1 | Introduzione

Mentre nei corsi di programmazione solitamente c'è 1 programmatore che scrive algoritmi per una sola macchina, in un contesto più grande ci sono molti programmatori, suddivisi in team, che rispondono a vari committenti e realizzano sistemi e componenti che devono essere anche mantenuti.

È necessario quindi organizzare lo sviluppo in un gruppo di lavoro complesso.

2 | The mythical man-month

Nel suo libro "The mythical man-month", Brooks, racconta la sua esperienza nello sviluppo di OS/360. Un progetto pubblicato con un anno di ritardo e con sforamenti di budget altissimi. Nel libro si parla di come negli anni 70 era molto comune finire nella cosidetta tar-pit, sia con un team piccolo che uno enorme. Ci si ritrovano tutti senza capire la causa e molteplici fattori simultanei portano alla discesa. Tutti se ne possono accorgere ma non se ne capisce la causa.

2.1 La tar-pit

Si parte spiegando che per passare da un semplice programma a un prodotto veramente usabile ci vuole almeno lo sforzo originale moltiplicato per 9. Ogni programmatore si diverte creando la sua piccola opera, tuttavia i problemi arrivano quando deve interagire con altri, ordini dall'alto e altri programmatori. Mentre creare è divertente, trovare e risolvere i bug è solo lavoro. Il prodotto finale spesso non sembra bello come dovrebbe.

2.2 Legge di Brooks

Perché un progetto non rispetta i tempi prestabiliti?

- Le tecniche di stima sono sempre ottimiste rispetto alla realtà. In un progetto ci sono molte attività, spesso collegate una all'altra, che possono andare nel modo sbagliato, aumentando di molto la probabilità di fallire.
- Si confonde sforzo con progresso, personale e mesi non sono intercambiali (il mitico mese-uomo)
- Le stime sono incerte
- Il progresso è monitorato in modo superficiale
- Si tende ad aggredire i ritardi aggiungendo personale, che causa ancora più ritardo

Il mese-uomo non può essere usato come unità di misura perche personale e mesi non sono intercambiali quando si parla di programazione. Il mese-uomo è valido solo per le attività che non necessitano di comunicazione tra il personale (e.g. raccogliere pomodori). È impossibile applicarlo

ad attività che non possono essere partizionate su più persone e per attività che richiedono comunicazione complessa come lo sviluppo software può generare risultati anche peggiori. Ogni risorsa umana aggiunta deve essere formata e questa attività non può essere partizionata, lo sforzo di comunicazione cresce di n(n-1)/2 ogni volta che ne viene aggiunta una. Il risultato è che aggiungendo risorse si arriva a un punto dove il tempo totale viene allungato e non ridotto. Legge di Brooks: aggiungendo personale a un progetto di sviluppo software in ritardo, lo farà tardare ancora di più.

2.3 La sala operatoria

Tra i programmatori se ne trova sempre uno che è 10 volte più capace e performante degli altri. Si potrebbe pensare di prendere solo questi ultimi e metterli a lavorare in un team piccolo, tuttavia non sarebbe possibile per progetti più grandi.

La sala operatoria di Mills:

- Il chirurgo: molto esperto in più settori, definisce le specifiche, progetta il programma, lo scrive e lo documenta. Conduce i test e controlla le versioni del programma.
- Il copilota: ha meno esperienza del chirurgo ma potrebbe sostituirlo, agisce come consigliere del chirugo. Non scrive il codice ma lo conosce molto bene e fa da intermediatore con il team.
- L'amministratore: si occupa di personale, paga, spazi per conto del chirurgo.
- L'editore: si occupa di revisionare la documentazione scritta dal chirurgo.
- Due segretari: uno per l'amministratore e uno per l'editore, il primo si occupa anche della corrispondenza.
- L'addetto al programmma: si occupa di mantenere i record del team, gestisce i file, mantiene gli input e gli output.
- Il fabbro: si occupa di utities, librerie e degli strumenti che verranno usati per programmare, togliendo questo pensiero a chi programma.
- Il tester: pianifica i test e mette alla prova i programmi.
- L'avvocato del linguaggio: mentre il chirugo pensa alla rappresentazione del programma, ci sarà un altro che conosce ogni segreto e trucco del linguaggio di programmazione per risolvere al meglio un problema. Può essere condiviso da più team.

L'intero team della sala operatoria agisce e pensa come una sola persona.

2.4 La cattedrale

Non è possibile scalare la sala operatoria e mantenere l'integrità del sistema. Perché ciò accada, bisogna separare l'architettura dall'implementazione. L'architetto deve occuparsi solo dell'architettura e deve esserci una distinzione netta.

L'integrità concettuale in un progetto è fondamentale, bisogna avere un solo set di idee anziché idee distinte e scoordinate.

La semplicità arriva dall'integrità concettuale, questa arriva da una o poche menti. Tuttavia in un grande progetto deve esserci altra forza lavoro, e questa sarà nettamente separata da chi propone l'architettura. Una volta stabilito cosa si deve fare, si cerca di capire come farlo.

Problemi di questo approccio sono che:

- Le specifiche diventano troppo ricche di funzionalità e costose
- Gli architetti diventano troppo creativi e traciano tutta l'inventività degli implementatori
- Gli implementatori rimangono senza far niente per colpa del collo di bottiglia delle specifiche

2.5 L'effetto del secondo sistema

L'architetto deve considerare che l'implementazione potrebbe non andare come pensava e quindi potrebbe superare i costi previsti. L'architetto non ha l'ultima parola sull'implementazione ma deve esserci dialogo.

Di solito il primo lavoro di un architetto è preciso e pulito, non sapendo cosa sta facendo procede cauto e si limita, spesso mette da parte cose troppo ambizione per una seconda volta. Il primo lavoro finisce e l'architetto acquisisce confidenza.

Il secondo lavoro quindi diventa pieno di funzionalità ambiziose mentre si tenta di generalizzare quanto appreso dal lavoro precedente.

2.6 Passaparola

Per far si che un gruppo di 10 architetti mantenga l'integrità concettuale di un sistema costruito da 1000 persone è necessario che tutti si riescano a capire.

Il manuale descrive cosa vede e cosa può fare l'utente, non specifica cosa accade dietro e ciò che l'utente non vede. Il manuale deve essere chiaro, preciso e completo, ogni definizione deve contenere tutte le informazioni anche se ripetitive perché l'utente legge solo le parti che gli servono.

2.7 La torre di Babele

La torre di Babebe è fallita per mancanza di comunicazione e di conseguenza organizzazione.

Nello sviluppo di un grande progetto è necessaria comunicazione di tre tipi:

- Informale: interpretazione di cosa è scritto
- Riunioni: regolari riunioni per risolvere dubbi
- Workbook: scritto dall'inizio

Il workbook è la struttura che devono avere i documenti che il progetto produce. Deve essere aggiornato e mantenuto.

3 | I modelli a Bazaar

Eric Raymond nota il successo di progetti bottom-up come il kernel Linux, che definisce a Bazaar, l'opposto del modello a cattedrale.

Il kernel di Linux è un progetto di dimensioni enormi dove hanno collaborato molte persone, nonostante questo ha funzionato.

Il modo di lavorare di Linus Torvalds era molto distante dalla solita cattedrale del tempo, si rilascia spesso e in anticipo il software, si accettavano proposte da tutti, assomigliava molto a un bazaar.

Osservando Linux e sviluppando un software di mail, Eric, inizia a imparare delle lezioni sul perché il modello a bazaar funziona, nonostanza secondo Brooks non dovesse funzionare.

1. Ogni buon progetto software inizia risolvendo un fastidio personale di uno sviluppatore. A Eric serviva un client di mail POP con una determinata feature.

2. Un buon programmatore sa cosa scrivere, uno molto bravo sa cosa riscrivere e riutilizzare. L'autore decide di prendere client fatti da altri e aggiungere la sua feature.

3. Metti in conto di buttarne via uno, lo farai in ogni caso. Eric trova un software che si adatta meglio alla sua necessità e decide di abbandonare il lavoro precedente.

4. Con l'atteggiamento giusto, verrai trovato da problemi interessanti. Il secondo client a cui Eric a contribuito passa interamente nelle sue mani.

5. Quando perdi interesse per un programma, il tuo ultimo dovere è di consegnarlo a un successore competente. Perché il precedente proprietario aveva perso interesse.

6. Trattare i tuoi utenti come co-sviluppatori è il modo più veloce per migliorare il codice e trovare bug. Molti utenti di Linux erano bravi a programmare, se invogliati possono aiutare trovare bug molto più velocemente, se il codice è open source.

7. Rilascia in anticipo. Rilascia spesso. E ascolta i tuoi clienti. Rilasciare Linux in uno stato ancora pieno di bug ha funzionato perché gli utenti erano contenti e hanno aiutato a trovare i bug velocemente, al contrario di un modello a cattedrale dove sarebbero passati mesi.

8. Legge di Linus: Dato un numero sufficiente di occhi, tutti i bug vengono a galla. Con la prospettiva di contribuire a qualcosa di grande e ottenere gratificazioni molti sviluppatori hanno aiutato a risolvere i problemi di Linux. Linus era disposto a rilasciare versioni anche

con seri problemi. Fare debug è un'attività parallelizzabile. Per limitare l'impatto sugli utenti, Linux veniva rilasciato in versioni stabili e meno stabili. Una cosa importante da notare è che se entrambe le parti (utente e sviluppatore) conoscono il codice sottostante, è molto più semplice identificare il bug. La legge di Brooks viene resa inefficace dal fatto che non tutti devono comunicare con tutti in un modello a bazaar.

9. Strutture dati furbe e del codice stupido, funzionano molto meglio rispetto al contrario. Quando Eric ha riscritto il client di mail, in molti casi riorganizzato la struttura per capirla meglio.

10. Se tratti i tuoi beta-tester come se fossero la tua risorsa più preziosa, risponderanno diventando la tua risorsa più preziosa. Eric ha coinvolto molto i suoi utenti durante lo sviluppo del software, mandando aggiornamenti costanti e incoraggiando a partecipare.

11. La cosa migliore dopo avere buone idee è riconoscere le buone idee dei tuoi utenti. A volte quest'ultima è meglio. Un utente ha suggerito la soluzione a un problema con SMTP, che ha portato alla soluzione migliore.

12. Spesso, le soluzioni più sorprendenti e innovative derivano dal rendersi conto che la tua concezione del problema era sbagliata. Un problema più grande è stato risolto ripendando interamente il programma e rimuovendone una grande parte, nonostante Eric abbia esitato all'inizio.

13. La perfezione (nel design) non si raggiunge quando non c'è più niente da aggiungere, ma quando non c'è più niente da togliere. È stato possibile migliorare il programma rimuovendo le parti inutili.

14. Qualsiasi strumento dovrebbe essere utile se usato come previsto, ma uno strumento davvero eccezionale si presta ad usi che non ti saresti mai aspettato. In seguito, applicare un protocollo nel software, ha portato a risolvere problemi precedenti in modo molto più semplice.

15. Quando scrivi un software gateway di qualsiasi tipo, sforzati di disturbare il meno possibile il flusso di dati e non buttare mai via le informazioni a meno che il destinatario non ti costringa a farlo. Fortunatamente Eric ha lasciato intoccato l'ultimo bit nei caratteri ASCII, in seguito ha potuto adottare MIME con molta facilità.

16. Quando il linguaggio non è Turing-completo, lo zucchero sintattico può essere tuo amico. Eric non è un fan dei linguaggi che assomigliano troppo all'inglese.

17. Un sistema di sicurezza è sicuro finché è segreto. Attenzione agli pseudo-segreti. L'autore ha evitato di implementare una funzione suggerita da un utente perché avrebbe dato un falso senso di sicurezza.

18. Per risolvere un problema interessante, inizia trovando un problema che ti interessa. Il modello a bazaar funziona se il progetto ha una base di utenti abbastanza interessata, non è possibile

far partire un progetto a bazaar da zero. Gli sviluppatori non devono avere ego in quello che scrivono e devono essere aperti a commenti e modifiche.

19. A condizione che il coordinatore dello sviluppo disponga di un mezzo di comunicazione come Internet, e sappia come guidare senza coercizione, molte teste sono inevitabilmente migliori di una. Linus è riuscito a creare un sistema dove molti sviluppatori hanno potuto esprimere il loro ego rimanendo altruisti.

4 | Il Progetto Debian

4.1 Introduzione

Il progetto Debian mira a produrre un sistema software con migliaia di componenti che girano su undici diverse architetture hardware, con tre diversi kernel del sistema operativo.

Le applicazioni vengono eseguite in ambienti molto complessi in cui un sistema operativo kernel, alcuni driver di periferica, librerie di sistema e grafiche, servizi comuni, ecc. coesistono per fornire la piattaforma software su cui gli utenti possono usufruirne. Per questo motivo, fin dall'inizio i sostenitori del software libero hanno puntato a produrre sistemi completi, liberi ed eseguibili su qualsiasi macchina. Storicamente, uno dei principali ostacoli a questo obiettivo è stata l'indisponibilità di un kernel di sistema operativo. Tuttavia, negli ultimi quindici anni diversi kernel (ad es, Linux, FreeBSD, GNU/Hurd, ecc.) sono stati resi disponibili alla comunità open source ed è stato possibile costruire piattaforme informatiche interamente libere che integrano utilità di base e sofisticati software applicativi.

La costruzione di una distribuzione coerente richiede un grande sforzo di coordinamento e di lavoro cooperativo, per cui la metafora del bazar sembra del tutto inappropriata. Tuttavia, nel caso di Debian, anche la metafora della cattedrale è inadeguata, dal momento che non sono presenti architetti principali e il lavoro è svolto interamente su base volontaria. Pertanto, io suggerisco la nuova metafora del kibbutz[1] per una comunità cooperativa di volontari che condividono un obiettivo comune. Le proprietà che caratterizzano tale comunità sono:

- le persone si uniscono alla comunità su base volontaria e non si aspettano di essere pagati per il loro lavoro;
- i membri concordano su un obiettivo finale ambizioso;
- i membri condividono una coscienza civile e accettano che il loro lavoro sia regolato da regole esplicite stabilite dalla democrazia diretta.

[1] Il kibbutz, talvolta kibbuz o kibuz in italiano è una forma associativa volontaria di lavoratori dello stato di Israele, basata su regole rigidamente egualitarie e sul concetto di proprietà collettiva.

4.2 La struttura e gli obiettivi di Debian

Il progetto Debian2 è stato avviato da Ian Murdock il 16 agosto 1993 per "mettere insieme, con altrettanta cura" una distribuzione di software Linux lavorando "apertamente nello spirito di Linux e GNU". Fin dall'inizio tutti i membri di Debian erano volontari e tuttora non sono pagati da Debian per svolgere il loro lavoro nel progetto. Tuttavia, dal novembre 1994 al novembre 1995 Debian è stata sponsorizzata dalla Free Software e Debian ha motivato la creazione di "Software in the Public Interest", un'organizzazione senza scopo di lucro che fornisce un meccanismo attraverso il quale il Progetto Debian può accettare donazioni. Una distribuzione Linux mette insieme pezzi di software che sono in genere costruiti da persone estranee ai distributori stessi. Il progetto Debian richiede che software incluso in un sistema Debian sia conforme alle "Linee guida Debian per il software libero": in pratica, il software deve essere rilasciato con una licenza open source che permetta la libertà di utilizzo, distribuzione e modifica senza discriminazioni e restrizioni. Il primo rilascio al grande pubblico di un sistema Debian GNU/Linux è stato fatto nel gennaio 1994 (ver. 0.91). Conteneva poche centinaia di programmi e fu messo insieme da una dozzina di sviluppatori. Oggi (gennaio 2004) il progetto conta 1268 membri distribuiti in tutto il mondo e gestisce più di 13.000 pacchetti binari (corrispondenti a più di 8.000 sorgenti), portati su 11 architetture diverse. Esistono almeno tre sistemi Debian completi: oltre a quello principale basato su Linux, ce n'è uno basato sul kernel BSD e un altro basato sul kernel GNU Hurd. Il fondatore di Debian, Ian Murdock, non lavora più attivamente al progetto dal 1996.

4.2.1 La struttura Debian

Tutti possono richiedere di diventare membri Debian. Per accettati nel progetto si deve dimostrare di avere delle competenze di base necessarie per la gestione dei pacchetti software e la comprensione delle "Debian Free Software Guidelines" e del "Debian Social Contract "3 .

Entrando a far parte del gruppo si dà il proprio consenso a contribuire al progetto secondo la Costituzione Debian. La Costituzione definisce un'organizzazione di cui fanno farte

- un Project Leader (DL);
- un Segretario del progetto (DS);
- un Comitato tecnico (TC);
- singoli sviluppatori.

Il DL, il DS e il presidente del TC devono essere tre persone diverse. Il lavoro è volontario: nessuno è obbligato a fare nulla e ognuno sceglie liberamente di essere assegnato a un compito che ritiene utile o interessante. Ogni anno viene nominato un nuovo DL con un'elezione generale che coinvolge tutti i sviluppatori che votano con il meccanismo di Condorcet. Il DL può prendere decisioni urgenti e nomina il DS e, insieme al TC, rinnova i membri del TC stesso. La CT è composta da un massimo di 8 membri, con un minimo di 4 persone, e decide le politiche tecniche. I

singoli sviluppatori possono annullare qualsiasi decisioni del DL e del TC emettendo una risoluzione generale a una maggioranza qualificata. Il DS è nominato dal DL e dal precedente DS ogni anno ed è DS è incaricato di gestire elezioni e di altre chiamate al voto e giudica eventuali controversie sull'interpretazione della Costituzione. Le proprietà e le attività finanziarie sono gestite da "Software in the Public Interest, Inc" (SPI), in cui ogni membro di Debian può essere un membro votante. La conseguenza di questa struttura organizzativa è che nessun singolo individuo può assumere il controllo personale del progetto. Ancora meglio, "ogni singolo sviluppatore può prendere qualsiasi decisione tecnica o non tecnica per quanto riguarda al proprio lavoro". Tuttavia, poiché la coerenza del prodotto finale è uno degli obiettivi su cui i membri sono d'accordo, questa libertà assoluta deve essere temperata da un coordinamento, ottenuto attraverso una serie di politiche che, dopo la discussione nelle mailing list (la maggior parte delle quali sono pubbliche) dove anche i non sviluppatori possono contribuire alla discussione, ma devono incontrare un alto grado di consenso generale per non essere scavalcate da risoluzioni generali.

Per studiare l'organizzazione di Debian è importante prendere in considerazione una serie di attori che interagiscono con la galassia Debian, senza essere necessariamente membri del progetto, ma possono influenzare il lavoro di Debian. Prima di tutto ci sono gli Autori a monte. Contribuiscono a Debian scrivendo software open source. In teoria non potrebbero nemmeno conoscere Debian. In pratica sono spesso in comunicazione diretta con gli sviluppatori Debian, perché all'interno di Debian viene fatto molto lavoro per scoprire e correggere i bug. Pertanto, è comune che gli sviluppatori Debian (d'ora in poi, DD) inoltrino agli autori upstream bug, patch, suggerimenti, richieste di nuove funzionalità, ecc. In secondo luogo, ci sono gli utenti, la soddisfazione dei quali è ovviamente una forza importante che indirettamente guida il progetto. Inoltre, i sistemi Debian forniscono una sofisticata infrastruttura per il tracciamento dei bug (Debian Bug System, DBTS), che è la via principale attraverso cui gli utenti possono segnalare problemi e proporre miglioramenti. Una terza categoria che vale la pena di menzionare per la sua crescente importanza è quella composta da persone che usano i sistemi Debian per costruire le proprie distribuzioni specializzate.

4.3 Il processo di sviluppo di Debian

Una distribuzione di un sistema Debian è composta da un programma di installazione e da un insieme di pacchetti software. Il programma di installazione è in grado di configurare il sistema su un gran numero di configurazioni hardware: questo rende l'installazione un'operazione piuttosto complessa. I pacchetti software possono essere recuperati da una serie di CD, da un disco rigido locale o dalla rete. Tutto lo sforzo di sviluppo di Debian si concentra sulla produzione di pacchetti. Un pacchetto è l'unità minima che può essere installato o rimosso da un sistema. Di conseguenza, ogni DD è responsabile di uno o più pacchetti e si dice che sia il manutentore di quel pacchetto. Quando un manutentore ha messo insieme il suo pacchetto, viene caricato in un repository pubblico dove gli utenti Debian di tutto il mondo possono provare a installarlo sui loro sistemi. Poiché finora il pacchetto è stato testato solo sulla macchina del DD, il suo stato è da considerarsi alpha-testing e il repository è chiamato distribuzione instabile. Tuttavia, nonostante il nome spaventoso, un numero

considerevole di utenti (e praticamente tutti i DD) prova i pacchetti dalla distribuzione instabile, quindi il test è abbastanza significativo. Se un pacchetto vive nella distribuzione instabile per dieci giorni senza che venga segnalato alcun bug critico viene automaticamente caricato in un altro repository, corrispondente a uno stato di beta-testing. Questa repository è nota come distribuzione di test. Quando tutti i bug critici per il rilascio sono corretti, viene rilasciata al pubblico una nuova distribuzione stabile. La distribuzione stabile è quella che viene distribuita ufficialmente da Debian e i pacchetti inclusi sono aggiornati solo per correggere le vulnerabilità di sicurezza.

L'obiettivo di ottenere una distribuzione coerente in cui tutti i programmi possano interagire senza problemi è molto complesso. Il problema sembra senza soluzione se una distribuzione viene ottenuta aggregando migliaia di pacchetti prodotti da centinaia di sviluppatori su decine di sistemi diversi. Ciononostante, i sistemi Debian sono stati in grado di ottenere un'ottima soddisfazione complessiva da parte degli utenti, come testimoniano i numerosi premi vinti nel 2003. In effetti, lo sforzo principale dei DD è diretto a garantire che i loro pacchetti siano pienamente conformi alle politiche Debian. Le politiche sono fondamentali nell'approccio di Debian alla distribuzione del software. La libertà dei DD è illimitata, a patto che rispettino le politiche concordate collettivamente. Le politiche sono spesso basate su standard internazionali o comunitari (ad esempio, il Filesystem Hierarchy Standard) e riguardano tutte le questioni globali che influiscono sulla coerenza di un sistema: ad esempio, distribuzione delle librerie, variabili d'ambiente, servizi condivisi, linguaggi di scripting. A volte assumono la forma di principi generali ("Gli script del manutentore devono essere idempotenti"), ma più spesso asseriscono a qualche proprietà verificabile automaticamente del pacchetto installato. Per i sottosistemi complessi esistono sottopolitiche speciali: per esempio, l'editor estensibile Emacs ha le proprie politiche che riducono i possibili conflitti tra l'enorme numero di pacchetti specifici per Emacs provenienti da fonti diverse. L'applicazione delle politiche è perseguita a diversi livelli, al fine di sfruttare la convalida incrociata per ridurre al minimo l'inconsistenza dei pacchetti incoerenti:

- durante l'assemblaggio dei pacchetti: la maggior parte delle politiche sono associate ad uno strumento (chiamato collettivamente "debhelper") che ne garantisce la corretta applicazione;

- durante il test del pacchetto: esistono diversi strumenti per verificare la conformità alle politiche prima di caricare il pacchetto nel repository pubblico. Il più importante è lintian, uno script che analizza un pacchetto alla ricerca di una trentina di categorie di violazioni delle policy. Inoltre, quando un pacchetto viene caricato su un repository pubblico viene rifiutato se i controlli falliscono;

- durante la distribuzione del pacchetto: ogni utente che rileva una incoerenza può segnalare un bug con una procedura automatica (reportbug).

Poiché le politiche sono pubbliche e disponibili su ogni sistema Debian, anche le violazioni non dannose possono essere scoperte (e le segnalazioni di bug dimostrano che spesso lo sono) e notificate ai DD. Ogni pacchetto presuppone implicitamente un ambiente di lavoro che gli fornisca alcuni servizi. I DD dovrebbero esplicitare queste ipotesi definendo un insieme di dipendenze per ogni

pacchetto. La ricchezza del linguaggio delle dipendenze di Debian permette di regolare sistemi installati: se A dipende da B, B deve essere installato per installare A; se A suggerisce B, B può migliorare le funzionalità di A, ma A può essere utilizzato nella maggior parte dei casi anche senza B. Inoltre, due pacchetti possono entrare in conflitto, un pacchetto può sostituirne un altro e un pacchetto A può fornire le funzionalità di B. Quest'ultima relazione rende utile l'esistenza di pacchetti virtuali (ad es. applicazione generica di mailer) che possono essere richiesti da altri. Per favorire il riutilizzo ed evitare le duplicazioni, Debian promuove il micro-packaging. Quindi è comune che da un singolo pacchetto sorgente vengano generati diversi pacchetti binari. Grazie a queste relazioni di dipendenza, l'installazione di una nuova applicazione su un sistema funzionante può essere indolore come digitare un comando "apt-get install applicazione": tutti i pacchetti necessari vengono recuperati da un repository pubblico (possibilmente su un insieme di CD), installati e configurati.

4.4 Personalizzazione e manutenzione di un sistema Debian

Uno dei valori aggiunti dei sistemi open source è che possono essere personalizzati per soddisfare al meglio le esigenze degli utenti. Tuttavia, la personalizzazione è anche rischiosa. Un sistema altamente personalizzato può essere molto difficile da mantenere in sincrono con lo sviluppo mainstream dell'open source. Supponiamo per esempio che un utente voglia usare un programma javalocal piuttosto che il programma java fornito dal pacchetto "java" di Debian. Se l'utente sovrascrive /usr/bin/java con java-local, il sistema di gestione dei pacchetti non verrà a conoscenza di questo cambiamento scartando la personalizzazione durante gli aggiornamenti. Per questo motivo, Debian introduce il concetto di diversificazione dei pacchetti, grazie al quale gli utenti possono mantenere le loro versioni deviate di programmi, pur godendo degli aggiornamenti tradizionali. Per esempio, lanciando il comando dpkg-divert –divert /usr/bin/java.debian /usr/bin/java tutte le future installazioni di pacchetto Debian "java" scriveranno il file /usr/bin/java in /usr/bin/java.debian. Inoltre, diversi programmi alternativi equivalenti possono essere installati in un sistema e si può utilizzare una semplice infrastruttura per mantenere un nome generico collegato all'alternativa preferita (ad esempio, xwww-browser può puntare a galeon, anche se sono installati sia mozilla che galeon sono entrambi installati). Queste possibilità rendono i sistemi Debian come base di partenza per distribuzioni specializzate: esempi di successo sono la distribuzione Knoppix (che funziona interamente da CD) e la distribuzione Familiar (destinata a essere eseguita su PDA): pur essendo molto diverse tra loro, condividono tutte la stessa infrastruttura di pacchetti e continuano a riutilizzare il lavoro quotidiano dei DD nonostante le loro personalizzazioni. Un altro problema che a volte ostacola gli utenti nell'aggiornamento dei loro sistemi personalizzati, è che le opzioni di configurazione possono essere scartate dalla nuova versione delle applicazioni. In linea di massima, la configurazione di un'applicazione è un processo in tre fasi. Le opzioni principali sono impostate a livello di sistema al momento dell'installazione dell'applicazione. Altre opzioni meno importanti sono più frequentemente modificate. Le opzioni utente sono modificate dagli utenti stessi e le impostazioni sono memorizzate nelle loro directory. Nei sistemi Debian, la conservazione delle opzioni principali tra

un aggiornamento e l'altro è ottenuta sfruttando il database debconf. Quando una nuova applicazione viene installata per la prima volta, vengono poste alcune domande all'utente. Le risposte fornite dall'utente vengono memorizzate in questo database e quando verrà installata una nuova versione dell'applicazione, vengono presentate all'utente solo le nuove opzioni. Le scelte dell'utente vengono mantenute per le opzioni invariate e lo script di installazione ha il compito di tradurle nella nuova sintassi dei file di configurazione. Inoltre, ogni volta che un aggiornamento influisce su un file di configurazione, viene emesso un avviso che chiede quale versione l'utente vuole mantenere e, se i file sono file di testo leggibili dall'uomo, come è comune nel mondo Unix, le differenze possono essere unite.

Un altro approccio che vale la pena menzionare è quello che può essere chiamato manutenzione dei pacchetti orientata agli aspetti. In qualsiasi sistema sufficientemente complesso, ci sono problemi trasversali all'intero sistema e che non possono essere facilmente impacchettati in un modulo isolato. La soluzione Debian a questo problema segue un approccio orientato agli aspetti: particolari eventi speciali del ciclo di vita del pacchetto sono esposti ad altri pacchetti e possono, inconsapevolmente dal punto di vista degli altri pacchetti, introdurre azioni che saranno eseguite quando questi eventi si verificheranno.