UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org

Edizione italiana a cura di ALSI e Tecnoteca
http://upgrade.tecnoteca.it

L'ingegneria del software libero: un campo da esplorare

di

Jesús M. González-Barahona e Gregorio Robles

(Traduzione italiana a cura di Raffaele Impagnatiello (ALSI – www.alsi.it) dell'articolo
Free Software Engineering: a Field to Explore
pubblicato sul Vol. IV,No. 4, Agosto 2003
della rivista online UPGrade, a cura del CEPIS)

Riassunto italiano: **La sfida del software libero o free software[1] non è quella di un nuovo competitore che, sulla base delle stesse regole, produce software in modo più veloce, più economico e con maggiore qualità. Il Software libero differisce dal software 'tradizionale' in molti aspetti fondamentali, a partire dalle ragioni filosofiche e dalle motivazioni, per continuare con le nuove regole economiche e di mercato, per poi terminare con un modo differente di produrre software. L'ingegneria del software non può ignorare questo fenomeno, e gli ultimi cinque anni circa hanno visto infatti molta ricerca su questi aspetti. Questo articolo esamina gli studi più significativi in questo campo ed i risultati che stanno producendo, con una visione tale da fornire al lettore lo stato dell'arte e le prospettive future di quello che abbiamo chiamato "Free Software Engineering".**

Parole chiave: Ingegneria del software, Ingegneria del software libero, Software libero.

1. Introduzione

Nonostante il "software libero" sia stato sviluppato da diversi decenni, solo negli ultimi anni si è cominciato a porre attenzione ai suoi modelli e processi di sviluppo secondo il punto di vista dell'Ingegneria del software. Così come non esiste un unico modello di sviluppo per il software proprietario, analogamente non ne esiste uno solo per il software libero [11], ma – ciò detto – esistono alcune caratteristiche interessanti che sono condivise da un gran numero di progetti da noi esaminati, caratteristiche che possono essere alla base del software libero.

Nel 1997 Eric S. Raymond ha pubblicato il suo primo articolo sull'argomento destinato ad essere ampiamente letto: *"La cattedrale ed il bazar"* [18], in cui descrive alcune caratteristiche dei modelli di sviluppo del software libero, sottolineando ciò che differenzia questi modelli dai modelli di sviluppo proprietari. Da allora quell'articolo è diventato uno dei più noti (e più criticati) nel mondo del software libero e, in un certo senso, è stato lo "sparo di partenza" per lo sviluppo dell'ingegneria del software libero.

[1] **Nota dell'editor di Upgrade**: la nostra politica editoriale è di continuare ad usare, nella versione inglese, il termine "free software", anche se siamo consapevoli che il temine "open source software" o semplicemente "open source" sembra vincere il confronto; sta diventando popolare anche "libre software" perché evita l'ambiguità della parola inglese "free". [NdT: in italiano è diffuso sia "software libero", sia "software open source ", sia "software a codice sorgente aperto"]

2. La cattedrale ed il bazar

Raymond opera una analogia tra il modo in cui erano costruite le cattedrali del medioevo ed il modo di produzione classico del software. Egli sostiene che in entrambi i casi c'è una chiara distribuzione di compiti e di ruoli, dove il progettista (Designer) è in cima a tutto e controlla il processo dall'inizio alla fine. La pianificazione è strettamente controllata in entrambi i casi, dando origine a processi chiaramente definiti, in cui – idealmente - ogni partecipante all'attività ha un ruolo molto limitato e specifico.

Raymond vede come analoghi al modello di costruzione delle cattedrali non solo i processi dei "pesi massimi" dell'industria del software (ad esempio il classico modello a cascata, i vari aspetti del Rational Unified Process, ecc.) ma anche alcuni progetti di software libero, come lo GNU (<http://www.gnu.org>/) e NetBSD (<http://www.NetBSD.org>). Secondo Raymond questi progetti sono centralizzati perché solo poche persone sono responsabili del progetto ed implementazione del software. Le attività che queste persone svolgono e i ruoli che ricoprono sono perfettamente definiti e per chiunque voglia inserirsi nella squadra di sviluppo, sarebbe necessaria l'assegnazione di una attività e di un ruolo in relazione alle necessità del progetto. Un'altra caratteristica è che i rilasci in questo tipo di progetti tendono ad essere distanziati nel tempo, secondo un rigido programma. Ciò comporta pochi rilasci del software, notevolmente distanziati tra loro, secondo fasi chiaramente definite dalla programmazione.

Il bazar è all'opposto del modello della cattedrale. Secondo Raymond, alcuni progetti di software libero, specialmente il kernel di Linux, sono stati sviluppati sulla falsariga di un bazar orientale. In esso non c'è una autorità di governo che controlla i processi che sono sviluppati e che pianifica accuratamente ogni passo. Inoltre i ruoli dei partecipanti possono cambiare costantemente (i venditori possono diventare clienti) senza direttive dall'esterno.

Forse la maggiore novità del libro di Raymond [18] consiste nella descrizione del processo che ha portato Linux ad essere una storia di successo nel mondo del software libero; si tratta di un insieme di "buoni metodi" che sfruttano appieno i vantaggi e le opportunità consentite dalla disponibilità del codice sorgente e dalla interattività dei sistemi e degli strumenti in rete.

Un progetto di software libero tende a nascere come risultato di una azione strettamente personale; comincia con uno sviluppatore che ammette di non essere in grado di risolvere completamente un problema. Lo sviluppatore deve almeno avere le conoscenze per cominciare a risolvere il problema. Dopo che lo sviluppatore è riuscito ad avere qualcosa di usabile, semplice e con qualche funzionalità (e se possibile ben progettato e ben codificato) la cosa migliore che può fare è condividere la soluzione con la comunità del software libero. Nel mondo del software libero questa prassi è nota come "rilascio anticipato", ed ha l'effetto di attirare l'attenzione di altre persone (generalmente sviluppatori) che hanno lo stesso problema e che sono interessati alla soluzione.

Uno dei principi base di questo modello di sviluppo è di considerare gli utenti come co-sviluppatori. Essi devono essere trattati bene, non solo perché possono fare pubblicità col sistema del passaparola, ma anche perché essi potranno eseguire il testing, che è una delle attività più dispendiose della produzione del software. Diversamente dal co-sviluppo, che non è facilmente scalabile, il debugging e il testing sono, per la loro stessa natura, altamente parallelizzabili. Infatti, è l'utente che prende il software e lo prova sulla sua macchina in condizioni determinate (un tipo di architettura, gli strumenti, ecc.) e si tratta di un'attività che, moltiplicata per il rilevante numero di architetture ed ambienti, richiederebbe un notevole impegno da parte del gruppo di sviluppo.

Se gli utenti sono considerati sviluppatori, c'è la possibilità che qualcuno di loro trovi un errore, lo corregga e mandi una patch agli sviluppatori del progetto, in modo da includere la soluzione nella versione successiva. Può anche succedere, ad esempio, che il problema sia approfondito e corretto da una persona diversa da chi ha segnalato per primo l'errore. In ogni caso, tutti questi scenari sono chiaramente molto favorevoli per lo sviluppo del software.

La situazione diventa ancora più efficace nel caso di rilasci frequenti; se si vede che i problemi sono affrontati rapidamente, c'è maggiore motivazione per trovare, segnalare e correggere gli errori. Ci sono anche vantaggi collaterali, come il fatto che l'integrazione è realizzata spesso (idealmente una volta al giorno) e quindi non è più necessaria la fase finale di integrazione dei moduli del prodotto. Questo principio del rilascio frequente permette una grande modularità [17] e massimizza l'effetto pubblicitario dei rilasci di una nuova versione del software. Come si può vedere in [5], la gestione dei rilasci nei progetti in grande scala tende a seguire un processo ben definito e in qualche modo complesso.

Per evitare che i rilasci frequenti respingano gli utenti che preferiscono la stabilità rispetto alla velocità di evoluzione del software, alcuni progetti di software libero hanno diversi filoni paralleli (*branches*) di sviluppo. Il caso più noto è il kernel di Linux, che ha un filone stabile – per chi apprezza l'affidabilità – ed un altro "instabile", orientato agli sviluppatori, con le più recenti innovazioni e funzionalità.

3. Il comando ed il processo decisionale nel bazar

Raymond presuppone che ogni progetto di software libero debba avere un "dittatore benevolo", un tipo di leader – che di solito è lo stesso fondatore del progetto -, responsabile della sua guida e che avrà sempre l'ultima parola nei momenti decisionali. Le abilità che questa persona deve avere sono principalmente la capacità di motivare e coordinare il progetto, capire gli utenti e i co-sviluppatori, raggiungere il consenso e integrare tutto ciò che può contribuire al progetto in qualche modo. I lettori noteranno che non abbiamo citato la competenza tecnica come uno dei più importanti requisiti, sebbene anche questa non farebbe male.

Al crescere delle dimensioni di alcuni progetti software libero e del numero di sviluppatori coinvolti, stanno cominciando ad emergere nuovi modi di organizzare i relativi processi decisionali. Il progetto di Linux, per esempio, ha una struttura gerarchica basata sulla delega di responsabilità da parte di Linus Torvalds, il "dittatore benevolo" (v. <http://www.linux.org/>). Ciò porta ad una situazione in cui ci sono parti di Linux con i loro "dittatori benevoli", anche se il loro potere è limitato dal fatto che Linus Torvalds ha sempre l'ultima parola. Questo caso è un chiaro esempio di come l'alto livello di modularità di un progetto di software libero ha dato origine ad uno specifico processo organizzativo e decisionale.

La Apache Foundation, <http://www.apache.org/>, dal canto suo è più meritocratica, poiché ha un comitato di direzione composto da chi ha dato significativi contributi al progetto. In realtà non è una meritocrazia in senso stretto, dove chi ha più contribuito ha più autorità su come procedere, ma il comitato di direzione è democraticamente eletto a scadenze molto frequenti dai membri della Apache Foundation (che sono responsabili della gestione di diversi progetti di software libero, come Apache, Jakarta, ecc.). Per diventare membro della Apache Foundation bisogna aver contribuito in modo continuo e rilevante ad uno o più progetti della fondazione. Lo stesso sistema è usato per altri progetti su larga scala, come il FreeBSD, <http://www.freebsd.org/>, e GNOME, <http://www.gnome.org/>.

Un altro interessante esempio di organizzazione formalizzata è il GCC Steering Commitee (Comitato direttivo del GCC), <http://gcc.gnu.org/steering.html>. È stato istituito nel 1998 per

impedire a chiunque di assumere il controllo del progetto GCC (GNU compiler collection, sistema di compilazione di GNU) ed è stato appoggiato, pochi mesi dopo, dalla FSF (Free Software Foundation, <http://www.fsf.org>), uno dei promotori del progetto GNU. In un certo senso è un comitato che si ricollega alla tradizione di un progetto similare (lo "egcs" che per qualche tempo è andato in parallelo al progetto GCC, e successivamente vi è confluito). La missione del Comitato è di assicurare che il progetto GCC rispetti la sua dichiarazione di intenti. I membri del Comitato sono eletti individualmente nell'ambito del progetto, così che in maggiore o minore misura rappresentino le diverse comunità che cooperano allo sviluppo del GCC (sviluppatori di strumenti di supporto per i linguaggi, sviluppatori del kernel, gruppi interessati alla programmazione dei sistemi embedded, ecc.).

L'incarico di direzione in un progetto di software libero non deve essere però scolpito nella pietra. Ci sono due motivi di base che possono obbligare il direttore di un progetto ad abbandonare il suo ruolo. Il primo motivo si ha nel caso di mancanza di interesse, tempo e motivazione nella conduzione del progetto. In questo caso l'interessato dovrebbe passare lo scettro del comando ad un altro sviluppatore. Recenti studi [7] mostrano che la direzione di un progetto tende a cambiare spesso, attraverso diverse generazioni di sviluppatori che conducono il progetto per un certo tempo. Il secondo motivo di abbandono dell'incarico di direzione è più problematico; è il proliferare del codice (*code forking*). Le licenze del software libero consentono a chiunque di prendere il codice sorgente, modificarlo e ridistribuirlo senza l'approvazione del "direttore" del progetto. Ciò non avviene frequentemente, eccetto che nei casi in cui si fa intenzionalmente per aggirare il responsabile del progetto (ed evitare il suo probabile veto alla modifica). Questo è un caso evidentemente simile ad un "colpo di stato", per quanto l'azione sia comunque legale e legittima. Il motivo per cui il responsabile del progetto cerca di mantenere la soddisfazione dei co-sviluppatori è per minimizzare il rischio della proliferazione del codice.

4. I processi del software libero

Sebbene il software libero non sia necessariamente collegato ad un particolare processo di sviluppo, c'è un ampio consenso sui più comuni processi usati. Non si può certo dire che ci siano progetti di software libero gestiti secondo il classico modello a cascata. Generalmente, i modelli di sviluppo dei progetti di software libero tendono ad essere più informali, principalmente perché i gruppi di sviluppo lavorano in modo volontario e senza corrispettivo finanziario, almeno non in forma diretta.

Il modo di raccolta dei requisiti nel mondo del software libero dipende sia dall'anzianità sia dalla dimensione del progetto. Nelle fasi iniziali, il fondatore del progetto e l'utente sono tendenzialmente la stessa persona. Successivamente, al crescere del progetto, la raccolta dei requisiti tende ad avvenire tramite mailing list e di solito si forma una chiara distinzione tra il gruppo di sviluppo – o almeno gli sviluppatori più attivi – e gli utenti. Nei progetti più grandi con molti utenti e sviluppatori, i requisiti sono raccolti con lo stesso strumento usato per la gestione degli errori. In questo caso si parla di attività invece che di errori, anche se il meccanismo usato per la loro gestione è lo stesso usato per la correzione degli errori (lo strumento li classifica per rilevanza, cause, ecc. e può essere usato per controllare se gli errori sono stati corretti o no). L'uso di questo strumento di pianificazione è una innovazione relativamente recente, ed è un chiaro segnale che nel mondo del software libero c'è stata una evoluzione da un'assenza di un qualche sistema di gestione verso un sistema centralizzato, sebbene con dei limiti. In ogni caso, non è usato normalmente il tipico documento di raccolta dei requisiti del modello a cascata.

Per quanto riguarda il disegno complessivo di sistema, di solito è documentato in modo completo solo nei grandi progetti. Per i casi di minore dimensione, probabilmente solo lo sviluppatore principale (o gli sviluppatori principali) è in possesso del progetto complessivo –

spesso solo nella propria testa – oppure lo diffonderà per le versioni successive del software. L'assenza di un qualche progetto di sistema non solo implica limiti nel senso di un possibile riuso dei moduli, ma è anche un grave handicap quando capiterà di aggiungere nuovi sviluppatori, nel senso che essi avranno una curva di apprendimento lenta e costosa. E non è neanche facile trovare il progetto di dettaglio, il che significa perdere molte occasioni di riuso del software.

Gli sviluppatori di software libero concentrano la maggior parte degli sforzi nella fase di implementazione, principalmente perché, agli occhi degli sviluppatori, l'implementazione è l'attività più piacevole. In quella fase il paradigma prevalente tende ad essere quello classico della prova e correzione finché si raggiunge il risultato voluto (dal punto di vista soggettivo del programmatore). Nel passato, i test di unità sono stati raramente inclusi nel codice, anche se ciò sarebbe stato di aiuto per altri sviluppatori che avevano necessità di modificare o aggiungere del codice. Nel caso di alcuni progetti su larga scala, ad esempio Mozilla, ci sono computer dedicati esclusivamente al download di archivi di deposito contenenti il codice più aggiornato, da usare per compilarlo per le diverse architetture [19]. Gli errori individuati sono diffusi nella mailing list degli sviluppatori.

Comunque, il testing automatizzato non è ancora consolidato. Normalmente sono gli stessi utenti che eseguono il testing, tramite una grande varietà di modalità d'uso e di architetture tra loro combinate. Questo ha il vantaggio per gli sviluppatori di parallelizzare il test ad un costo minimo. Lo svantaggio di questo modello consiste nel come organizzare le cose in modo che ci sia il riscontro degli utenti e che questo riscontro sia il più efficiente possibile. Rispetto alla manutenzione del software libero – nel senso di manutenzione delle versioni precedenti – questo aspetto può assumere più o meno importanza, in relazione al tipo di progetto. Ad esempio, in progetti che richiedono grande stabilità, come il kernel dei sistemi operativi, le versioni precedenti del progetto sono sottoposte a manutenzione, poiché il passaggio ad una nuova versione può essere traumatico. Ma, di solito, in molti progetti di software libero, se si trova un errore in una versione precedente, gli sviluppatori non tendono a correggerla, ma invece suggeriscono agli utenti di usare la versione più aggiornata, nella speranza che in essa l'errore sia stato già corretto come risultato della evoluzione del software.

5. Le critiche a "La cattedrale ed il bazar"

L'articolo di Raymond [18] manca di sistematicità e rigore, a causa della sua natura saggistica e per sua stessa ammissione non pienamente scientifica. La critica più frequente è riferita al fatto che descrive principalmente una sola esperienza – Linux – e cerca di estrapolare le conclusioni per tutti i progetti di software libero, mentre, come si può vedere in [14], l'esistenza di una estesa comunità come quella del kernel Linux è l'eccezione piuttosto che la regola.

Ancora più critici sono coloro che ritengono che Linux sia un esempio di sviluppo che segue il modello delle cattedrali. Essi sostengono che c'è, ovviamente, una guida intellettuale (o almeno qualcuno che è al vertice) e c'è un sistema gerarchico stabilito tramite delega di responsabilità fino ai muratori-programmatori. C'è anche una divisione del lavoro, anche se in modo implicito. Nel riferimento [2] la critica va oltre, sostenendo – non senza una dose di acrimonia ed arroganza nel ragionamento – che la metafora del bazar è contraddittoria.

Un altro dei punti più criticati di Raymond [18] è la sua affermazione che la legge di Brooks *"Aggiungere risorse umane ad un progetto software che è in ritardo comporta un ritardo maggiore"* [4] non sia valida nel caso del software libero. In [12] si può leggere che, nella sua vera essenza, stiamo parlando di ambienti differenti, e ciò che a prima vista appare come una incongruenza rispetto alla legge di Brooks è, ad uno studio più esauriente, solo un'illusione.

6. Studi quantitativi

Il software libero permette che gli sviluppatori usino appieno l'analisi quantitativa del codice e di tutti gli altri parametri implicati nella produzione, grazie alla loro accessibilità. Questo può essere di aiuto per alcuni settori dell'ingegneria del software – come per esempio l'ingegneria del software empirica – grazie all'esistenza di una notevole quantità di informazioni a cui si può accedere senza troppe interferenze con lo sviluppo del software libero. Gli autori di questo articolo sono convinti che questa visione può essere di grande aiuto nella analisi e nella comprensione dei meccanismi alla base della creazione del software libero (e del software in generale). Inoltre ci può anche condurre ad avere modelli previsionali del software e quindi riscontri in tempo reale.

L'idea che sta alla base è molto semplice: poiché abbiamo l'opportunità di studiare l'evoluzione di un gran numero di progetti software, facciamolo. Specialmente perché non solo è disponibile pubblicamente lo stato corrente dei progetti, ma anche tutti i loro sviluppi passati, il che vuol dire che tutte queste informazioni possono essere estratte, analizzate e raggruppate per costituire una base di conoscenze allo scopo di valutare lo "stato di salute" dei progetti, facilitare le decisioni e prevedere eventuali complicazioni.

La prima analisi quantitativa di una qualche rilevanza del software libero risale al 1998, anche se è stata pubblicata soltanto all'inizio del 2000 [10]. Il suo scopo era di ottenere una conoscenza empirica sulla partecipazione degli sviluppatori nel settore del software libero. A tal fine gli autori di [10] hanno eseguito uno studio statistico sulle dichiarazioni del nome dell'autore che gli sviluppatori scrivono nei file di intestazione del codice sorgente. Si è visto che la partecipazione al progetto segue la legge di Pareto: 80% del codice è opera del 20% degli sviluppatori più attivi, mente il restante 80% degli sviluppatori contribuisce all'altro 20% del codice. Diversi studi successivi hanno confermato ciò e ne hanno esteso la validità ad altre forme di partecipazione, ad esempio i messaggi alla mailing list, le segnalazioni di errore, ecc.. Lo strumento usato per condurre lo studio è stato diffuso nella modalità di "free license", offrendo perciò la possibilità di riprodurre i risultati e di eseguire nuove analisi.

In uno studio successivo, Koch [13] ha compiuto ulteriori passi ed ha anche analizzato le interazioni avvenute in un progetto di software libero. Nel suo caso l'origine delle informazioni è stata la mailing list e l'archivio delle versioni del progetto GNOME. L'aspetto più interessante dello studio di Koch è l'analisi economica. Egli si concentra sul controllo della validità dei modelli classici di previsione dei costi (i punti funzione, il COCOMO, ecc.) e mostra i problemi che derivano dalla loro applicazione, sebbene ammetta che i risultati ottenuti – usati con la dovuta cautela – sono solo in parte affidabili. Koch conclude che il software libero avrebbe bisogno di propri metodi e modelli di studio, poiché quelli attuali non sono ben adattabili. Comunque, sembrerebbe ovvio che l'opportunità di ottenere in maniera pubblica un gran numero di dati relativi allo sviluppo di software libero dovrebbe renderci ottimisti sulla disponibilità, in un prossimo futuro, di metodi e modelli appropriati. Lo studio di Koch può essere considerato come la prima analisi quantitativa completa, anche se carente di una chiara metodologia e soprattutto mancante di strumenti ad hoc in grado di verificare i risultati ed analizzare altri progetti.

Nel 2000 Mockus et al. [16] hanno presentato il primo studio di progetti di software libero che include la descrizione completa del processo di sviluppo e delle sue strutture organizzative, assieme ad attestazioni qualitative e quantitative. Per fare ciò, hanno usato il registro delle modifiche e le segnalazioni di errore, allo scopo di quantificare la partecipazione degli sviluppatori, le dimensioni del nucleo centrale del codice, gli autori del codice, la produttività, la densità dei difetti e gli intervalli di risoluzione degli errori. In un certo senso si tratta ancora di uno studio dell'ingegneria del software classica, eccettuato il fatto che la raccolta dei dati è avvenuta interamente tramite una ispezione semiautomatica dei dati che i progetti mettevano a disposizione

pubblicamente su Internet. Analogamente a [13], l'articolo di Mockus et al. non fornisce strumenti software o un procedimento automatico che permetta la riapplicazione del metodo da parte di altri gruppi di ricerca.

I riferimenti [23] e [24] trattano l'analisi quantitativa delle linee di codice e dei relativi linguaggi usati nella distribuzione Red Hat (<http://www.redhat.com/>). Barahona e altri hanno invece seguito un cammino analogo in una serie di articoli sulla distribuzione di Debian (<http://www.debian.org/>, v. [6] e [8]). Tutti i riferimenti forniscono una radiografia di queste due note distribuzioni di GNU/Linux, tramite i dati forniti da uno strumento che conta il numero "fisico" delle linee di codice dei programmi (le linee che non sono né bianche né commenti). Tralasciando il clamoroso dato delle linee di codice totali (l'ultima versione stabile – ad oggi – Debian 2.2, nota come Woody, ha più di 100 milioni di linee di codice) si può anche vedere la distribuzione del codice rispetto ai linguaggi di programmazione. La possibilità di studiare l'evoluzione nel tempo delle diverse versioni di Debian porta ad alcuni risultati interessanti [8], tra i quali c'è il fatto che la dimensione media dei singoli programmi è rimasta praticamente costante negli ultimi cinque anni, ciò vuol dire che la tendenza naturale a crescere dei programmi è stata neutralizzata dall'aggiunta di programmi più piccoli. Si è anche visto che l'importanza del linguaggio C, anche se esso rimane predominante, sta diminuendo nel tempo, mentre i linguaggi di script come Python, PHP, Perl e Java mostrano una crescita esplosiva. I classici linguaggi compilati (Pascal, Lisp, Ada, Modula, ...) sono chiaramente in declino. Infine, quegli articoli includono una sezione che mostra i risultati ottenuti se si fosse applicato il COCOMO - un modello classico di stima dello sforzo, risalente all'inizio degli anni 80 [3] – per ottenere una stima delle risorse, della durata e dei costi.

Sebbene si tratti di lavori pionieristici, la maggior parte degli studi presentati in questo paragrafo sono più o meno limitati ai progetti che essi analizzano. Il metodo usato è stato progettato per adattarsi ai progetti studiati, ed è in parte manuale e solo in pochi casi la parte automatizzata può essere usata in modo generalizzato in altri progetti di software libero. Questo implica che per studiare un nuovo progetto si richiederà un grande dispendio, poiché i metodi usati devono essere riadattati e si dovranno ripetere le attività manuali eseguite.

Per questi motivi, gli ultimi studi ([20], [9]) si sono concentrati anche nella creazione di una infrastruttura di analisi che integra diversi strumenti, in modo da automatizzare il più possibile il processo. Ci sono due ovvie ragioni per fare questo. La prima è che dopo aver investito tempo e risorse per creare uno strumento di analisi per un progetto – sottolineando la sua generalità – usarlo per altri progetti di software libero comporta uno sforzo minimo. In secondo luogo l'analisi tramite un insieme di strumenti che analizzano i progetti da diversi punti di vista – a volte complementari, a volte no – permette di avere una prospettiva più ampia del progetto. In [15] queste iniziative sono studiate in modo più approfondito.

7. Conclusioni e sviluppi futuri

Dopo questa breve ma intensa storia dell'ingegneria del software applicata allo sviluppo di software libero, si può dire che essa è ancora allo stadio iniziale. Ci sono diversi aspetti molto importanti che hanno bisogno di essere accuratamente studiati ed esaminati in dettaglio, allo scopo di derivare un modello che spieghi – almeno in parte – come creare il software libero. I temi che devono essere affrontati in un prossimo futuro sono due: la classificazione dei progetti di software libero e la creazione di un metodo basato il più possibile su strumenti automatici di analisi; l'uso della conoscenza così acquisita servirà a creare modelli di comprensione dello sviluppo del software libero, facilitando i processi decisionali.

Attualmente una delle principali debolezze è la mancanza di una rigorosa classificazione del software libero. Gli attuali criteri sono troppo approssimativi e semplicistici: si esaminano assieme dei progetti che hanno diversa organizzazione, diverse caratteristiche sia tecniche sia di altro tipo. Non può essere negata la tesi che Linux, con la sua estesa comunità di sviluppatori ed il grande numero di utenti, è di per sé differente e si comporta diversamente da altri progetti più limitati in termini di sviluppatori ed utenti. È chiaro che una classificazione più esaustiva permetterebbe il riuso delle esperienze acquisite in progetti simili (o meglio, con caratteristiche simili) e renderebbe più agevoli le previsioni sul progetto, sui rischi associati, ecc..

Il secondo tema che deve affrontare l'ingegneria del software libero, strettamente correlato al primo e alle attuali tendenze, è la creazione di una metodologia e di strumenti di supporto per il software libero. Una metodologia chiara e concisa permetterebbe di studiare tutti i progetti con lo stesso "metro di giudizio", di verificare il loro stato corrente, conoscere la loro evoluzione, e naturalmente di classificarli. Gli strumenti sono essenziali per affrontare il problema, perché dopo il loro sviluppo, permetteranno di analizzare migliaia di progetti ad un minimo costo aggiuntivo. Uno degli intenti della ingegneria del software libero è che, partendo da un ridotto numero di parametri che indicano dove trovare in rete le informazioni su un certo progetto (l'indirizzo dell'archivio delle versioni software, gli archivi della relativa mailing list, la collocazione del sistema per la gestione degli errori ed una breve rassegna sul sistema), si possa compiere una analisi esaustiva del progetto. I gestori del progetto avrebbero quasi a portata di mano una analisi completa, una sorta di "cartella clinica" che consentirebbe loro di valutare la salute del progetto e di avere indicazioni sugli aspetti da migliorare.

Nel momento in cui fossero disponibili metodi, classificazioni e modelli, si aprirebbero grandi possibilità grazie alla simulazione ed in particolare agli "agenti intelligenti". Tenendo presente che il nostro punto di partenza è un sistema di risaputa complessità, è di grande interesse poter creare modelli dinamici in cui rappresentare i diversi agenti che partecipano alla produzione di software. Siamo a conoscenza di diverse proposte per la simulazione del software libero, ma esse risultano abbastanza elementari ed incomplete. Ciò è dovuto in un certo modo al fatto che c'è una grande scarsità di conoscenze sulle interazioni che avvengono nella generazione del software libero. Se le informazioni sul progetto possono essere inquadrate accuratamente ed elaborate nel corso di tutta la sua storia, gli agenti intelligenti potrebbero essere cruciali per conoscere come il progetto evolverà. Ci sono alcune proposte sugli approcci a questo problema; attualmente una delle più avanzate è descritta in [1].

Per riassumere, in questo articolo abbiamo visto che l'ingegneria del software libero è un campo giovane ed ancora largamente inesplorato. I suoi primi passi stati fatti in scritti di tipo saggistico in cui – con una certa mancanza di rigore scientifico – era proposto un modello più efficiente di sviluppo; successivamente è stato compiuto un progresso graduale nello studio sistematico del software libero dal punto di vista dell'ingegneria. Attualmente, dopo diversi anni di report e di analisi quantitative e qualitative dei progetti di software libero, si sta compiendo un notevole sforzo per avere una infrastruttura globale che consenta, velocemente e in modo almeno parzialmente automatizzato, di classificare, analizzare e creare modelli dei progetti. Quando l'analisi dei progetti di software libero non sarà così dispendiosa in tempo e risorse come lo è oggi, è probabile che inizi una nuova fase dell'ingegneria del software libero, in cui altri tipi di tecniche si imporranno sulla scena, tecniche il cui scopo principale sarà la previsione delle evoluzioni del software e l'anticipazione delle eventuali complicazioni.

Riferimenti[2]

[2] NdT: un riferimento in italiano sui temi oggetto dell'articolo è il libro di Mariella Berra e Angelo Raffaele Meo *"Informatica solidale – storia e prospettive del software libero"*, Bollati Boringhieri, Torino, 2001

[1] Ioannis Antoniades, Ioannis Samoladas, Ioannis Stamelos, G. L. Bleris. Dynamical simulation models of the Open Source Development process, 2003, in corso di pubblicazione in Free/Open Source Software Development, published by Stefan Koch, Idea Inc, Vienna.

[2] Nikolai Bezroukov. A Second Look at the Cathedral and the Bazaar, December 1998. <http://www.firstmonday.dk/issues/issue4_12/bezroukov/index.html>.

[3] Barry W. Boehm, 1981. Software Engineering Economics, Prentice Hall.

[4] Frederick P. Brooks Jr., 1975. The Mythical Man-Month: Essays on Software Engineering, Addison Wesley.

[5] Justin R. Ehrenkrantz. Release Management Within Open Source Projects, May 2003. <http://opensource.ucc.ie/icse2003/3rd-WS-on-OSS-Engineering.pdf>.

[6] Jesús M. González Barahona, Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José Centeno González, Vicente Matellán Olivera. Counting potatoes. The size of Debian 2.2, Upgrade, vol. 2, issue 6, December 2001. <http://upgrade-cepis.org/issues/2001/6/up2-6Gonzalez.pdf>. Disponibile anche in <http://people.debian.org/~jgb/debian-counting/>.

[7] Jesús M. González Barahona, Gregorio Robles. Unmounting the code god assumption, May 2003, Proceedings of the Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering (Genova, Italia).
<http://libresoft.dat.escet.urjc.es/html/downloads/xp2003-barahona-robles.pdf>.

[8] Jesús M. González Barahona, Gregorio Robles, Miguel A. Ortuño Pérez, Luis Rodero Merino, José Centeno González, Vicente Matellán Olivera, Eva M. Castro Barbero, Pedro de las Heras Quirós. Anatomy of two GNU/Linux distributions, 2003", in corso di pubblicazione in "Free/Open Source Software Development, published by Stefan Koch, Idea Inc, Vienna.

[9] Daniel Germán, Audris Mockus. Automating the Measurement of Open Source Projects, May 2003. <http://opensource.ucc.ie/icse2003/3rd-WS-on-OSS-Engineering.pdf >.

[10] Rishab Aiyer Ghosh, Vipul Ved Prakash, The Orbiten Free Software Survey, May 2000. <http://www.firstmonday.dk/issues/issue5_7/ghosh/index.html>.

[11] Kieran Healy, Alan Schussman, The Ecology of Open Source Software Development, January 2003. <http://opensource.mit.edu/papers/healyschussman.pdf>.

[12] Paul Jones. Brooks' Law and open source: The more the merrier?, May 2000, <http://www-106.ibm.com/developerworks/opensource/library/os-merrier.html?dwzone=opensource>.

[13] Stefan Koch, Georg Schneider. Results from Software Engineering Research into Open Source Development Projects Using Public Data, 2000. <http://wwwai.wu-wien.ac.at/~koch/forschung/sw-eng/wp22.pdf >.

[14] Sandeep Krishnamurthy. Cave or Community? An Empirical Examination of 100 Mature Open Source Projects, May 2002. <http://opensource.mit.edu/papers/krishnamurthy.pdf>.

[15] Jesús M. González Barahona, Gregorio Robles Martínez. Libre Software Engineering. <http://libresoft.dat.escet.urjc.es/>.

[16] Audris Mockus, Roy T. Fielding, James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla, June 2000. <http://www.research.avayalabs.com/techreport/ALR-2002-003-paper.pdf>.

[17] Alessandro Narduzzo, Alessandro Rossi. Modularity in Action: GNU/Linux and Free/Open Source Software Development Model Unleashed, May 2003.
<http://opensource.mit.edu/papers/narduzzorossi.pdf >.

[18] Eric S. Raymond. The Cathedral and the Bazaar, Musings on Linux and Open Source by an Accidental Revolutionary, May 1997. <http://catb.org/~esr/writings/cathedral-bazaar/>.

[19] Christian Robottom Reis, Renata Pontin de Mattos Fortes. An Overview of the Software Engineering Process and Tools in the Mozilla Project, February 2002.
<http://opensource.mit.edu/papers/reismozilla.pdf>.

[20] Gregorio Robles, Jesús González Barahona, José Centeno González, Vicente Matellán Olivera, Luis Rodero Merino. Studying the evolution of libre software projects using publicly available data, May 2003, Proceedings of the 3rd Workshop on Open Source Software Engineering at the 25th International Conference on Software Engineering. <http://opensource.ucc.ie/icse2003/3rd-WS-on-OSS-Engineering.pdf >.

[21] Ilkka Tuomi. Internet, Innovation, and Open Source: Actors in the Network, 2001.
<http://www.firstmonday.dk/issues/issue6_1/tuomi/>.

[22] Paul Vixie. Software Engineering, 1999.
<http://www.oreilly.com/catalog/opensources/book/vixie.html>.

[23] David A. Wheeler. Estimating Linux's Size, July 2000. <http://www.dwheeler.com/sloc>.

[24] David A. Wheeler. More Than a Gigabuck: Estimating GNU/Linux's Size, June 2001.
<http://www.dwheeler.com/sloc>.

Note biografiche sugli autori

Jesús M. González-Barahona è professore presso l'Università Rey Juan Carlos di Madrid. Le sue ricerche riguardano il settore dei sistemi distribuiti e l'elaborazione paritaria (peer to peer) su larga scala. Si interessa anche dell'ingegneria del software libero. Ha cominciato a promuovere il software libero nel 1991. Collabora attualmente a diversi progetti di software libero (tra cui Debian) e con varie associazioni, come Hispalinux ed EuroLinux. Scrive su diversi media su temi collegati al software libero ed è consulente di aziende sulle relative strategie. È membro della associazione ATI, coordinatore della sezione sul software libero di Novática ed è stato editor di diverse monografie di Novática e di UPGrade.
<jgb@gysc.escet.urjc.es>

Gregorio Robles è professore presso l'Università Rey Juan Carlos di Madrid. Il suo lavoro di ricerca è incentrato sullo studio dello sviluppo di software libero da un punto di vista ingegneristico, specialmente sugli aspetti quantitativi. Ha sviluppato e/o partecipato al progetto di procedure per automatizzare l'analisi del software libero, ed anche di strumenti di sviluppo per il software libero. Ha partecipato alla ricerca FLOSS, finanziata dal progetto IST della Commissione Europea sul software libero.
<grex@gysc.escet.urjc.es>