

Progettazione del software

di Paul Vixie

La progettazione software è un campo più vasto che non lo "scrivere programmi". Tuttavia, in molti progetti Open Source, i programmi vengono semplicemente scritti e distribuiti. Ci sono chiari precedenti storici del fatto che il software non richiede necessariamente di essere progettato per poter essere usato largamente e con soddisfazione. In questo saggio vedremo alcuni elementi generali di progettazione software, quindi i corrispondenti di questi elementi nella comunità Open Source e infine le implicazioni delle differenze fra i due approcci.

Il processo di progettazione software

Gli elementi di un processo di progettazione software vengono generalmente enumerati come segue:

- Requisiti di marketing
- Progetto a livello di sistema
- Progetto dettagliato
- Implementazione
- Integrazione
- Testing sul campo
- Supporto

Nessuno degli elementi di questo processo dovrebbe avviarsi prima che i precedenti siano sostanzialmente completati e ogni qualvolta viene operata una modifica a qualche elemento, tutti gli elementi dipendenti dovrebbero essere revisionati o rifatti alla luce di quella modifica. È possibile che un dato modulo sia specificato e implementato prima che i moduli da questo dipendenti siano stati completamente specificati: questa pratica è nota come sviluppo avanzato o ricerca.

È assolutamente essenziale che ogni elemento della progettazione software includa diversi tipi di revisione: revisione dei pari, revisione del consulente/della dirigenza e revisione interdisciplinare.

Gli elementi della progettazione software (si tratti di documenti o di codice sorgente) devono avere numeri di versione e una storia verificabile. "Registrare" una modifica a un elemento dovrebbe richiedere una qualche forma di revisione, e la profondità della revisione dovrebbe corrispondere direttamente allo scopo della modifica.

Requisiti di marketing

Il primo passo di un processo di progettazione software è creare un documento che descriva il target di clientela e le ragioni per cui necessiti di questo prodotto, e procede quindi con la lista delle caratteristiche del prodotto indirizzate a queste necessità. Il Marketing Requirements Document (documento dei requisiti di marketing, MRD) è il terreno di battaglia su cui si decide la risposta alle domande "Che cosa costruire? Chi lo userà?".

In molti progetti finiti male il MRD era stato tramandato come fosse una tavoletta di pietra con iscrizioni dal marketing ai progettisti, che avrebbero quindi piagnucolato incessantemente contro le leggi della fisica e su come fosse loro impossibile costruire il prodotto data la mancanza di scorte di Kryptonite o di chissà cosa. Il MRD è uno sforzo congiunto, in cui i progettisti non solo revisionano, ma anche scrivono gran parte del testo.

Progetto a livello di sistema

Si tratta di una descrizione di alto livello del prodotto, in termini di "moduli" (a volte "programmi") e dell'interazione fra di essi. Gli scopi di questo documento sono in primo luogo acquisire la fiducia che il prodotto possa funzionare ed essere costruito, e in secondo luogo formare una base per la stima dell'ammontare totale di lavoro che la sua costruzione richiederà.

La progettazione a livello di sistema dovrebbe anche delineare il piano di testing a livello di sistema, in termini di necessità del cliente e se esse siano soddisfatte dal progetto di sistema proposto.

Progetto dettagliato

Il progetto dettagliato è il luogo dove ogni modulo richiamato nel documento di progetto a livello di sistema è

descritto nel dettaglio. L'interfaccia di ogni modulo (formato delle righe di comando, chiamate dell'API, strutture di dati visibili esternamente) dev'essere completamente definita a questo punto, così come le dipendenze fra i moduli. Due cose che evolveranno dal progetto dettagliato sono un grafico PERT o GANT che mostra come i lavori debbono essere fatti e in quale ordine lo debbano essere, e delle stime più accurate del tempo necessario a completare ogni modulo.

Ogni modulo ha bisogno di un piano di test di unità, che dica agli implementatori quali situazioni di test o quale tipo di situazioni di test essi debbano generare nei loro testing di unità per verificare la funzionalità. Si noti che ci sono test di unità aggiuntivi e non-funzionali che verranno discussi in seguito.

Implementazione

Ogni modulo descritto nel documento di progetto dettagliato dev'essere implementato. Questa fase include il minuscolo atto della codificazione o programmazione, che è il cuore e l'anima di ogni processo di progettazione software. È un peccato che questo minuscolo atto sia talvolta la sola parte della progettazione software che venga insegnata (o imparata), dal momento che è anche la sola parte della progettazione software che possa essere oggetto di autoapprendimento.

Un modulo può considerarsi implementato quando è stato creato, testato e usato con successo da alcuni altri moduli (o dal processo di testing a livello di sistema). La creazione di un modulo consiste nel classico ciclo scrittura-compilazione-ripetizione. Il testing dei moduli include il test funzionale a livello di unità e il test di regressione richiamati dal progetto dettagliato, e anche il testing di prestazione/stress e l'analisi di copertura del codice.

Integrazione

Quando tutti i moduli sono nominalmente completi si può operare l'integrazione a livello di sistema. È qui che tutti i moduli vengono trasferiti in un'unica raccolta di sorgente, quindi collegati e confezionati come sistema. L'integrazione si può operare incrementalmente, in parallelo con l'implementazione dei vari moduli, ma non può assolutamente accostarsi allo stato di finitezza finché tutti i moduli non siano sostanzialmente completi.

L'integrazione include lo sviluppo di un test a livello di sistema. Se il pacchetto creato deve potersi installare da sé (il che può significare tanto scompattare un archivio compresso quanto copiare dei file da un CD-ROM), allora dovrebbe esserci modo di farlo automaticamente, su sistemi di crash & burn dedicati o in ambienti di contenimento/simulazione.

Talvolta, nell'arena del middleware, il pacchetto non è che una raccolta di sorgente costruita, nel qual caso non esisterà strumento d'installazione e il testing di sistema dovrà essere fatto sulla raccolta così costruita.

Una volta che il sistema sia stato installato (se installabile), il processo automatizzato di testing a livello di sistema dovrebbe essere in grado di invocare ogni comando pubblico e di richiamare ogni punto pubblico di entrata, con ogni immaginabile combinazione di argomenti sensata. Se il sistema è in grado di creare un database di qualsiasi tipo, allora il testing automatizzato a livello di sistema dovrebbe crearne uno e quindi usare strumenti esterni (scritti separatamente) per verificare l'integrità del database. È possibile che i test di unità provvedano ad alcuni di questi bisogni, e tutti i test di unità dovrebbero essere eseguiti in sequenza durante i processi d'integrazione, costruzione e confezione.

Testing sul campo

Il testing sul campo s'inizia di solito internamente. Questo significa che impiegati dell'organizzazione che ha prodotto il software lo eseguiranno sui loro computer. Questo dovrebbe in ultima analisi comprendere tutti i sistemi "a livello di produzione": desktop, portatili, server. La dichiarazione di cui dovrete ambire nel momento in cui chiederete ai vostri clienti di usare un nuovo sistema software (o una nuova versione di un sistema software esistente) è: "lo usiamo noi stessi". Gli sviluppatori del software dovrebbero essere disponibili per l'assistenza tecnica diretta durante il testing sul campo interno.

Risulterà alla fine necessario eseguire il software esternamente, cioè sui computer dei clienti (o possibili tali). È consigliabile, per quest'esercizio, scegliere clienti "amichevoli", dal momento che è probabile che troveranno molti difetti, alcuni anche banali e ovvii, semplicemente perché i loro schemi d'uso e le loro abitudini saranno probabilmente diversi da quelli dei vostri utenti interni. Durante il testing sul campo esterno, gli sviluppatori del software dovrebbero restare presso il fronte di questo percorso di scalata.

I difetti riscontrati durante il testing sul campo debbono passare al vaglio dei capo-sviluppo e degli esperti del marketing tecnico, per determinare quali possano essere riparati nella documentazione, quali debbano essere riparati prima che la versione corrente venga rilasciata e quali possano essere riparati nella release successiva (oppure mai).

Supporto

I difetti del software, riscontrati durante il testing sul campo o dopo che il software sia stato distribuito, dovrebbero essere registrati in un sistema che tenga traccia delle revisioni (tracking system). Questi difetti dovrebbero infine venir assegnati a un progettista software che potrà proporre una modifica sia alla definizione che alla documentazione del sistema, o alla definizione di un modulo, o all'implementazione di un modulo. Queste modifiche dovrebbero includere aggiunte ai test di unità e/o a livello di sistema, in forma di un test di regressione per mostrare il difetto e quindi per mostrare che vi è stato posto rimedio (e per evitare che si ripresenti in seguito).

Proprio come l'MRD era uno sforzo congiunto di progettazione e marketing, così il supporto è uno sforzo congiunto fra progettazione e servizio clienti. I terreni di battaglia in quest'impresa sono le liste dei bug, la categorizzazione di particolari bug, il numero massimo di difetti critici ammesso in una release software distribuibile, eccetera.

Dettagli del testing

Analisi della copertura del codice

L'analisi della copertura del codice s'inizia con la strumentazione del codice di programma, talvolta a opera di un preprocessore, altre volte di un modificatore di codice d'oggetto, altre ancora usando una modalità speciale del compilatore o del linker, per tener traccia di tutti i possibili path del codice in un blocco di codice sorgente e per registrare, durante la sua esecuzione, quali siano stati percorsi.

Considerate il seguente, piuttosto tipico frammento di C:

```
1. if (read(s, buf, sizeof buf) == -1)
2. error++;
3. else
4. error = 0
```

Se la variabile d'errore non è stata inizializzata, allora il codice è difettoso, e se mai la riga 2 venga eseguita, il risultato del resto del programma sarà indefinito. La probabilità che un errore in read (e che esso restituisca un valore di ritorno di -1) occorra durante un testing normale è abbastanza bassa. Il modo per evitare eventi costosi in termini di supporto causati da questo tipo di bug è assicurarsi che i vostri test di unità tentino ogni possibile path del codice e che i risultati siano corretti in tutti i casi.

Ma aspettate: il meglio viene adesso. I path del codice sono combinatori. Nel nostro esempio, la variabile d'errore può essere stata inizializzata prima: per esempio, da un simile frammento di codice il cui predicato ("fallimento della chiamata di sistema") era falso (cioè, nessun errore era occorso). L'esempio che segue, di codice patentemente difettoso che non avrebbe comunque alcuna chance di passare una qualsiasi revisione di codice, mostra come sia facile che delle cose semplici diventino complicate:

```
1. if (connect(s, &sa, &sa_len) == -1)
2. error++;
3. else
4. error = 0;
5. if (read(s, buf, sizeof buf) == -1)
6. error++;
7. else
8. error = 0;
```

Ci sono ora quattro path di codice da testare:

1. righe 1-2-5-6
2. righe 1-2-5-8
3. righe 1-4-5-6
4. righe 1-4-5-8

È di norma impossibile testare ogni possibile path del codice: possono essercene centinaia perfino attraverso una piccola funzione di poche dozzine di righe. E d'altra parte, limitarsi ad accertare che i vostri test di unità siano in grado (magari in riprese successive) di usare ogni linea del codice, non è sufficiente. Questo tipo di analisi della copertura non si trova nella valigia degli attrezzi di qualunque progettista software nel campo: ed ecco perché il QA (Questions & Answers) è la sua specialità.

Test di regressione

Rimediare a un bug non basta. "Evidente all'ispezione" è spesso un'espressione di rinuncia, usata per coprire una più insidiosa: "scrivere il test per assegnare l'arma del delitto sarebbe difficile". D'accordo, ci sono molti bug che risultano evidenti all'ispezione, quali la divisione per la costante zero. Ma per riuscire a capire quali riparare, bisogna ispezionare il codice circostante per capire quello che l'autore (non voi, si spera) intendeva. Questo tipo di analisi dovrebbe venire documentata come parte del rimedio, o come parte dei commenti nel codice sorgente, o in entrambe le forme.

Nel caso più comune, il bug non risulta evidente all'ispezione e il rimedio si troverà in una parte del codice sorgente diversa da quella in cui il programma ha smarrito il core o comunque si è comportato irregolarmente. In questi casi, dovrebbe essere scritto un nuovo test che usi il path difettoso del codice (o lo stato di programma difettoso o quant'altro) e quindi il rimedio dovrebbe essere verificato su questo nuovo test di unità. Dopo la revisione e il check-in, il nuovo test di unità dovrebbe pure essere sottoposto a check-in, in modo che, se il medesimo bug è reintrodotta più tardi come effetto collaterale o modifica d'altro tipo, il reparto QA avrà qualche speranza di beccarlo prima che lo facciano i clienti.

Progettazione di software Open Source

Un progetto Open Source può includere ciascuno degli elementi succitati e, a essere onesti, qualcuno l'ha fatto. Le versioni commerciali di BSD, BIND e Sendmail sono tutti esempi del processo standard di progettazione software: ma non cominciarono in quel modo. Un processo di progettazione software con tutti i crismi è molto avido di risorse, e avviarne uno richiede normalmente degli investimenti che di solito implicano un qualche piano di rientro.

Un caso anche troppo comune di progetto Open Source è quello in cui i soggetti coinvolti si stanno divertendo e vogliono che il loro lavoro sia usato dal maggior numero possibile di persone: per questo lo danno via gratis e spesso senza porre alcuna restrizione alla ridistribuzione. Si tratta magari di gente che non ha accesso agli strumenti software cosiddetti "di livello commerciale" (analizzatori della copertura del codice, interpreti bounds-checking, verificatori dell'integrità della memoria). E le cose che più sembrano divertirli sono: programmare, confezionare ed evangelizzare: niente QA, niente MRD, di norma niente date di rilascio troppo vincolanti.

Rivisitiamo ciascuno degli elementi del processo di progettazione software e vediamo che cosa di solito avviene in un progetto Open Source non sovvenzionato. Fatto per amore.

Requisiti di marketing

Quelli dell'Open Source, di solito, tendono a costruirsi da sé gli strumenti che servono loro o che amerebbero avere. A volte questo capita in coincidenza di un giorno di lavoro, e spesso a opera di qualcuno il cui lavoro principale è qualcosa del tipo amministrazione di sistema piuttosto che progettazione software. Se, dopo molte iterazioni, un sistema software raggiunge una massa critica e assume una vita propria, verrà distribuito via Internet in file compressi e altri utenti cominceranno a richiedere nuove funzioni oppure, semplicemente, si siederanno al computer, se le scriveranno e le invieranno.

Il terreno di battaglia per un MRD Open Source è di solito una mailing list o un newsgroup, in cui utenti e sviluppatori, senza intermediari, si scambiano frizzi e lazzi. Il consenso è qualunque cosa con cui lo sviluppatore concordi o di cui si ricordi. La mancanza di consenso si traduce abbastanza spesso in biforcazioni del codice, dove altri sviluppatori cominciano a rilasciare le proprie versioni. L'analogo Open Source di un MRD può essere molto istruttivo ma ha margini pericolosamente taglienti e la risoluzione dei conflitti non è spesso possibile (o nemmeno ci si prova).

Progetto a livello di sistema

Di norma non c'è nessun progetto a livello di sistema in uno sforzo Open Source non sovvenzionato. O il progetto di sistema è implicito, uscendo armato come Minerva dalla testa di Zeus, o evolve nel tempo (come

il software stesso). Di solito, quando un software Open Source raggiunge la versione 2 o 3, esiste un progetto di sistema, anche se non si trova scritto da nessuna parte.

È qui, più che in ogni altra discrepanza dalle regole normali del percorso della progettazione software, che l'Open Source si guadagna la reputazione di cui gode, di leggera eccentricità. La mancanza di un MRD o anche di un QA come si deve può essere compensata dalla presenza di programmatori realmente in gamba (o di utenti realmente amichevoli), ma quando non c'è progetto di sistema (sia pure solo nella testa di qualcuno), la qualità del progetto si limiterà da se stessa.

Progetto dettagliato

Un'altra vittima della mancanza di fondi e della voglia di divertirsi è il progetto dettagliato. C'è bensì chi trova che un DDD sia una cosa divertente su cui lavorare, ma si tratta per lo più di gente che può divertirsi quanto vuole scrivendo DDD durante la sua giornata lavorativa. Il progetto dettagliato finisce con l'essere un effetto collaterale dell'implementazione. "Lo so, ho bisogno di un parser, allora me lo scrivo". Documentare l'API in forma di simboli esterni in file d'intestazione o di linee-guida è considerato facoltativo e può anche non venire in mente a nessuno se l'API non è destinata alla pubblicazione o a essere usata fuori dal progetto. È un peccato, perché una quantità di codice buono e altrimenti riusabile rimane in questo modo nascosto. Perfino moduli che non sono riusabili o strettamente legati al progetto in cui sono stati creati, e le cui API non sono parte delle caratteristiche distribuibili, dovrebbero avere linee-guida che spieghino quello che fanno e come richiamarle. È d'immenso aiuto a chiunque altro voglia potenziare il codice, dal momento che bisogna cominciare con il leggerlo e comprenderlo.

Implementazione

Ecco la parte divertente. L'implementazione è la cosa più amata dai programmatori, è quello che li tiene alzati fino alle ore piccole a smanettare quando potrebbero invece dormire. L'opportunità di scrivere codice è la motivazione primaria per quasi tutti gli sforzi di sviluppo software Open Source. Se ci si concentra su questo singolo aspetto della progettazione software a esclusione degli altri, c'è una grandissima libertà d'espressione.

I progetti Open Source sono il modo in cui la maggior parte dei programmatori sperimenta nuovi stili, siano stili d'indentazione o modi di nominare le variabili, "tentativi di risparmiare memoria" o "tentativi di salvare cicli della CPU" o quello che si voglia. E ci sono alcuni manufatti di grande bellezza che aspettano, racchiusi in archivi compressi un po' dovunque, in cui alcuni programmatori hanno provato uno stile per la prima volta e ha funzionato.

Un progetto Open Source non sovvenzionato può avere il rigore e la coerenza che vuole: gli utenti eseguiranno il codice se è funzionale; la maggior parte della gente se ne infischia se lo sviluppatore ha cambiato stile tre volte durante il processo d'implementazione. Agli sviluppatori, generalmente, importa, o se no, basta poco tempo perché imparino che importa. In situazioni come queste, le vecchie osservazioni di Larry Wall sulla programmazione come forma d'espressione artistica vanno esattamente a segno.

La principale differenza in un'implementazione Open Source non sovvenzionata è che la revisione è informale. Di solito non c'è consulente o pari che esamini il codice prima che sia rilasciato. E di solito non ci sono test di unità, di regressione né di nessun tipo.

Integrazione

L'integrazione di un progetto Open Source comporta come norma la scrittura di alcune pagine-guida, la verifica che funzioni su tutti i tipi di sistemi ai quali lo sviluppatore ha accesso, la ripulitura del Makefile per rimuovere detriti e corpi estranei che possano esservi insinuati durante la fase d'implementazione, scrivere un README, fare un archivio compresso, caricarlo su qualche sito per FTP anonimo e inviare una nota a qualche mailing list o newsgroup annunciandone disponibilità e ubicazione agli interessati.

Si noti che il newsgroup comp.sources.unix è stato riportato in vita da Rob Braun nel 1998 ed è un luogo molto adatto per annunci di pacchetti software Open Source nuovi o aggiornati. Funge anche da deposito/archivio.

È vero, non c'è testing a livello di sistema. Ma, del resto, non c'è nemmeno piano di test a livello di sistema né test di unità. Infatti, gli sforzi Open Source risultano alquanto sguarniti sull'intero fronte del testing (con eccezioni, come Perl e PostgreSQL). Questa lacuna nel testing pre-rilascio, tuttavia, non è una debolezza, come si spiega sotto.

Testing sul campo

Il software non sovvenzionato gode del miglior testing a livello di sistema in tutto il settore industriale, escludendo dal nostro paragone solo i test della NASA sui robot operanti nello spazio. La ragione è semplice: gli utenti tendono a essere molto più amichevoli quando non li si fa pagare nulla e gli utenti avanzati, i power user (spesso sviluppatori a loro volta) sono molto più collaborativi e prodighi di consigli quando possono leggere ed emendare il codice sorgente di un programma che eseguono.

L'essenza del testing sul campo è nella sua mancanza di rigore. Quello che la progettazione software si aspetta dai suoi testatori sul campo sono schemi d'uso che siano intrinsecamente imprevedibili al momento in cui il sistema viene progettato e costruito: in altre parole, l'esperienza di utenti reali nel mondo reale. I progetti Open Source non sovvenzionati, in questo campo, sono semplicemente imbattibili.

Un vantaggio aggiuntivo goduto dai progetti Open Source è la revisione da parte dei pari, cioè da parte di dozzine o centinaia di altri programmatori a caccia di bug tramite la lettura del codice sorgente piuttosto che limitandosi a far girare gli eseguibili pacchettizzati. Alcuni di questi lettori si concentreranno nella ricerca di difetti nella sicurezza, dei quali molti non verranno mai riportati (se non ad altri cracker), ma questo pericolo toglie poco al vantaggio complessivo di avere un numero incalcolabile di elementi esterni al progetto che leggono il codice sorgente. Questi elementi esterni possono davvero tenere uno sviluppatore Open Source all'erta come nessun consulente o dirigente riuscirebbe mai a fare.

Supporto

"Oh, mi spiace!" è quello che si dice a un utente che riporta un bug, oppure "Oh, mi dispiace, grazie!" se, insieme, l'utente manda anche una patch. "Ehi, per me funziona" è il modo in cui gli sviluppatori Open Source fanno il controllo dei bug. Sembra caotico? Lo è. La mancanza di supporto può trattenere alcuni utenti dal volere (o potere) usare programmi Open Source non sovvenzionati, ma crea anche opportunità per consulenti o distributori di software per vendere contratti di assistenza e/o versioni potenziate e/o commerciali.

Quando la comunità di produttori Unix incontrò per la prima volta un forte desiderio da parte dei suoi utenti di distribuire software Open Source preconfezionato con i loro sistemi-base, la prima reazione fu più o meno: "fate come vi pare, ma noi non li supporteremo". Il successo di aziende come Cygnus ha portato al riesame di questa posizione, ma lo scontro culturale ha radici piuttosto profonde. Le software house tradizionali, e fra queste i produttori Unix, non sono semplicemente in grado di stilare un piano o un budget per il costo di vendita di un'iniziativa di assistenza quando si danno modifiche non revisionate da parte di innumerevoli elementi esterni.

Talvolta la risposta è quella di riportare il software all'interno, farlo passare attraverso il normale processo QA comprendente il testing di unità e di sistema, l'analisi di copertura del codice e così via. Questo può comportare una progettazione inversa dell'MRD e del DDD per fornire un qualche contesto al QA (cioè, a quali funzionalità indirizzare i test). Altre volte, la risposta è riscrivere i termini dell'accordo di assistenza sostituendo ai "risultati garantiti" i "massimi sforzi possibili". In ultima analisi, lo spazio di mercato del supporto del software verrà occupato da chi saprà trarre impulso da tutti questi innumerevoli elementi esterni, dal momento che molti di loro sono gente valida che scrive software valido, e la cultura Open Source è più efficace, in molti casi, nel generare il livello di funzionalità che gli utenti effettivamente richiedono (ne faccia fede Linux contro Windows).

Conclusioni

La progettazione è un campo vecchio, e non importa se si tratti di software, hardware o ponti ferroviari: gli elementi del processo di progettazione sono sostanzialmente gli stessi:

- identificare un requisito e i suoi richiedenti;
- progettare una soluzione che venga incontro al requisito;
- modularizzare il progetto; pianificare l'implementazione;
- costruirlo; testarlo; consegnarlo; supportarlo.

Alcuni campi mettono più enfasi su certe fasi. Per esempio, i costruttori di ponti ferroviari di solito non si preoccupano tanto di un MRD, del processo di implementazione o del supporto, ma devono stare attentissimi al SDD e al DDD, e naturalmente al QA.

Il momento cruciale nella conversione di un "programmatore" in un "progettista software" è quell'istante in cui egli si accorge che la progettazione è un campo e che egli vi può entrare, ma che ciò gli richiederà un'attitudine mentale sostanzialmente diversa, e molto più lavoro. Gli sviluppatori Open Source possono andare avanti per anni con successo prima che la differenza fra la programmazione e la progettazione del software li raggiunga e gliela faccia pagare, e questo per il semplice fatto che i progetti Open Source soffrono in ritardo della mancanza di rigore progettistico.

Questo capitolo non ha dato che una panoramica molto ristretta della progettazione software, e (io spero) ha fornito qualche motivazione e un po' di contesto ai programmatori Open Source perché contemplino la possibilità di entrare in questo campo. Ricordate che il futuro è sempre un ibrido di tutto quanto di meglio è andato a comporre il passato e il presente. La progettazione software non è affare esclusivo di quelli che girano con il regolo e quattro penne nel taschino della camicia: è un campo ricco e vario, consistente di molte, ben sperimentate tecniche per costruire sistemi di alta qualità, in particolare sistemi di alta qualità che non sono gestibili secondo l'approccio del "geniale programmatore solitario" che è proprio dei progetti Open Source.