il formato eletto; se si sta sviluppando per GNU/Linux, *BSD, o qualche UN*X, ci si assicuri che il codice sorgente sia sempre disponibile in formato tar compresso con gzip (.tar.gz). Il compress di UNIX (.Z) è fuori moda ed inutile; la maggior velocità dei computer ha portato alla ribalta bzip2 (.bz2) come mezzo di compressione più efficace. Ora io rendo disponibili tutti i miei rilasci come tarball compressi sia con gzip che con bzip2.

I pacchetti binari dovrebbero sempre essere specifici per una distribuzione; se si ha la possibilità di compilare pacchetti binari per la versione attuale di una delle distribuzioni principali di Linux, gli utenti ne saranno felici. Si cerchi di promuovere le relazioni con gli utenti o gli sviluppatori di grandi distribuzioni, per sviluppare un sistema per la creazione coerente di pacchetti binari. È spesso una buona idea fornire RPM per RedHat (.rpm), deb per Debian (.deb), e RPM di sorgenti (SRPM) se possibile. Da ricordare: *anche se è una cosa gentile fornire questi pacchetti binari, confezionare e rilasciare i sorgenti dovrebbe avere sempre la priorità. Gli utenti o i colleghi sviluppatori possono creare i pacchetti binari, e lo faranno.*

2.6.3. Sistemi di controllo di versione

Un sistema di controllo di versione può rendere meno problematici molti di questi problemi di confezionamento (e molti altri problemi menzionati in questo HOWTO). Se si sta usando *NIX, CVS è la scelta migliore; vi consiglio di tutto cuore il libro di Karl Fogel sull'argomento (e la versione pubblicata in HTML (http://cvsbook.red-bean.com/)).

CVS o no, si farebbe probabilmente bene ad investire un po' di tempo per imparare un sistema di controllo di versione, perché fornisce una soluzione automatica per risolvere molti dei problemi descritti in questo HOWTO. Non sono a conoscenza di alcun sistema di controllo di versione libero per Windows o Mac OS, ma so che esistono client CVS per entrambe le piattaforme. Siti web come SourceForge (http://sourceforge.net) rendono un ottimo servizio, con una interfaccia web per CVS carina e facile da usare.

Vorrei dedicare più spazio in questo HOWTO a CVS, perché mi piace un sacco (lo uso anche per tenere ordine tra le versioni di questo HOWTO!), ma penso che sia al di fuori degli obiettivi di questo documento, e in più ha già i suoi HOWTO dedicati. Il più degno di nota è *CVS Best Practices HOWTO* [CVSBESTPRACTICES], incluso nella bibliografia allegata.

2.6.4. Suggerimenti e dritte utili per la presentazione

Altri suggerimenti utili comprendono:

- *ci si assicuri che il proprio programma sia sempre disponibile nello stesso posto.* Spesso questo significa rendere accessibile via FTP o via web una singola directory in cui possa essere velocemente riconosciuta la versione più recente. Una tecnica efficace è fornire un collegamento simbolico chiamato "ilproprioprogetto-latest" che punti sempre alla versione più recente rilasciata, o di sviluppo, della propria applicazione. Si ricordi che questa locazione riceverà molte richieste di download in corrispondenza delle date di rilascio, perciò ci si assicuri che il server scelto abbia una larghezza di banda adeguata;
- *ci si assicuri che ci sia un indirizzo di email coerente per la segnalazione di bug.* Di solito è una buona idea scegliere a questo scopo qualcosa che NON sia il proprio indirizzo di posta elettronica principale, ad esempio ilproprioprogetto@host o ilproprioprogetto-bugs@host. In questo modo, se mai si decidesse di passare in consegna la manutenzione a qualcun altro, o se il proprio indirizzo di posta cambiasse, basterebbe cambiare la destinazione di inoltro di questo indirizzo speciale. Questo permette anche che sia più di una persona a gestire il flusso entrante di posta che si genererà se il progetto diventerà enorme, come si spera.

3. Mantenere un progetto: interagire con gli sviluppatori

Una volta che il progetto è partito, sono stati superati gli ostacoli più insidiosi nel processo di sviluppo del programma. Porre delle fondamenta salde è essenziale, ma il processo di sviluppo in sé è ugualmente importante, e fornisce altrettante opportunità di fallimento. Nelle prossime due sezioni verrà descritta la conduzione di un progetto, discutendo il modo per sostenere uno sforzo di sviluppo costante attraverso le interazioni con gli sviluppatori e con gli utenti.

Al rilascio del programma, esso diventa software libero. Questa transizione è qualcosa di più che avere un'utenza più vasta: rilasciando il programma come software libero, il *proprio* software diventa software *della comunità del software libero*. Il corso degli sviluppi futuri sarà rimodellato, rediretto, e completamente determinato dagli utenti e, principalmente, dagli altri sviluppatori della comunità.

La differenza principale tra lo sviluppo di software libero e lo sviluppo di software proprietario è la base degli sviluppatori. Come leader di un progetto di software libero, sarà necessario attrarre e tenersi stretti gli sviluppatori, mentre i leader di progetti di software proprietario non devono preoccuparsi di queste cose nella stessa misura. *Come persona che dirige lo sviluppo di un progetto di software libero, si dovrà controllare il lavoro dei colleghi sviluppatori prendendo decisioni responsabili, e scegliendo responsabilmente di non prendere decisioni. Gli sviluppatori dovranno essere diretti senza essere autoritari o dispotici. Si dovrà lottare per guadagnarne il rispetto, e averne sempre per loro.*

3.1. Delegare il lavoro

A questo punto, avete ipoteticamente seguito i primi passi della programmazione di un software, la creazione di un sito web e di un sistema di documentazione, per poi procedere (come sarà descritto nella Sezione 4.3) al rilascio, a beneficio del resto del mondo. Il tempo passa, e se le cose andranno bene la gente sarà interessata e vorrà aiutare. Le patch cominceranno ad affluire.

Come il genitore di un figlio che cresce, è ora tempo di reprimere una smorfia di dispiacere, sorridere, e fare la cosa più difficile nella vita di un genitore: è tempo di lasciarlo andare via.

La delega è il modo politico di descrivere questo processo di "lasciare andare via": è il processo di affidare parte della responsabilità e del potere sul progetto ad altri sviluppatori responsabili e coinvolti. È difficile farlo, per chiunque abbia investito una grande quantità di tempo ed energie in un progetto, ma è essenziale per la crescita di qualsiasi progetto di software libero. Una persona da sola può seguire solo una quantità limitata di cose; un progetto di software libero è nulla senza il coinvolgimento di un *gruppo* di sviluppatori, che può essere mantenuto vivo solo attraverso una guida rispettosa e responsabile, e attraverso la delega.

Mentre il progetto procede, si noteranno persone che investono quantità significative di tempo e di sforzi nel progetto; saranno quelle che inviano più patch, che pubblicano più messaggi sulle mailing list, e che partecipano a lunghe discussioni via email. È responsabilità del manutentore del progetto contattare queste persone, e tentare di spostare un po' del potere e della responsabilità della posizione di manutentore su di loro (se lo vogliono). Ci sono diversi semplici modi per farlo.

Smentendomi un po', delegare non vuol dire necessariamente decidere in comitato. In molti casi sì, ed è provato che funziona; in altri casi ha creato dei problemi. Managing Projects the Open Source Way (http://news.linuxprogramming.com/news_story.php3?ltsn=2000-10-31-001-05-CD) sostiene che "i progetti OSS funzionano bene quando una persona è il leader indiscusso di un team e prende le decisioni importanti (modifiche di progettazione, date di rilascio, e così via)." Questo spesso è vero, ma vorrei spronare gli sviluppatori a prendere in considerazione le idee che il leader del progetto non deve essere necessariamente il fondatore, e che questi importanti poteri non necessariamente devono essere tutti nelle mani di una stessa persona: il gestore dei rilasci può essere diverso dal capo sviluppatore. Queste situazioni sono politicamente delicate, perciò si faccia attenzione e ci si assicuri che sia inevitabile, prima di dare pieni poteri alle persone.

3.1.1. Come delegare

Si potrebbe scoprire che altri sviluppatori sono più esperti o competenti. Il lavoro come manutentori non significa dover essere il migliore o il più brillante; significa avere la responsabilità di mostrare buona capacità di giudizio, e di riconoscere quali soluzioni sono mantenibili e quali no.

Come ogni cosa, è più facile guardare gli altri delegare che farlo in prima persona. In una frase: *essere sempre in cerca di altri sviluppatori qualificati che mostrino un interesse e un coinvolgimento continuato nel proprio progetto, e cercare di spostare la responsabilità verso di loro.* Le idee seguenti potrebbero essere buoni punti di partenza o buone fonti di ispirazione.

3.1.1.1. Concedere ad un gruppo più ampio di persone il permesso di scrittura sul proprio repository CVS, e fare uno sforzo effettivo per promuovere la gestione in comitato

Apache (http://httpd.apache.org/) è un esempio di progetto condotto da un piccolo gruppo di sviluppatori, che votano sulle principali problematiche tecniche e sull'ammissione di nuovi membri, ed hanno tutti accesso in scrittura al repository principale dei sorgenti. Il loro processo è descritto nei dettagli online. (http://httpd.apache.org/ABOUT_APACHE.html)

Il Debian Project (http://www.debian.org/) è un esempio estremo di gestione in comitato. Ad un conteggio aggiornato, più di 700 sviluppatori hanno piena responsabilità su qualche aspetto del progetto; tutti questi sviluppatori possono caricare file sul server FTP principale, e votare sulle problematiche principali. Il corso del progetto è determinato dal suo contratto sociale (http://www.debian.org/social_contract), e da una costituzione (http://www.debian.org/devel/constitution). Per facilitare questo sistema ci sono dei team speciali (ad esempio il team per l'installazione, quello per la lingua giapponese), ed anche un comitato tecnico ed un capo progetto. La responsabilità principale del capo progetto è di “nominare delegati o delegare decisioni al Comitato Tecnico.”

Anche se entrambi questi progetti operano su una scala che il proprio progetto non avrà (almeno all’inizio), il loro esempio è utile. L’idea di Debian di un capo progetto che non fa *niente altro* che delegare serve da esempio portato all’estremo di come un progetto può coinvolgere e dar poteri ad un numero enorme di sviluppatori, e crescere sino ad una dimensione enorme.

3.1.1.2. Nominare pubblicamente qualcuno come gestore del rilascioper uno specifico rilascio

Un gestore del rilascio ha solitamente la responsabilità di coordinare il collaudo, imporre un congelamento del codice, controllarne la stabilità e la qualità, impacchettare il software, e metterlo nei posti appropriati per essere scaricato.

Quest’uso del gestore del rilascio è un buon modo per prendere una pausa e spostare su qualcun altro la responsabilità di accettare e rifiutare patch. È un buon modo di definire molto chiaramente un segmento di lavoro del progetto come appartenente ad una certa persona, ed è un ottimo modo di concedere a se stessi lo spazio per respirare.

3.1.1.3. Delegare il controllo di un intero ramo

Se il proprio progetto sceglie di avere rami (come descritto nel la Sezione 3.3), potrebbe essere una buona idea nominare qualcun altro responsabile di un ramo. Se si preferisce concentrare le proprie energie sui rilasci di sviluppo e sull’implementazione di nuove funzionalità, si lasci il controllo totale sui rilasci stabili ad uno sviluppatore adatto al compito.

L’autore di Linux, Linus Torvalds, incoronò pubblicamente Alan Cox come “l’uomo dei kernel stabili”: tutte le patch per i kernel stabili vanno ad Alan, e se Linus fosse per qualsiasi ragione strappato via dal suo lavoro su Linux, Alan Cox sarebbe più che adeguato a prendere il suo posto, in quanto erede riconosciuto per la manutenzione di Linux.

3.2. Accettare e rifiutare patch

Questo HOWTO ha già accennato al fatto che come manutentori di un progetto di software libero, una delle primarie e più importanti responsabilità sarà accettare e rifiutare patch inviate da altri sviluppatori.

3.2.1. Incoraggiare un buon patching

Come persone che gestiscono o curano la manutenzione del progetto non si realizzeranno materialmente molte patch. Comunque val la pena di conoscere la sezione di ESR sulle *Buone pratiche di patching* nel *Software Release Practices HOWTO*[ESRHOWTO]. Non sono d'accordo con la sua affermazione che le patch bruttissime da vedere o non documentate meritino probabilmente di essere gettate via alla prima occhiata: semplicemente, questa non è stata la mia esperienza, specialmente avendo a che fare con correzioni di bug che spesso non sono affatto nella forma di patch. Naturalmente, questo non vuol dire che *mi piace* ricevere patch fatte malamente: se si ricevono brutte patch, se si ricevono patch senza alcuna documentazione, specialmente se sono qualcosa di più di correzioni di bug banali, potrebbe valer la pena di giudicare la patch secondo qualcuno dei criteri spiegati nell'HOWTO citato, e poi inviare alla gente il link al documento cosicché possano rifarla nel "modo giusto."

3.2.2. Giudizio tecnico

In *Open Source Development with CVS*, Karl Fogel sostiene in modo convincente che le cose più importanti da ricordare quando si rifiutano o accettano patch sono:

- una solida conoscenza dello scopo del programma (è l' "idea" di cui si parlava in la Sezione 2.1);
- la capacità di riconoscere, facilitare, e dare una direzione all'"evoluzione" del programma, cosicché esso possa crescere e cambiare e incorporare funzionalità che non erano state originariamente previste;
- la necessità di evitare divagazioni che possano espandere troppo la portata del programma, divagazioni che spingerebbero il progetto verso una morte prematura sotto il peso della propria ingestibilità.

Questi sono i criteri che, come manutentori del progetto, si dovrebbero tenere in conto ogni volta che si riceve una patch.

Fogel approfondisce la questione, ed afferma che "le domande da porsi quando si deve decidere se implementare (od approvare) un cambiamento sono:"

- porterà benefici ad una percentuale significativa della comunità di utenti del programma?
- si adatta bene al dominio del programma, o ad una estensione naturale ed intuitiva di tale dominio?

Le risposte a queste domande non sono mai semplici, ed è sicuramente possibile (e persino probabile) che la persona che ha inviato la patch possa avere opinioni diverse dalla propria su tali risposte. Tuttavia, se la risposta ad almeno una di queste domande è "no," è proprio dovere rifiutare il cambiamento: contrariamente, il progetto diventerà ingestibile e non mantenibile, e potrà alla lunga fallire.

3.2.3. Rifiutare delle patch

Rifiutare una patch è probabilmente il lavoro più difficile e delicato che il manutentore di un qualsiasi progetto di software libero deve affrontare; ma qualche volta deve essere fatto. Come accennato prima (nel la Sezione 3 e nel la Sezione 3.1), si dovrà cercare di bilanciare le proprie responsabilità e i propri poteri nel prendere quelle che si pensa siano le migliori decisioni tecniche, con il fatto che si potrà perdere supporto dagli altri sviluppatori sembrando ubriachi di potere o troppo autoritari o possessivi verso il progetto, che dopotutto appartiene alla comunità. Si consiglia di tenere a mente questi tre concetti principali quando si rifiutano delle patch (o altri cambiamenti).

3.2.3.1. Proporlo alla comunità

Uno dei modi migliori di giustificare una decisione di rifiutare una patch, cercando di non far sembrare di mantenere una presa ferrea sul proprio progetto, è di non prendere la decisione completamente da soli. Potrebbe aver senso redirigere le proposte di cambiamenti e le decisioni più difficili verso una mailing list di sviluppo dove possano essere discussi e dibattuti. Ci saranno alcune patch (correzione di bug, etc.) che saranno sicuramente accettate, ed alcune che saranno ritenute così fuori posto da non meritare neanche un'ulteriore discussione: sono quelle che rientrano nella zona grigia tra questi due gruppi a poter meritare un rapido inoltro ad una mailing list.

Si consiglia caldamente di seguire questo processo. Come manutentore del progetto si avrà la preoccupazione di prendere la decisione migliore per il progetto, per gli utenti e gli sviluppatori del progetto, e per voi stessi come capi progetto responsabili: redirigere le cose ad una mailing list dimostrerà il proprio senso di responsabilità e la propria conduzione attenta, poiché chiede lumi sugli interessi della comunità per poter meglio porsi al suo servizio.

3.2.3.2. I problemi tecnici non sono sempre una buona giustificazione

Specialmente all'inizio della vita del progetto, si scoprirà che molti cambiamenti sono difficili da implementare, introducono nuovi bug, o hanno altri problemi tecnici. Si cerchi di vedere oltre: specialmente per le funzionalità aggiunte, le buone idee non sempre vengono da buoni programmatori. Il valore tecnico è una ragione valida per rimandare l'applicazione di una patch, ma non è sempre una buona ragione per rifiutare un cambiamento immediatamente. Anche i cambiamenti piccoli valgono lo sforzo di lavorare insieme allo sviluppatore per risolvere i bug ed incorporare il cambiamento, se si pensa che sia una buona aggiunta al progetto: questo sforzo contribuirà a rendere il proprio progetto un progetto della comunità, attirerà uno sviluppatore nuovo o con meno esperienza nel progetto, ed insegnerà perfino ai suoi membri qualcosa che potrebbe tornare utile nel preparare la prossima patch.

3.2.3.3. Buone maniere

Dovrebbe essere superfluo dirlo, ma *prima di tutto, in ogni occasione, essere cortesi.* Se qualcuno ha un'idea, e ci tiene a sufficienza da scrivere del codice ed inviare una patch, significa che ci tiene davvero, è motivato, ed è già coinvolto: il proprio obiettivo come manutentore è assicurarsi che ne invii altre. Può aver fatto cilecca questa volta, ma la prossima volta la sua potrà essere l'idea o la funzionalità che rivoluzionerà il progetto.

Per prima cosa, è proprio dovere giustificare con chiarezza e concisione la scelta di non incorporare un cambiamento, e ringraziare; far capire che l'aiuto è stato molto apprezzato, e che dispiace davvero non poter incorporare i cambiamenti. Poi far capire che si desidera che l'autore resti coinvolto, e che si spera che la prossima patch o idea si integri meglio col progetto, perché il suo lavoro è stato apprezzato e si vorrebbe vederlo nella propria applicazione. Se mai è capitato di veder rifiutata una propria patch dopo avervi investito una grossa quantità di tempo, riflessione, ed energie, si cerchi di ricordare come ci si sente: non è piacevole; tenerlo a mente quando si dovrà deludere qualcuno. Non è mai facile, ma è necessario fare tutto il possibile per renderlo il meno spiacevole possibile.

3.3. Rami stabili e rami di sviluppo

L'idea di avere rami stabili e rami di sviluppo è stata già descritta brevemente nel la Sezione 2.4 e nel la Sezione 3.1.1.3; questi cenni testimoniano alcuni dei modi in cui i rami multipli possono influenzare il software. I rami possono far evitare (in qualche misura) alcuni dei problemi legati al rifiuto delle patch (descritti nel la Sezione 3.2), in quanto permettono di compromettere temporaneamente la stabilità del progetto senza incidere sugli utenti che hanno bisogno di quella stabilità.

Il modo più comune di dividere in rami il progetto è avere un ramo stabile e uno per lo sviluppo. Questo è il modello seguito dal kernel Linux, ed è descritto nel la Sezione 2.4. In questo modello, c'è *sempre* un ramo stabile ed un ramo in sviluppo. Prima di ogni nuovo rilascio, il ramo di sviluppo entra in un "blocco delle funzionalità", come descritto nel la Sezione 3.4.1, in cui i cambiamenti principali e le funzionalità aggiunte sono rifiutati o messi in attesa fino a che il kernel di sviluppo viene rilasciato come nuovo ramo stabile, e gli sviluppi principali riprendono sul nuovo ramo di sviluppo. Le correzioni di bug e i piccoli cambiamenti che presumibilmente non avranno grandi ripercussioni negative, vengono incorporati sia nel ramo stabile che nel ramo di sviluppo.

Il modello di Linux fornisce un esempio estremo. In molti progetti non c'è bisogno di avere costantemente disponibili due versioni: può avere senso avere due versioni solo in prossimità di una release. Il progetto Debian ha sempre storicamente reso disponibili sia una distribuzione stabile che una instabile, ed ha allargato l'insieme sino ad includere: versione stabile, instabile, di collaudo, sperimentale, e (intorno alla data di rilascio) una distribuzione congelata che incorpora solo correzioni di bug nella transizione da instabile a stabile. Ci sono pochi progetti la cui dimensione avrebbe bisogno di un sistema come quello di Debian; tuttavia, questo uso dei rami dimostra come essi possano essere usati per far coesistere uno sviluppo coerente ed efficace con il bisogno di produrre rilasci regolari ed utilizzabili.

Nel cercare di impostare un albero di sviluppo, può essere utile tenere a mente alcune cose.

Minimizzare il numero di rami

Debian può riuscire a fare buon uso di quattro o cinque rami, però contiene dei gigabyte di software, in più di 5000 pacchetti, compilati per 5 o 6 diverse architetture. Due è probabilmente un buon limite superiore. Troppi rami confonderanno i propri utenti (non si possono contare le volte che ho dovuto descrivere il sistema di Debian, quando ancora aveva solo due o certe volte tre rami!), i potenziali sviluppatori, ed anche voi stessi. I rami possono aiutare, ma hanno un costo, perciò vanno usati con molta parsimonia.

Assicurarsi che tutti i diversi rami abbiano una spiegazione

Come accennato nel paragrafo precedente, rami differenti *confonderanno* gli utenti. Si faccia tutto il possibile per evitarlo, spiegando chiaramente i diversi rami in una pagina ben visibile sul proprio sito, ed in un file README nella directory FTP o web.

Si potrebbe anche mettere in guardia da un errore che Debian potrebbe aver fatto: i termini "instabile," "in collaudo," e "sperimentale" sono vaghi, e difficili da classificare in ordine di stabilità (o di instabilità, secondo le circostanze); si cerchi di spiegare a qualcuno che "stabile" significa in realtà "ultra stabile", e che "instabile" non comprende in realtà alcun software instabile, ma è in realtà software stabile non collaudato come distribuzione.

Se si ha l'intenzione di usare dei rami, specialmente nelle prime fasi, si ricordi che le persone sono abituate a comprendere i termini "stabile" e "di sviluppo", e con questa semplice e comune divisione dei rami probabilmente non si sbaglierà.

Assicurarsi che tutti i rami siano sempre disponibili

Come molte delle cose scritte in questo documento, probabilmente non dovrebbe essere necessario dirlo ma l'esperienza insegna che non è sempre ovvio per tutti: è una buona idea separare fisicamente rami diversi in directory, o alberi di directory diversi sul proprio sito FTP o web. Linux lo realizza mantenendo i kernel in sottodirectory v2.2, v2.3, ecc., così è immediatamente ovvio (una volta che si conosce il loro schema di numerazione delle versioni) quale directory è per il più recente rilascio stabile, e quale per lo sviluppo corrente. Debian lo realizza dando a tutte le distribuzioni dei nomi (ad esempio woody, potato, etc.) e modificando i collegamenti simbolici chiamati "stable," "unstable" e "frozen" per puntare (per nome) a quella certa distribuzione che si trova in quel certo stato. Entrambi i metodi funzionano, e ce ne sono altri: in ogni caso, è importante che i diversi rami siano sempre disponibili, siano accessibili da posizioni coerenti, e che i diversi rami siano chiaramente distinguibili l'uno dall'altro, cosicché gli utenti sappiano esattamente cosa vogliono e dove trovarlo.

3.4. Altri problemi nella gestione dei progetti

In un progetto di software libero possono sorgere altri problemi nell'interazione con gli sviluppatori, ma non è possibile trattarli in dettaglio in un HOWTO di queste dimensioni, e con questi obiettivi. Vi prego di non esitare a contattarmi se scoprite delle omissioni importanti.

Altre più piccole questioni che vale la pena di menzionare sono:

3.4.1. Congelamento

Per quei progetti che scelgono di adottare un modello di sviluppo suddiviso (la Sezione 3.3), il congelamento è un concetto con cui vale la pena di familiarizzare.

I congelamenti possono avere due forme principali. Un "congelamento delle funzionalità" è un periodo in cui al programma non vengono aggiunte funzionalità significative. Le funzionalità esistenti (anche ossature di funzionalità a malapena funzionanti) possono essere migliorate e perfezionate, ed è un periodo in cui si correggono i bug. Questo tipo di congelamento viene solitamente messo in atto qualche tempo (un mese o due) prima di un rilascio: è facile rimandare un rilascio in attesa di "ancora una sola funzionalità", e un congelamento aiuta ad evitare questa situazione, piantando i paletti che servono; dà agli sviluppatori lo spazio di cui hanno bisogno per rendere un programma pronto per il rilascio.

Il secondo tipo di congelamento è il "congelamento del codice", che è molto più simile al rilascio di un software: una volta che un software è entrato in un "congelamento del codice", tutti i cambiamenti al codice sono scoraggiati e sono consentite solo le modifiche volte a correggere bug noti. Questo tipo di congelamento di solito segue un "congelamento delle funzionalità", e precede di poco un rilascio. La maggior parte del software rilasciato è in quello che potrebbe essere interpretato come una sorta di "congelamento del codice" di alto livello.

Anche si sceglie di non nominare mai un gestore dei rilasci (la Sezione 3.1.1.2), se è in atto un congelamento dichiarato pubblicamente sarà più agevole giustificare il rifiuto o lo slittamento di patch (la Sezione 3.2) prima di un rilascio.

3.5. Forks

Non ero sicuro sul modo di trattare il forking in questo documento (o se dovessi davvero trattarlo). Un "fork" si ha quando un gruppo di sviluppatori prende del codice da un progetto di software libero e lo usa per iniziare un progetto di software libero completamente nuovo. L'esempio più famoso di fork fu quello tra Emacs and XEmacs: i due emacs sono basati sulla stessa base di codice, ma per ragioni tecniche, politiche e filosofiche lo sviluppo fu spezzato in due progetti, che ora sono in competizione.

La versione breve della sezione sul fork è: *non lo si faccia.* I fork costringono gli sviluppatori a scegliere di lavorare con uno solo dei due progetti, causando divisioni politiche molto spiacevoli, e lavoro ridondante. Fortunatamente, di solito la sola minaccia di un fork è sufficiente a spaventare il manutentore o i manutentori del progetto, al punto di convincerli a correggere il loro modo di condurlo.

Nel suo capitolo su "The Open Source Process", Karl Fogel descrive il modo migliore di fare un fork, se proprio è necessario; se si è stabilito che è assolutamente necessario, e che le divergenze con le persone che minacciano il fork sono assolutamente senza soluzione, il libro di Fogel sarà buon punto di partenza.

4. Mantenere un progetto: interagire con gli utenti

Se avete seguito fino a qui, congratulazioni: ci si sta avvicinando alla fine di questo documento. Questa ultima sezione descrive alcune delle situazioni in cui, nella propria veste di manutentori del progetto, si interagisce con gli utenti, e fornisce alcuni suggerimenti su come queste situazioni possano essere gestite efficacemente.

Interagire con gli utenti è difficile. Nella discussione dell'interazione con gli sviluppatori, l'assunto di partenza è che in un progetto di software libero un manutentore del progetto debba costantemente adoperarsi per attrarre e conservarsi gli sviluppatori, che possono andarsene con facilità in qualsiasi momento.

Gli utenti della comunità del software libero sono diversi dagli sviluppatori, e sono anche diversi dagli utenti del mondo del software proprietario, quindi dovrebbero essere trattati diversamente da ciascuno di questi due gruppi. Seguono alcuni dei punti in cui i gruppi differiscono significativamente:

- le linee di demarcazione tra utenti e sviluppatori si confondono in modo totalmente estraneo a qualsiasi modello di sviluppo proprietario: i propri utenti sono spesso i propri sviluppatori, e viceversa;
- nel mondo del software libero, spesso si sarà l'unica scelta per i propri utenti. Poiché nel mondo del software libero c'è così tanta enfasi sul non replicare il lavoro degli altri, e poiché l'elemento della competizione, presente nel modello del software proprietario, è assente (o almeno è presente in una forma estremamente diversa) nel modello di sviluppo del software libero, probabilmente il proprio sarà il solo progetto che fa quello che fa, nel modo in cui lo fa. Questo vuol dire che la propria ricettività nei confronti degli utenti è ancora più importante che nel mondo del software proprietario;
- in modo quasi paradossale, i progetti di software libero soffrono conseguenze meno immediate o terribili dall'ignorare del tutto i propri utenti. Spesso è anche la cosa più facile da fare: poiché non si deve competere con un altro prodotto, ci sono buone possibilità che non ci si dovrà affrettare a raggiungere le funzionalità dell'ultimo prodotto della concorrenza. Questo vuol dire che il processo di sviluppo dovrà essere guidato o da una forte automotivazione, o da un obbligo morale verso i propri utenti, o da entrambi.

Si può cercare di affrontare questa situazione, unica nel suo genere, solo indirettamente: gli sviluppatori e i manutentori devono necessariamente dare ascolto agli utenti, ed essere quanto più ricettivi possibile. Una solida conoscenza della situazione descritta sopra è lo strumento migliore a disposizione di uno sviluppatore di software libero per correggere il suo stile di sviluppo o di conduzione del progetto per meglio adattarsi a questo singolare processo. Questi capitoli cercheranno di introdurre alcuni dei punti più difficili o importanti nelle interazioni di un progetto con gli utenti, e forniranno alcuni consigli su come affrontare tali interazioni.

4.1. Collaudo e collaudatori

Gli utenti, oltre che essere sviluppatori, sono anche (e forse più frequentemente) collaudatori. Prima di diventare il bersaglio di invettive, riformulo la frase: *alcuni utenti* (quelli che si prestano esplicitamente come volontari) sono anche collaudatori.

È importante che questa distinzione sia fatta al più presto, perché non tutti gli utenti vogliono diventare collaudatori: molti utenti vogliono usare software stabile, e non gli interessa avere l'ultimo, bellissimo software, con le ultime, bellissime funzionalità. Questi utenti si aspettano un software stabile, collaudato, senza bug importanti o evidenti, e si arrabbieranno se si ritroveranno a fare da collaudatori. Questo è un'altro modo ancora in cui un modello di sviluppo separato (citato nel la Sezione 3.3) può tornare utile.

"Managing Projects the Open Source Way (http://news.linuxprogramming.com/news_story.php3?ltsn=2000-10-31-001-05-CD)" descrive ciò di cui un buon collaudo dovrebbe andare in cerca.

Condizioni limite

: Lunghezze massime dei buffer, conversione dati, limiti superiori/inferiori, e così via.

Comportamento inappropriato

: È una buona idea scoprire cosa farà un programma se un utente gli fornisce un valore che non si aspetta, preme il pulsante sbagliato, ecc. Porsi un sacco di domande del tipo "cosa succederebbe se"; pensare a qualsiasi cosa che *potrebbe* fallire o *potrebbe* andare storto, e scoprire cosa il programma fa in quei casi.

Fallimenti gradevoli

: La risposta a molti dei "cosa succederebbe se" di cui sopra è probabilmente "un fallimento", che spesso è la sola risposta possibile. Ora, ci si assicuri che avvenga gradevolmente, e che quando il programma si arresta dia qualche indicazione del perché si è arrestato o ha dato un malfunzionamento, cosicché l'utente o il programmatore capiscano cosa sta succedendo.

Conformità agli standard

: Se possibile, ci si assicuri che i propri programmi siano conformi agli standard. Se il programma è interattivo, non essere troppo creativi con le interfacce: se non è interattivo, assicurarsi che comunichi con altri programmi, e con il resto del sistema, attraverso canali appropriati e comunemente riconosciuti.

4.1.1. Collaudo automatizzato

Per molti programmi, molti errori comuni possono essere individuati con mezzi automatici. I test automatizzati sono solitamente abbastanza efficaci nell'individuare errori in cui si è incappati spesso in passato, o semplici dimenticanze; non sono molto efficaci nel trovare errori, anche importanti, che siano completamente imprevisti.

CVS è fornito di uno script per la shell Bourne chiamato sanity.sh, che vale la pena di guardare; Debian usa un programma, lintian, che controlla i pacchetti Debian in cerca di tutti gli errori più comuni. Anche se l'uso di questi script può non essere di aiuto, c'è un gran numero di altri software per il controllo dell'integrità sulla rete che può fare al proprio caso (ci si senta liberi di inviarmi delle segnalazioni); nessuno di essi produrrà un rilascio privo di bug, ma eviteranno almeno alcune sviste importanti. Infine, se i propri programmi diventano uno sforzo a lungo termine, si scoprirà che ci sono certi errori che si tende a ripetere: si dia inizio ad una raccolta di script che individuano questi errori, per contribuire a tenerli fuori dai rilasci futuri.

4.1.2. Collaudo eseguito da collaudatori

Per tutti i programmi che si basano sull'interattività con l'utente, molti bug saranno scoperti solo attraverso il collaudo eseguito da utenti, che premono veramente i tasti e i pulsanti del mouse. Per questo avete bisogno di collaudatori: il maggior numero possibile di collaudatori.

La parte più difficile del collaudo è trovare i collaudatori; è solitamente una buona tattica pubblicare su una mailing list o un newsgroup pertinente un messaggio che annuncia una determinata data di rilascio prevista, e descrive a grandi linee le funzionalità del programma: se si dedica un po' di tempo alla stesura dell'annuncio, si avrà la certezza di ricevere qualche risposta.

La seconda parte più difficile del collaudo è *mantenere* i collaudatori, e mantenerli attivamente coinvolti nel processo di collaudo. Fortunatamente, ci sono alcune tattiche sperimentate che si possono applicare a questo scopo.

Rendere le cose facili ai collaudatori

I collaudatori stanno facendo un favore, perciò si renda loro la vita quanto più facile possibile. Questo significa che si dovrebbe avere cura di impacchettare il proprio programma in modo che sia facile da trovare, estrarre, installare, e disinstallare; significa anche che si dovrebbe spiegare ad ogni collaudatore che cosa si sta cercando, e rendere il modo per segnalare i bug semplice e certo. La chiave è fornire una struttura il più definita possibile, per rendere il lavoro dei collaudatori facile, ma di conservare quanta più flessibilità possibile, per quelli che vogliono fare le cose in modo un po' diverso.

Essere reattivi verso i collaudatori

Quando i collaudatori inviano delle segnalazioni di bug gli si risponda, e in fretta: anche se si risponde solo per dire che il bug è già stato corretto, risposte veloci e coerenti danno la sensazione che il loro lavoro sia ascoltato, importante, ed apprezzato.

Ringraziare i propri collaudatori

Ringraziare personalmente i collaudatori ogni volta che mandano una patch. Ringraziarli pubblicamente nella documentazione, e nella sezione "about" del proprio programma. Essere grati ai propri collaudatori, il proprio programma non sarebbe possibile senza il loro aiuto: assicurarsi che lo sappiano. Si dia loro una pacca sulla spalla pubblicamente, per essere sicuri che il resto del mondo lo sappia: sarà apprezzato più di quanto si possa immaginare.

4.2. Impostare un'infrastruttura di supporto

Anche se il collaudo è importante, la gran parte delle interazioni e responsabilità verso gli utenti ricade nella categoria del supporto. Il modo migliore di essere certi che gli utenti siano adeguatamente supportati nell'uso del programma è impostare una buona infrastruttura a questo scopo, cosicché gli sviluppatori e gli utenti si aiutino a vicenda, e su di voi ricada un peso più leggero; in questo modo, la gente otterrà anche risposte migliori e più tempestive alle proprie domande. Questa infrastruttura può avere diverse forme principali.

4.2.1. Documentazione

Non dovrebbe sorprendere che il punto chiave per ogni infrastruttura di supporto sia una buona documentazione. Questo argomento è stato ampiamente trattato nel la Sezione 2.5, e non sarà ripetuto qui.

4.2.2. Mailing list

A parte la documentazione, mailing list efficaci sono lo strumento più importante per fornire supporto agli utenti. Portare avanti una mailing list nel modo giusto è qualcosa di più complicato che semplicemente installare su una macchina un software che gestisca mailing list.

4.2.2.1. Liste separate

Una buona idea è separare le mailing list per gli utenti e per gli sviluppatori (magari project-user@host e project-devel@host), e far rispettare la divisione: se la gente pubblica una domanda di sviluppo su -user, si chieda gentilmente di ripubblicarla su -devel, e viceversa. Si sottoscrivano entrambi i gruppi, e si incoraggino tutti gli sviluppatori principali a fare lo stesso.

Questo sistema fa sì che nessuno resti bloccato a fare tutto il lavoro di supporto, e fa sì che gli utenti ne sappiano di più sul programma, in modo da poter aiutare i nuovi utenti rispondendo alle loro domande.

4.2.2.2. Scegliete bene il software per le mailing list

Il software per le mailing list non dovrebbe essere scelto d'impulso: si cerchi la maggior semplicità possibile, pensando ad una facile accessibilità per utenti senza molta esperienza tecnica. Anche l'accessibilità via web agli archivi della lista è importante.

I due principali programmi di software libero per la gestione di mailing list sono majordomo (http://www.greatcircle.com/majordomo/) e GNU Mailman (http://www.list.org/). Dopo essere stato per lungo tempo un sostenitore di majordomo, ora consiglierei ad ogni progetto di usare GNU Mailman. Quest'ultimo rispetta i criteri sopra elencati e la fa più semplice: fornisce un buon programma di gestione di mailing list ad un manutentore di un progetto di software libero, piuttosto che una buona applicazione di gestione di mailing list ad un amministratore di mailing list.

Ci sono altre cose da tenere in considerazione nel mettere in piedi una mailing list: se fosse possibile collegare le mailing list a Usenet, e fornirle sia sotto forma di digest che renderle disponibili sul web, si farà piacere ad alcuni utenti, e l'infrastruttura di supporto sarà un po' più accessibile.

4.2.3. Altre idee di supporto

Una mailing list e della documentazione accessibile non esauriscono quello che si può fare per mettere in piedi una buona infrastruttura di supporto agli utenti: siate creativi. Se inciampate in qualcosa che funziona bene, mandatemi una email e lo includerò qui.

4.2.3.1. Rendetevi accessibili

Non sono mai troppi i mezzi per farsi contattare: se si frequenta spesso un canale IRC, non si esiti ad elencarlo nella documentazione di progetto; riportare gli indirizzi di posta elettronica ed ordinaria, e i modi per essere raggiunti via ICQ, AIM, o Jabber, se pertinenti.

4.2.3.2. Software di gestione dei bug

Per molti grandi progetti software, l'uso di software di gestione dei bug è essenziale per tenere traccia di quali errori sono stati corretti, quali non sono stati corretti, e quali sono in corso di correzione, e da parte di chi. Debian usa il Debian Bug Tracking System (http://bugs.debian.org) (BTS), anche se può non essere la scelta migliore per ogni progetto (sembra che attualmente si stia piegando sotto il suo stesso peso). Oltre ad un browser web davvero buono, il progetto Mozilla ha generato un sotto-progetto che ha avuto come conseguenza la nascita di un sistema di tracciamento dei bug chiamato bugzilla (http://www.mozilla.org/projects/bugzilla/), che è diventato estremamente affidabile, e che mi piace molto.

Questi sistemi (ed altri simili) possono essere poco maneggevoli, perciò gli sviluppatori dovrebbero fare attenzione a non passare più tempo sul sistema di tracciamento dei bug che sui bug, o sui progetti, stessi. Se un progetto continua a crescere, l'uso di un sistema di tracciamento può fornire una semplice via standard per utenti e collaudatori per segnalare bug, e per sviluppatori e manutentori per correggerli, e per tenerne traccia in modo ordinato.

4.3. Rilasciare il programma

Come accennato in precedenza, la prima regola dei rilasci è: *rilasciare qualcosa di utile*. Un software che non funziona, o non è utile, non attrarrà nessuno verso il proprio progetto; la gente se ne allontanerà, e probabilmente la prossima volta che vedrà annunciare una nuova versione tirerà diritto senza fermarsi. Un software che funziona a metà, se è utile, incuriosirà la gente, desterà il loro appetito per le prossime versioni, e li incoraggerà a partecipare al processo di sviluppo.

4.3.1. Quando rilasciare

Prendere la decisione di rilasciare il software per la prima volta è incredibilmente importante, e incredibilmente stressante; ma bisogna farlo. Si cerchi di fare qualcosa che sia abbastanza completo da essere usabile, ed abbastanza incompleto da lasciare una certa flessibilità ed un certo spazio per l'immaginazione dei futuri sviluppatori. Non è una decisione facile; si chieda aiuto alla mailing list di un Linux User Group locale, o ad un gruppo di amici sviluppatori.

Una tattica è produrre prima un rilascio "alpha" o "beta", come descritto sotto nel la Sezione 4.3.3. In ogni caso, la maggior parte delle linee guida descritte sopra si applicano ancora.

Quando si sente istintivamente che è il momento giusto, e si ritiene di avere ben soppesato la situazione diverse volte, si incrocino le dita, e si salti il fosso.

Dopo aver rilasciato per la prima volta, decidere quando rilasciare diventa meno stressante, ma altrettanto difficile da giudicare. Mi piacciono i criteri per mantenere un buon ciclo di rilasci che fornisce Robert Krawitz nel suo articolo "Free Software Project Management" (http://www.advogato.org/article/196.html). Consiglia di chiedersi: "questo rilascio..."

- contiene un numero di nuove funzionalità o di correzioni di bug sufficiente a giustificare lo sforzo?
- è abbastanza lontano dal precedente da lasciare all'utente il tempo di lavorarci?
- funziona sufficientemente bene da consentire all'utente di portare a termine il proprio lavoro (qualità)?

Se la risposta a queste tre domande è sì, probabilmente è tempo per un rilascio. Se in dubbio, si ricordi che chiedere un consiglio non guasta.

4.3.2. Come rilasciare

Se si sono seguite le linee guida descritte in questo HOWTO fino a questo punto, la tecnica per produrre un rilascio sarà la parte facile della cosa: se sono state predisposte ubicazioni coerenti per la distribuzione, e il resto dell'infrastruttura descritta nelle sezioni precedenti, per rilasciare basta costruire il pacchetto, controllarlo quando è finito, caricarlo nel posto appropriato e poi far sì che il sito web registri l'aggiornamento.

4.3.3. Rilasci alpha, beta, e di sviluppo

Quando si progettano dei rilasci, vale la pena di considerare il fatto che non tutti i rilasci devono per forza essere dei rilasci completamente numerati, gli utilizzatori di software sono abituati ai pre-rilasci. Tuttavia, si dovrà fare attenzione ad etichettare questi rilasci accuratamente, altrimenti causeranno più problemi di quanto necessario.

Si osserva spesso che molti sviluppatori di software libero sembrano avere le idee confuse sul ciclo di rilascio. "Managing Projects the Open Source Way (http://news.linuxprogramming.com/news_story.php3?ltsn=2000-10-31-001-05-CD)" suggerisce di memorizzare la frase: "Una Alpha non è una Beta. Una Beta non è un Rilascio"; penso anch'io che sia una buona idea.

Rilasci alpha

Il software in alpha ha funzionalità complete, ma talvolta solo parzialmente funzionanti.

Ci si aspetta che i rilasci alpha siano instabili, forse un po' insicuri, ma certamente utilizzabili; *possono* avere bug noti e ghiribizzi che devono ancora essere risolti. Prima di rilasciare un'alpha, si ricordi che *i rilasci alpha sono comunque rilasci*, la gente non si aspetta nightly build dei sorgenti su CVS: un'alpha dovrebbe funzionare, ed aver passato ad un minimo di collaudo e correzione di bug.

Rilasci beta

Il software in beta ha funzionalità complete ed è operativo, ma è sotto collaudo, e contiene qualche bug ancora da risolvere.

Ci si aspetta in genere che i rilasci Beta siano utilizzabili e leggermente instabili, anche se sicuramente *non insicuri*. I rilasci beta di solito non ammettono un rilascio completo a meno di un mese di distanza; possono contenere piccoli bug noti, ma nessuno importante. Tutte le funzionalità principali dovrebbero essere completamente implementate, anche se sui dettagli precisi ci può ancora essere del lavoro da fare. I rilasci beta sono un ottimo strumento per aguzzare l'appetito dei potenziali utenti fornendo loro una visione molto realistica di dove il progetto andrà in un prossimo futuro, e può contribuire a mantenere vivo l'interesse, fornendo alla gente *qualcosa*.

Rilasci di sviluppo

"Rilascio di sviluppo" è un termine molto più vago di "alpha" o "beta". Di solito scelgo di riservare questa espressione per la discussione di un ramo di sviluppo, anche se ci sono altri modi di usarla. Così tanti modi, in realtà, che secondo me questa espressione è troppo inflazionata: il popolare gestore di finestre Enlightenment (http://www.enlightenment.org) non ha rilasciato *niente altro che* rilasci di sviluppo. Molto spesso si usa questo termine per descrivere rilasci che non sono ancora neanche alpha o beta; se dovessi rilasciare una versione pre-alpha di un pacchetto software per mantenere vivo l'interesse per un mio progetto, probabilmente la etichetterei così.

4.4. Annunciare il progetto

Bene, è fatta. Il proprio progetto di software libero è stato (almeno per gli scopi di questo HOWTO) progettato, costruito, e rilasciato. Tutto ciò che rimane da fare è dirlo al mondo, cosicché vengano a provarlo e, si spera, saltino a bordo per lo sviluppo. Se ogni cosa è in ordine come descritto sopra, questo sarà un processo rapido e indolore: basterà un tempestivo annuncio per mettersi sullo schermo radar della comunità del software libero.

4.4.1. Mailing list e Usenet

Annunciare il proprio software sul gruppo Usenet comp.os.linux.announce (news:comp.os.linux.announce). Se si vuole mettere l'annuncio solo in due posti, scegliere c.o.l.a e freshmeat.

In ogni caso, la posta elettronica è ancora il modo in cui su Internet la maggior parte della gente riceve informazioni: è una buona idea mandare un messaggio che annuncia il programma ad ogni mailing list pertinente che si conosce, e ad ogni altro gruppo di discussione Usenet pertinente.

Karl Fogel raccomanda di inserire nel campo oggetto una semplice descrizione del fatto che il messaggio è un annuncio, il nome del programma, la versione, ed una descrizione della sua funzionalità lunga mezza riga. In questo modo, un eventuale utente o sviluppatore interessato sarà immediatamente attratto dall'annuncio. L'esempio di Fogel assomiglia a:

```
Oggetto: ANN: aub 1.0, un programma per assemblare file binari Usenet
```

Il resto della mail dovrebbe descrivere rapidamente e concisamente la funzionalità del programma, in non più di due paragrafi, e dovrebbe fornire collegamenti alla pagina web del progetto, e collegamenti diretti ai download per quelli che vogliono provarlo subito; questo formato funzionerà sia per la pubblicazione su Usenet che su mailing list.

Si dovranno ripetere questi annunci costantemente, negli stessi posti, per ogni rilascio successivo.

4.4.2. freshmeat.net

Citato in precedenza nel la Sezione 2.1.2.1. Nell'odierna comunità del software libero, annunciare il progetto su freshmeat è quasi più importante che annunciarlo sulle mailing list.

Si visiti il sito web freshmeat.net (http://freshmeat.net) o la loro pagina di inserimento dei progetti (http://freshmeat.net/add-project/) per pubblicare il proprio progetto sul loro sito e nella loro base dati: oltre ad un grande sito web, freshmeat fornisce una newsletter quotidiana che evidenzia tutti i rilasci del giorno, e raggiunge un pubblico enorme (personalmente la scorro ogni giorno in cerca di nuovi rilasci interessanti).

4.4.3. Mailing List di progetto

Se sono state create delle mailing list per il proprio progetto, le nuove versioni dovrebbero sempre essere annunciate su queste liste. Ho notato che per molti progetti gli utenti richiedono una mailing list di soli annunci, a bassissimo traffico, per essere avvisati quando vengono rilasciate nuove versioni; freshmeat.net ora consente agli utenti di sottoscrivere un particolare progetto, in modo da ricevere mail ogni volta che viene annunciata una nuova versione attraverso il loro sistema: è gratuito, e può sostituire una mailing list di soli annunci. Secondo me, male non fa.

Bibliografia

Libri cartacei

Karl Fogel, *Open Source Development with CVS*, Coriolis Open Press, 1999, 1-57610-490-7.

> La "Open Source Development with CVS" di Fogel è molto di più del suo sottotitolo. Per usare le parole dell'editore: "*Open Source Development with CVS* è uno dei primi libri disponibili che insegna lo sviluppo e l'implementazione di software Open Source". Include anche il miglior manuale introduttivo e testo di riferimento per CVS che abbia mai visto. Il libro era *così ben fatto* che mi ha spinto a scrivere questo HOWTO, perché il ruolo che esso cercava di esercitare mi era parso così importante e utile. Gli si dia un'occhiata, o lo si compri se possibile, se seriamente interessati a condurre un progetto di software libero.

Lawrence Lessig, *Code and Other Laws of Cyberspace*, Basic Books, 2000, 0-465-03913-8.

> Anche se tratta solo brevemente il software libero (e lo fa camminando in punta di piedi intorno alla questione software libero/open source, con l'uso della flaccida espressione "codice aperto", che solo un avvocato potrebbe coniare), il libro di Lessig è notevole. Scritto da un avvocato, parla

di come la regolamentazione su Internet non sia esercitata con le leggi, ma con il codice stesso, e di come la natura del codice determinerà la natura delle libertà future. Oltre ad essere una lettura veloce e godibile, racconta un po' di storie carine, e spiega quanto *c'e' bisogno* di software libero, con molta più intensità di qualsiasi cosa abbia letto, a parte "Right to Read" di RMS. (http://www.gnu.org/philosophy/right-to-read.html)

Eric Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly, 1999, 1-56592-724-9.

Anche se devo dire per onestà che non sono più quel fan di ESR che ero in passato, questo libro si è dimostrato insostituibile nel portarmi dove sono oggi. Il saggio che dà il titolo al libro fa un buon lavoro nel tratteggiare il processo del software libero, e fa un lavoro sbalorditivo nell'argomentare a favore dello sviluppo di software libero/open source come direzione da prendere per avere del software migliore. Il resto del libro contiene altri articoli di ESR, articoli che per la maggior parte sono pubblicati sul suo sito: eppure, è una bella cosa da possedere in versione cartacea, ed è una cosa che ogni appassionato di software libero/open source dovrebbe leggere.

Risorse accessibili via web

George N Dafermos, *Management and Virtual Decentralized Networks: The Linux Project (http://firstmonday.org/issues/issue6_11/dafermos/)*.

Poiché l'articolo ha un suo riassunto iniziale, ho pensato di inserirlo qui parola per parola:

> Questo articolo esamina il più recente dei paradigmi - l'Organizzazione a rete virtuale - e si chiede se lavoratori della conoscenza dispersi geograficamente possano collaborare per un progetto senza alcuna pianificazione centralizzata. La coordinazione, la gestione ed il ruolo della conoscenza emergono come le aree centrali di interesse. Sono stati scelti come caso di analisi il Linux Project e il suo modello di sviluppo, e sono stati identificati i fattori critici di successo per questa struttura organizzativa. Lo studio prosegue con la formulazione di una struttura di massima che può essere applicata a tutti i tipi di lavoro decentrato virtuale, e trae la conclusione che la creazione di valore viene massimizzata quando c'è intensa interazione, e condivisione di informazioni senza inibizioni, tra l'organizzazione e la comunità che la circonda. Perciò, il potenziale successo o fallimento di questo paradigma organizzativo dipende dal grado di dedizione e coinvolgimento della comunità circostante.

Questo articolo è stato segnalato a me in quanto autore di questo HOWTO, e ne sono rimasto molto impressionato. È stato scritto da uno studente specializzando in management, e penso che l'articolo abbia centrato il bersaglio nel valutare il progetto Linux come esempio di un nuovo paradigma di gestione, un paradigma in cui vi porrete al centro, nel vostro ruolo di manutentori di un progetto di software libero.

Come sviluppatore che cerca di controllare un'applicazione, e di condurla al successo nel mondo del software libero, non sono sicuro di quanto sia utile la discussione di Dafermos, tuttavia fornisce sicuramente una giustificazione teorica al mio HOWTO: la gestione di progetti di software libero *è* un animale diverso dalla gestione di progetti di software proprietario. Se si è interessati alle modalità concettuali e teoriche in cui la gestione di progetti di software libero differisce da altri tipi di gestione, questo è un ottimo articolo da leggere. Se questo HOWTO risponde a dei "come?", Dafermos risponde a dei "perché?" (ben più difficili da sostenere), e fa davvero un buon lavoro.

Richard Gabriel, *The Rise of "Worse is Better" (http://www.jwz.org/doc/worse-is-better.html)*.

Un articolo ben scritto, anche se penso che il titolo abbia confuso tanta gente, quanta il resto del saggio ne abbia aiutata. Offre una buona spiegazione di come progettare programmi che avranno successo, e che si conserveranno mantenibili crescendo.

Montey Manley, *Managing Projects the Open Source Way (http://news.linuxprogramming.com/news_story.php3?ltsn=2000-10-31-001-05-CD)*, Linux Programming (http://www.linuxprogramming.com), 31 ottobre 2000.

In uno dei migliori articoli sull'argomento che abbia letto, Monty ricapitola alcuni dei punti principali cui ho accennato, compresi: avvio di un progetto, collaudo, documentazione, organizzazione e leadership di un team, e diversi altri argomenti. Per quanto l'articolo sia più dogmatico di quanto io cerchi di essere, penso sia un articolo importante, e l'ho trovato molto utile nello scrivere il presente HOWTO; in tutti i punti in cui ho attinto qualcosa da questo articolo, ho cercato di citarlo.

Ho molti problemi in sospeso con questo pezzo; per avere un buon punto di vista critico si consiglia di leggerlo [KRAWITZ] contemporaneamente all'articolo di Monty.

Eric Steven Raymond, *Software Release Practice HOWTO (http://www.tldp.org/HOWTO/Software-Release-Practice-HOWTO/index.html)*.

Ad una prima occhiata, l'HOWTO delle pratiche di rilascio scritto da ESR sembra avere molto in comune con questo documento; ad un esame più dettagliato, le differenze diventano evidenti, anche se i due restano strettamente imparentati. Il suo documento, letto in congiunzione con questo, darà al lettore un buon quadro di come lavorare alla gestione di un progetto. L'HOWTO di ESR entra un po' più nei dettagli su come scrivere, e in quali linguaggi scrivere, e tende a dare istruzioni e liste di controllo più specifiche ("chiamate questo file così, non così"), mentre il presente HOWTO tratta le cose in modo più astratto. Ci sono diverse sezioni molto simili. Questo HOWTO è anche *molto* più breve.

La mia citazione preferita dal suo HOWTO è: “"Gestire un progetto bene, quando tutti i partecipanti sono volontari, presenta alcune sfide molto particolari. Questo è un argomento troppo vasto per essere trattato in un HOWTO.” Ah, davvero? Forse sto solo facendo un lavoro scadente.

Vivek Venugopalan, *CVS Best Practices (http://www.magic-cauldron.com/cm/cvs-bestpractices/index.html)*.

Venugopalan fornisce uno dei migliori saggi sull’uso efficace di CVS in cui mi sono imbattuto. È scritto per gente che abbia già una buona conoscenza di CVS. Nel capitolo sui rami descrive quando e come creare dei rami, ma non dà informazioni su quali comandi CVS bisogna usare per farlo. Questo va bene (sono stati scritti altri HOWTO su CVS, più tecnici), ma i novellini di CVS dovranno passare del tempo sul manuale di riferimento di Fogel, prima di poter trovare questo di qualche utilità.

Venugopalan produce liste di controllo di cose da fare prima, dopo, e in prossimità dei rilasci. Vale sicuramente una lettura, poiché molte delle sue idee risparmieranno un sacco di mal di testa agli sviluppatori, per un lungo periodo di tempo.

Articoli su Advogato

Stephen Hindle, *’Best Practices’ for Open Source? (http://www.advogato.org/article/262.html)*, Advogato (http://www.advogato.org), 21 marzo 2001.

Riferendosi principalmente alla pratica della programmazione (come fanno di solito la maggior parte degli articoli sull’ argomento), l’articolo parla un po’ della gestione dei progetti (“Usatela!”), e un pochino della comunicazione all’interno di un progetto di software libero.

Bram Cohen, *http://www.advogato.org/article/258.htmlHow to Write Maintainable Code*, Advogato (http://www.advogato.org), 15 marzo 2001.

Questo articolo accenna a quel dibattito sulla "scrittura di codice mantenibile" che ho cercato con ogni mezzo di evitare nel mio HOWTO. È uno dei migliori (e più diplomatici) articoli sull’argomento che abbia trovato.

Robert Krawitz, *Free Source Project Management (http://www.advogato.org/article/196.html)*, Advogato (http://www.advogato.org), 4 novembre 2000.

Questo articolo mi ha reso felice, perché ha affrontato molti dei problemi che avevo riguardo l'articolo di Monty su LinuxProgramming (http://www.linuxprogramming.com). L'autore sostiene che Monty suggerisce semplicemente l'applicazione di tecniche di gestione dei progetti antiquate, per software proprietario, invece di cercare di elaborare qualcosa di nuovo. Ho trovato questo articolo estremamente ben meditato, e lo ritengo una lettura essenziale per ogni gestore di progetti di software libero.

Lalo Martins, *Ask the Advogatos: why do Free Software projects fail? (http://www.advogato.org/article/128.html)*, Advogato (http://www.advogato.org), 20 luglio 2000.

Anche se l'articolo è poco più di una domanda, leggere le risposte a questa domanda fornite dai lettori di Advogato può essere d'aiuto. In molti modi, questo HOWTO costituisce la mia risposta alla domanda posta nell'articolo, ma ci sono altre risposte possibili, molte delle quali possono mettere in discussione quello che è scritto in questo HOWTO: vale la pena di dare un'occhiata.

David Burley, *In-Roads to Free Software Development (http://www.advogato.org/article/107.html)*, Advogato (http://www.advogato.org), 14 giugno 2000.

Questo documento è stato scritto come risposta a un'altro articolo su Advogato (http://www.advogato.org/article/72.html). Anche se non riguarda la conduzione di un progetto, descrive alcuni modi per iniziare con lo sviluppo di software libero senza iniziare un progetto. Penso sia un articolo importante. Se si è interessati ad impegnarsi nel software libero, questo articolo dimostra alcuni dei modi in cui si può farlo senza iniziare veramente un progetto (cosa che, come spero questo HOWTO abbia mostrato, non va presa alla leggera).

Jacob Moorman, *Importance of Non-Developer Supporters in Free Software (http://www.advogato.org/article/72.html)*, , Advogato (http://www.advogato.org), 16 aprile 2000.

Questo di Moorman è un articolo breve, ma apporta alcuni contributi utili. Il commento che ricorda agli sviluppatori di ringraziare i loro collaudatori ed utenti finali è preziosissimo, e spesso dimenticato.

Leslie Orchard, *On Naming an Open Source Project (http://www.advogato.org/article/67.html)*, Advogato (http://www.advogato.org), 12 aprile 2000.

Non avevo nemmeno una sezione, in questo HOWTO, su come scegliere il nome da dare al progetto, fino a che l'articolo di Leslie Orchard me lo ha ricordato: grazie a Leslie per aver scritto questo articolo!

David Allen, *Version Numbering Madness (http://www.advogato.org/article/40.html)*, Advogato (http://www.advogato.org), 28 febbraio 2000.

In questo articolo, David Allen affronta lo schema di numerazione delle versioni “Major.Minor.Patch” nella sua interezza; è bene leggerlo mentre si legge la Sezione 2.4. L’articolo mi è piaciuto, e inoltre descrive alcuni dei progetti che ho citato nella mia discussione della numerazione delle versioni.

A. GNU Free Documentation License

A.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. 3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- **C.** State on the Title Page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.

- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version .

A.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.7. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and dispbibute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document , on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the

Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation (http://www.gnu.org/fsf/fsf.html) may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/ (http://www.gnu.org/copyleft).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.