Indice

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.

5 | Metodologia Agile

La "metodologia agile" è un approccio allo sviluppo del software basato sulla distribuzione continua di software efficienti creati in modo rapido e iterativo.

Benché l'espressione faccia riferimento specifico a un determinato sviluppo del software, questa metodologia non prevede regole da rispettare tassativamente nell'ambito dello sviluppo del software. Si tratta piuttosto di un tipo di approccio alla collaborazione e ai flussi di lavoro fondato su una serie di valori in grado di guidare il nostro modo di procedere.

In pratica, le metodologie di sviluppo software agile consistono nel rilasciare rapidamente modifiche al software in piccole porzioni con l'obiettivo di migliorare la soddisfazione dei clienti. Le metodologie utilizzano approcci flessibili e il lavoro di gruppo per concentrarsi sul miglioramento continuo. Con l'adozione dello sviluppo agile i vari team, costituiti da pochi sviluppatori ciascuno, si organizzano in autonomia e collaborano direttamente con i rappresentanti aziendali tramite incontri periodici durante l'intero ciclo di vita dello sviluppo del software. La metodologia agile consente di adottare un approccio più leggero alla stesura della documentazione software e di integrare le modifiche in qualsiasi fase del ciclo di vita, anziché ostacolarle.

5.1 I valori Agili

La metodologia agile, come la conosciamo oggi, risale al 2001. Nel tentativo di trovare un'alternativa al project management "a cascata", che struttura un progetto software in una serie di sequenze lineari, un gruppo di sviluppatori software ha delineato un nuovo metodo redigendo il Manifesto per lo sviluppo agile di software. In questo documento i programmatori propongono un nuovo approccio allo sviluppo del software descrivendo 4 aspetti chiave che a loro parere devono avere la priorità su altri. I team di sviluppo software devono tenere conto di:

- **Individui e interazioni** rispetto a processi e strumenti
- **Un software efficiente** rispetto a una documentazione esaustiva
- **La collaborazione con il cliente** rispetto alla negoziazione dei contratti
- **La preparazione ad affrontare il cambiamento** rispetto all'esecuzione di un piano

Gli autori chiariscono comunque che tutti gli aspetti indicati sopra sono importanti, ognuno a suo modo. Sostengono tuttavia che dare la priorità agli aspetti a sinistra (in grassetto) rispetto

a quelli a destra può portare a risultati migliori nello sviluppo del prodotto. Il Manifesto non determina una serie di pratiche da seguire, ma è da intendere come una guida verso un nuovo modo di pensare allo sviluppo del software.

Il Manifesto ha portato all'ottenimento di svariati risultati pratici. Ad esempio, anziché sviluppare software in modo sequenziale, ovvero da una fase all'altra, che è il modo in cui il metodo a cascata garantisce la qualità del prodotto, un metodo agile può promuovere lo sviluppo e il test come processi continui e simultanei. In sostanza, lo sviluppo a cascata presuppone il completamento di una fase prima di poter passare a quella successiva, mentre il modello agile supporta più sequenze contemporanee alla volta.

5.2 Le origini della metodologia Agile

L'approccio agile al lavoro è stato concepito per far fronte ai limiti percepiti del metodo a cascata, che trae le sue sorgenti dal metodo manifatturiero basato sulla catena di montaggio del 1913 di Henry Ford, applicato più tardi allo sviluppo di software. Dalla sua nascita nel 2001, lo sviluppo agile si è fatto strada nel settore dei software e del project management subendo molti cambiamenti. L'impulso che ha spinto alcuni sviluppatori a introdurre una metodologia agile era legato alla mancata capacità dei cicli di produzione e dei metodi di collaborazione a cascata di portare ai risultati attesi. Questo problema ha iniziato a diffondersi nei primi anni '90, quando potevano trascorrere anni tra l'identificazione di un'esigenza aziendale e l'erogazione di un'applicazione efficiente volta a risolverla. Quegli anni erano caratterizzati da esigenze di business in costante mutamento, e molti dei progetti software venivano addirittura annullati prima che fossero erogati. Questa perdita di tempo e di risorse ha portato molti sviluppatori software a cercare un'alternativa. Messe in difficoltà dalle continue innovazioni, le organizzazioni hanno iniziato ad adottare sempre più strategie di trasformazione digitale per tenere il passo con un settore in continua evoluzione. E qui entra in gioco lo sviluppo agile del software. La metodologia agile è alla base di molti flussi di lavoro digitale odierni. Il cloud computing, con la sua infrastruttura IT flessibile e scalabile, è cresciuto di pari passo con le richieste dello sviluppo agile di software. Lo sviluppo cloud native integra la nozione di software "agile" per ottenere una serie di servizi interconnessi e scalabili volti a soddisfare le esigenze aziendali. Il concetto di DevOps rompe il confine tra sviluppo del software e operazioni. L'approccio SRE (Site Reliability Site Engineering) è un'implementazione di DevOps che prevede l'uso del software come strumento per la gestione dei sistemi e l'automazione di tali operazioni. Tenendo conto della continua evoluzione del software, i metodi CI/CD mettono a disposizione degli sviluppatori gli strumenti necessari per accelerare i tempi di deployment di un nuovo codice. Ne deduciamo quindi che il concetto di "metodologia agile" incorpora un'idea agile in grado di rispondere alle esigenze dei suoi clienti (ovvero, gli sviluppatori software) nel tempo. È importante tenere conto di questo aspetto quando si analizzano alcuni framework agili, che hanno nomi diversi e spesso variano da un'implementazione all'altra.

5.3 Framework Agili

Framework agili per lo sviluppo di software come Scrum, Kanban o Extreme Programming (XP) costituiscono la base dei più famosi processi di sviluppo di software quali DevOps e l'integrazione e il deployment continui (CI/CD). Scrum è forse il framework agile più conosciuto in adozione al giorno d'oggi, sarà discusso nella prossima sezione. Altri framework agili, ad esempio Kanban, precedono il Manifesto agile. Ma questi framework sono considerati agili perché promuovono i valori delineati nel Manifesto agile. I framework agili e gli approcci verso la metodologia agile sono troppi per poterli menzionare tutti in questa sede.

5.4 Scrum

Scrum è un approccio basato sulla teoria del controllo empirico dei processi: le decisioni vengono prese sulla base dell'esperienza (empirismo). Tutti gli aspetti del lavoro devono essere visibili ai responsabili del risultato finale (trasparenza). Per rendere trasparenti questi elementi, il Team Scrum ispeziona di frequente il prodotto mentre lo sta sviluppando (ispezione). Così il processo e il prodotto possono essere adattati immediatamente nel caso di nuove esigenze o di condizioni mutate del mercato (adattamento). Come tutte le metodologie Agile, Scrum si basa sulla divisione del progetto in più fasi, chiamate Sprint. Ad ogni Sprint il gruppo di lavoro presenta nuove funzionalità, operative e implementabili, che vengono fatte valutare al cliente. Si configura così un sistema iterativo che consente di incrementare poco alla volta, ma molto di frequente, le funzionalità del progetto. E allo stesso tempo si può verificare costantemente l'andamento complessivo e la soddisfazione del cliente su quanto già prodotto.

5.4.1 Ruoli nello Scrum

Sono 3 i ruoli che vengono individuati nella metodologia Scrum: Product Owner, Scrum Master e Team di Sviluppo.

- **Il Product Owner** definisce il lavoro da svolgere e l'ordine con cui viene completato. Raccoglie la voce degli stakeholder (clienti, management e chiunque abbia un interesse nel prodotto), le necessità dell'utente finale, i requisiti del mercato e sulla base di questi elementi stabilisce le priorità di sviluppo per il Team Scrum.
- **Lo Scrum Master** è il responsabile del processo, e un leader a servizio (servant-leader) dello Scrum Team. Conoscitore esperto della metodologia Scrum, sa come applicarla e si assicura che il Team comprenda e segua le regole che la caratterizzano, perché il progetto abbia successo.
- **Il Team di Sviluppo** è la squadra di lavoro, composta da 3 a 9 persone. Anche lo Scrum Master può far parte del Team di Sviluppo. Chi concretamente porta a termine gli Sprint e

fornisce le funzionalità da implementare è questo insieme coordinato di persone, autogestito e cross-funzionale.

Autogestito perché decide in autonomia come sviluppare le funzionalità individuate dal Product Owner. Cross-funzionale perché contiene al proprio interno tutte le competenze e le funzioni necessarie per portare a termine i task.

5.4.2 Eventi nello Scrum

Uno sprint è composto dai seguenti eventi:

- **Sprint** è il cuore del framework Scrum. Della durata compresa tra 2 e 4 settimane, ogni Sprint è nella pratica un mini progetto, il cui scopo è presentare un Incremento che risponde ai requisiti di "Fatto", utilizzabile e potenzialmente rilasciabile.
- **Sprint Planning** è l'inizio di ogni Sprint: una riunione in cui il Team Scrum concorda l'obiettivo dello Sprint (lo Sprint Goal)
- **Daily Scrum** è un evento giornaliero di 15 minuti in cui il Team di Sviluppo programma la giornata di lavoro.
- Alla fine di ogni Sprint è prevista una riunione, la **Sprint Review**, in cui lo Scrum Team discute insieme agli stakeholder l'Incremento realizzato. La riunione serve per dimostrare quanto è stato svolto, e per iniziare a raccogliere gli input per le successive riunioni di Sprint Planning.
- Dopo la Sprint Review si apre la **Sprint Retrospective**, un momento formale in cui il Team Scrum analizza lo Sprint concluso, identifica le cose che sono andate bene, individua i miglioramenti che possono essere fatti e pianifica il modo in cui implementarli nello Sprint successivo. Dura al massimo tre ore.

5.5 Tecniche di lavoro

- **Pair-programming** - si programma in coppie con una sola tastiera
  - Obbliga a rendere espliciti i ragionamenti
  - Aiuta a mantenere il focus sull'obiettivo
  - Diffonde la conoscenza totale della codebase (riducendo anche i rischi in caso di assenza di un collaboratore)
- **Codice condiviso** - tutto il team 'e responsabile di tutto il codice e pu'o modificarlo a piacimento

- **Refactoring** - Sono piccole trasformazioni che non cambiano la semantica del codice, spesso attuabili automaticamente con un editor "consapevole" del linguaggio di programmazione.
- **Test-driven Development** Il test di unit'a viene scritto prima dell'unit'a stessa, servendo come "specifica" (ma senza la necessaria generalità!)

6 | Software Configuration Managment

Nell'ingegneria del software, la gestione della configurazione del software (SCM o S/W CM) è il compito di tracciare e controllare le modifiche al software, parte del più ampio campo interdisciplinare della gestione della configurazione. Le pratiche di SCM includono il controllo delle revisioni e la creazione di linee di base. Se qualcosa va storto, l'SCM può determinare cosa è stato modificato e chi lo ha fatto. Se una configurazione funziona bene, l'SCM può determinare come replicarla su molti host. Di cosa si occupano SCM:

- Gli artifacts sono file o, più raramente, directory;
- L'SCM permette di tracciare/controllare le revisioni degli artifact e le versioni delle risultanti configurazioni;
- A volte forniscono supporto per la generazione del prodotto a partire da una ben determinata configurazione.

Il meccanismo di base per controllare l'evoluzione delle revisioni è che ogni cambiamento è regolato da:

- **check-out** - dichiara la volontà di cambiare un determinato artifact;
- **check-in** - detto anche commit, dichiara la volontà di registrare un determinato change-set.

Quando il repository è condiviso da un gruppo di lavoro, nasce il problema di gestirne l'accesso concorrente:

- modello pessimistico - il sistema garantisce l'accesso agli artifact in mutua esclusione attivando un lock al check-out
- modelo ottimistico - il modello si disinteressa del problema e fornisce supporto per le attività di merge di change-set paralleli potenzialmente conflittuali.

6.1 Merge

Merge è un'oprazione delicata che generalmente viene trattata con strategie diverse:

- Lavoro parallelo su artifact diversi;

- Lavoro parallelo sullo stesso artifact con hunk diversi;
- Lavoro parallelo sullo stesso artifact con hunk uguali.

6.1.1 Terminologia per i merge

La terminologia più usata fa riferimento alla coppia di programmi POSIX diff e patch:

- diff calcola la differenza tra due revisioni R0 e R1 per righe, cercando di minimizzare il numero di inserimenti e cancellazioni;
- patch è il programma che permette si applicare il diff solo a R0 per ottenere R1.

6.1.2 3-way merge

Qando, come nel caso di lavoro parallelo sullo stesso artifact, le due rvisioni hanno un antenato comune si può facilitare il lavoro di merge. Siamo A' e A" due revisioni, con antenato comune A:

- hunk uguale nelle tre revisioni - inalterato;
- hunk uguale in due delle tre revisioni:
  - A' e A" uguali - merge A'
  - A e A' uguali - merge A"
  - A e A" uguali - merge A'
- hunk diverso nelle tre revisioni deve essere valutato a mano.

6.2 Git

Git è un DVCS (Distributed Version Control System), cioè sistema software per il controllo di versione distribuito, realizzato nel 2005 da Linus Torvalds.

6.2.1 Nozioni base di Git

- **file system** - directory in cui si trovano i file;
- **blob** - il contenuto di un file. Gli oggetti blob non hanno nome, data, ora, né altri metadati. Git memorizza ogni revisione di un file come un oggetto blob distinto.
- **tree** - l'equivalente di una directory: contiene una lista di nomi di file, ognuno con alcuni bit di tipo e il nome di un oggetto blob o albero che è il file, il link simbolico, o il contenuto di directory. Questo oggetto descrive un'istantanea dell'albero dei sorgenti.

- **commit** (revisione) - collega i tree in una cronologia. Contiene il nome di un oggetto albero (della directory dei sorgenti di livello più alto), data e ora, un messaggio di archiviazione (log message), e i nomi di zero o più oggetti di commit genitori. Le relazioni tra i blob si possono trovare esaminando gli oggetti albero e gli oggetti commit.
- **index** - uno strato intermedio che serve da punto di collegamento fra il database di oggetti e l'albero di lavoro.
- **database** - ha una struttura semplice. L'oggetto viene messo in una directory che corrisponde ai primi due caratteri del suo codice hash; Il resto del codice costituisce il nome del file che contiene tale oggetto.
- **tag** (etichetta) - un contenitore che contiene riferimenti a un altro oggetto, può tenere metadati aggiuntivi riferiti a un altro oggetto. Il suo uso più comune è memorizzare una firma digitale di un oggetto commit corrispondente a un particolare rilascio dei dati gestiti da Git.

I blob sono file binari "non parlanti" che conservano informazioni eterogenee, quali: file testuali o binari, immagini, codice sorgente, archivi. Qualsiasi tipo di file è compresso in un file binario prima di essere salvato in repository Git. Git ne calcola l'hash con l'algoritmo SHA-1.

Come riferirsi ad un commit:

- Nome - hash del commit (anche le prime lettere);
- Nickname - un tag, un branch, HEAD;
- Parentela:
  - n̂ - ennesimo padre;
  - ñ ennesimo antenato.
- Data.

6.2.2 Git workflows

L'utilizzo dei branch in git è fondamentale, si può procedere a piacere o utilizzare delle tecniche già collaudate.

GitFlow

Un modello di branching diventato molto popolare si chiama GitFlow, è stato concepito nel 2010 e si adatta bene a software che vengono rilasciati a versioni, mentre l'autore ne sconsiglia l'uso nelle webapp.

Fare branch e merge è molto semplice in git, per questo possono essere utilizzati senza paura.

In GitFlow si inizia creando una repo centralizzata, chiamata 'origin', che contiene la 'truth'. Ogni sviluppatore effettua 'pull' e 'push' con 'origin', in più è possibile effettuare dei 'fetch' tra diversi sviluppatori e creare dei sotto-team.

I due branch più importanti sono 'master' e 'develop'. Hanno vita infinita rispetto agli altri branch. Consideriamo 'master' il branch principale, dove il codice sorgente riflette sempre uno stato 'production-ready' (che può andare nelle mani dei clienti). Il branch 'develop' agisce come branch di integrazione, è dove si trovano i delivered per la prossima release, è la base per creare le nightly builds.

Quando il codice in 'develop' raggiunge un punto stabile, viene effettuato il branch su 'master' e viene assegnato un tag col numero di versione. Per definizione i commit in 'master' sono rilasci di nuove versioni, può essere presente un 'hook' che rilascia la versione ai clienti.

Ci sono poi dei branch con vita limitata:

- Feature branches
- Release branches
- Hotfix branches

Questi branch seguono delle regole specifiche. È importante notare che questa è una convenzione, tutti i branch in git sono uguali.

I feature branches:

- Devono partire da 'develop'
- Devono effettuare merge in 'develop'
- Si possono chiamare in qualunque modo tranne che 'master', 'develop', 'release-*', 'hotfix-*'

In questo branch si sviluppa una nuova feature, non si sa ancora in che release verrà aggiunta, entrerà nella prossima release quando verrà effettuato il merge, oppure verrà eliminata. Il merge deve essere effettuato in modalità no-ff in modo da mantenere la storia del feature branch.

I release branches:

- Devono partire da 'develop'
- Devono effettuare merge in 'develop' e 'master'
- Devono chiamarsi 'release-*'

Si utilizza questo branch per gli ultimi preparativi per rilasciare la versione, come ad esempio bugfix o sistemare i metadati. Una volta creato il release branch, il branch 'develop' può ricevere le feature della prossima release. Il branch viene creato quando tutte le feature su 'develop' sono abbastanza stabili e pronte al rilascio. Il nome del branch decide il numero di versione. Quando la release è pronta si fa merge su 'master', e si assegna il tag della versione, poi si fa merge su 'develop' per portare i bugfix effettuati sul release branch.

Gli hotfix branches:

- Devono partire da 'master'
- Devono effettuare merge in 'develop' e 'master'
- Devono chiamarsi 'hotfix-*'

Si comportano come i release branches, ma vengono creati quando una situazione di emergenza in produzione richiede un fix immediato. Se esiste un release branch, al posto di fare merge in 'develop' bisogna fare merge in quello di release. L'hotfix arriverà comunque a 'develop' da release.

6.2.3 Sottomoduli Git

Succede spesso che mentre si lavora su un progetto, è necessario utilizzare un altro progetto al suo interno. Forse è una libreria sviluppata da una terza parte o che stai sviluppando separatamente e utilizzando in più progetti. In questi scenari si pone un problema comune: si desidera essere in grado di trattare i due progetti come separati ma allo stesso tempo essere in grado di utilizzarne uno dall'interno dell'altro.

Il problema con l'inclusione della libreria è che è difficile personalizzare la libreria in qualsiasi modo e spesso è più difficile distribuirla, perché è necessario assicurarsi che ogni client disponga di quella libreria. Il problema con la copia del codice nel tuo progetto è che tutte le modifiche personalizzate apportate sono difficili da unire quando nuove modifiche a monte diventano disponibili.

Git risolve questo problema usando i sottomoduli. I sottomoduli ti consentono di mantenere un repository Git come sottodirectory di un altro repository Git. Ciò ti consente di clonare un altro repository nel tuo progetto e mantenere separati i tuoi commit.

Le dipendenze sono necessarie quanto un pericolo. C'è il rischio della sindrome NIH ed è opportuno evidenziare quelle utilizzate e chiarirle.

A volte le dipendenze sono richieste dal cliente o sono necessarie per integrarsi a sistemi esistenti. Le dipendenze si trovano anche nella progettazione (UML), nel codice e nel testing (junit) e durante il deploy (pacchettizzazioni, build automation, docker).

6.2.4 Limiti di git e la pull request

Git da se offre solo un livello di autorizzazione e nessun livello di tipo review. La pull request permette a un collaboratore di aggiungere delle modifiche a una repo di un altro sviluppatore. I progetti open source soffrono di questi limiti e certi ambienti di hosting hanno provato a risolversi, ad esempio con nuovi meccanismi o workflow come GitHub Flow.

Il meccanismo di fork permette di mantenere legami tra repo su un sito di hosting ma con owner e autorizzazioni diverse.

Viene prevista una fase di review tra la pull request e il merge della pull request.

6.2.5 Gerrit

Gerrit è un server git che fornisce:

- Revisioni del codice: revisionare un commit prima di effettuare un merge
- Controllo degli accessi: permessi di lettura e scrittura specifici per branch

Gerrit mappa i suoi concetti su git, serve solo un client git standard per usarlo. Per mandare un commit in review si utilizza il prefisso 'refs/for/'.

La webapp di Gerrit permette di vedere i commit in review con i relativi diff. Si possono aggiungere commenti e voti. I voti sono divisi per labels (categorie) e alcuni sono bloccanti (veto) mentre altri possono essere assegnati automaticamente, i voti determinano l'approvazione del commit.

Si introducono i ruoli di Approver e Verifier.

L'Approver deve determinare:

- I cambiamenti sono adatti allo scopo del progetto?
- I cambiamenti si adattano all'architettura del progetto?
- I cambiamenti introducono falle che potrebbero causare problemi in futuro?
- Le best practice del progetto sono state rispettare?
- I cambiamenti sono un buon modo per eseguire la funzione descritta?
- Vengono introdotti rischi per la sicurezza o per la stabilità?

Può rispondere con LGTM (Looks Good To Me).

Il Verifier deve:

- Applicare il cambiamento in locale
- Fare una build e testarla

Può rispondere con Verified o Fails.

7 | Dependency Hell

Finora abbiamo visto la tar-pit e come sviluppare in gruppo può creare difficoltà nella comunicazione e nella divisione del lavoro. È necessario coordinarsi e uno strumento disponibile sono i software di configuration management, attraverso questi si controlla l'evoluzione e la revisione dei manufatti. Collaborare in modo ordinato richiede uno sforzo che eccede scrivere soltanto del codice. Per inserire il proprio lavoro in uno sforzo collettivo ci vogliono parecchie energie e bisogna rispettare policy del gruppo (azienda o kibbutz).

Le applicazioni dipendono da:

- Kernel
- Device drivers
- Librerie di sistema
- Librerie di supporto

Se il linguaggio è interpretato, le librerie di sistema non sono solitamente una cosa a cui pensare, rimane il problema delle librerie di supporto. Utilizzare una libreria è utile perché non è necessario reinventare ciò che esiste già, molti cadono nel tranello della sindrome Not Invented Here (rifare qualcosa che esiste già solo per farlo in casa). Esiste il rischio di aggiungere troppe librerie (e quindi dipendenze) inutili, che può avere conseguenze catastrofiche. Solitamente l'aggiunta di una libreria va documentata e motivata.

Distributori di pacchetti come Debian devono il loro successo a una documentazione ricca delle dipendenze. Ogni pacchetto ha un control file che ne specifica le caratteristiche, le dipendenze vengono classificate come:

- **Depends**: un pacchetto non viene configurato se la lista di questi pacchetti non è stata configurata.
- **Recommends**: una dipendenza forte ma non obbligatoria, il pacchetto tecnicamente funziona lo stesso senza.
- **Suggests**: un pacchetto è più utile se utilizzato con questa lista di pacchetti, ma è ragionevole usarlo anche senza.
- **Enhances**: il contrario di Suggests, si dichiara qui che un pacchetto può migliorare l'utilizzo di un altro.

- **Pre-Depends**: come Depends ma viene forzata l'installazione dei pacchetti necessari prima di installare questo pacchetto.

Un pacchetto Debian può avere degli script che vengono eseguiti prima o dopo che un pacchetto viene installato o rimosso. Ogni pacchetto Debian ha una priorità:

- **Required**: necessari al funzionamento del sistema, permettono il boot e l'installazione di altro software.
- **Important**: pacchetti che si dovrebbero trovare un sistema tipo Unix.
- **Standard**: pacchetti che si trovano su un sistema Linux, permettono limitate operazioni come scaricare file.
- **Optional**: pacchetti che nel dubbio si vorrebbero installare e che non hanno requisiti particolari, tipo TeX.
- **Extra**: pacchetti da installare solo se servono e che possono generare conflitti.
- **Essential**: pacchetti che il package manager si riufuta di rimuovere.

Un'installazione Debian di default si ferma a Standard.

Il **dependency hell** sorge quando diversi pacchetti hanno dipendenze dagli stessi pacchetti o librerie condivise, ma dipendono da versioni diverse e incompatibili dei pacchetti condivisi. Se il pacchetto o la libreria condivisi possono essere installati solo in una singola versione, l'utente potrebbe dover risolvere il problema ottenendo versioni più recenti o precedenti dei pacchetti dipendenti. Questo, a sua volta, potrebbe interrompere altre dipendenze e spingere il problema a un altro set di pacchetti. Una forma di dependency hell molto nota è il "DLL hell" di Microsoft Windows. Le DLL sono l'implementazione di Microsoft di librerie condivise. Le librerie condivise consentono di raggruppare codice comune in un wrapper, la DLL, che viene utilizzata da qualsiasi software applicativo sul sistema senza caricare più copie in memoria.

Il problema è dato dall'installazione system-wide delle librerie, per questo spesso vogliamo installazioni specifiche per utente o applicazione. In Python è possibile creare un 'virtualenv' dove installare librerie specifiche per applicazione, senza installare sul sistema. Tuttavia questo può portare a molti duplicati e altri problemi.

8 | Versionamento Semantico

Dato un numero di versione MAJOR.MINOR.PATCH, si incrementa:

- Versione MAJOR quando apporti modifiche con API incompatibili.
- Versione MINOR quando si aggiungono funzionalità compatibili con versioni precedenti delle API.
- Versione PATCH quando si apportano correzioni di bug compatibili con le versioni precedenti.

Il versionamento semantico aiuta a capire la compatibilità di certe dipendenze in modo da uscire facilmente dal dependency hell.

9 | Build Automation

Costruire (assemblare) un prodotto software fatto di molti componenti è tutt'altro che banale:

- Dipendenze da componenti che non controlliamo (dependency hell)
- Dipendenze fra componenti che stiamo sviluppando

9.1 Make

Uno strumento che aiuta nella build di un software è Make. Make è uno strumento di automazione della compilazione che crea automaticamente programmi eseguibili e librerie dal codice sorgente leggendo file chiamati Makefile che specificano come derivare il programma di destinazione.

Come funziona Make:

- Le dipendenze definiscono un grafo aciclico che ammette un unico ordinamento topologico (in quanto si passa una sola volta da ogni target)
- I processi di generazione (recipes) sono eseguiti seguendo l'ordinamento topologico
- Nei make moderni è possibile eseguire processi di generazione indipendenti in parallelo

Il modello di make assume un ambiente di build fisso. L'ipotesi è irrealistica perfino nel mondo dello sviluppo anni '70 (C/UNIX). Compilatori, librerie cambiano molto anche nell'ambito degli standard.

9.2 Autotools

GNU Autotools è una suite di strumenti di programmazione progettati per aiutare a rendere i pacchetti di codice sorgente portabili su molti sistemi Unix-like. Può essere difficile rendere portabile un programma software: il compilatore C differisce da sistema a sistema; alcune funzioni di libreria mancano su alcuni sistemi; i file di intestazione possono avere nomi diversi. Un modo per gestirlo è scrivere codice condizionale, con blocchi di codice selezionati per mezzo di direttive del preprocessore (#ifdef); ma a causa dell'ampia varietà di ambienti di costruzione questo approccio diventa rapidamente ingestibile. Autotools è progettato per affrontare questo problema in modo più gestibile:

- 'configure' verifica le caratteristiche dell'ambiente di costruzione
- genera un config.h con le #define giuste
- genera un Makefile

9.3 Apache Ant

Apache Ant è uno strumento software per l'automazione dei processi di compilazione del software in sostituzione dello strumento di compilazione Make di Unix. È simile a Make, ma è implementato utilizzando il linguaggio Java e richiede la piattaforma Java. A differenza di Make, che usa il formato Makefile, Ant usa XML per descrivere il processo di compilazione del codice e le sue dipendenze.

9.4 Gradle

Gradle è uno strumento di automazione delle build per lo sviluppo di software multilinguaggio. Controlla il processo di sviluppo nelle attività di compilazione e confezionamento fino a test, distribuzione e pubblicazione. I linguaggi supportati includono Java (oltre a Kotlin, Groovy, Scala), C/C++ e JavaScript. Gradle si basa sui concetti di Apache Ant e Apache Maven e introduce un linguaggio domain-specific basato su Groovy e Kotlin rispetto a XML utilizzato da Ant/Maven. Gradle utilizza un grafico aciclico diretto per determinare l'ordine in cui è possibile eseguire le attività, fornendo la gestione delle dipendenze. Funziona su Java Virtual Machine. Raccoglie anche dati statistici sull'utilizzo delle librerie software in tutto il mondo.

10 | Continuos Integration e Delivery

La Continuous Integration è una pratica di sviluppo software in cui i membri di un team integrano il proprio lavoro frequentemente, di solito ogni persona integra almeno quotidianamente, portando a più integrazioni al giorno. Ogni integrazione viene verificata da una build automatizzata (compreso il test) per rilevare gli errori di integrazione il più rapidamente possibile. Molti team ritengono che questo approccio porti a problemi di integrazione significativamente ridotti e consenta a un team di sviluppare software coeso più rapidamente.

Operazioni della Continuos Integration:

- Configuration Management
- Esplicitazione delle dipendenze
- Test d'unità, d'integrazione, d'accettazione
- Build automatizzate
- Deployment simulato o automatizzato

Tradizionalmente, l'integrazione è una delle parti più lunghe e rischiose dei progetti software. L'integrazione continua (incrementale) minimizza il rischio di fallimento.

Processo di Continuous Integration (almeno una volta al giorno):

1. Lavoro su una copia locale sulla macchina di sviluppo.
2. Build vs. Compile: oltre alla costruzione esecuzione di test.
3. Build funzionante sulla macchina di sviluppo.
4. Caricamento sulla macchina d'integrazione.
5. Build funzionante sulla macchina d'integrazione.

Il Continuous Integration è cambiato nel tempo, ad oggi è molto facile creare branch. Si può creare un CI Server che monitora, ad esempio, GitHub, effettua l'integrazione, esegue i test e fornisce i feedback.

Il build non dovrebbe impiegare più di 10 minuti, altrimenti si perde il feedback immediato. Che fare se è necessario più tempo?

- Deployment pipeline: il build è spezzato in più fasi: commit build, slower tests build, ecc.
- Il commit build impiega meno di 10 min, il resto parte poi.

Il sistema dove avviene l'integrazione dovrebbe essere il più possibile "simile" a quello di produzione. Il deployment verso l'ambiente di produzione può essere automatizzato (garantendo così un maggior controllo sulla effettiva configurazione in uso!).

10.1 Principi di Continuos Integration

- **Mantieni un repository del codice sorgente** - questo elemento è propedeutico a tutti gli altri principi, perché senza avere un repository del codice è impossibile automatizzare il build ed i test.
- **Automatizza il processo di build** - per build si intende il processo che trasforma il codice sorgente in un artefatto eseguibile. Deve essere possibile eseguire la compilazione e la creazione dei pacchetti da mettere sui server in modo completamente automatizzato, senza alcun intervento umano.
- **Rendi il build auto-testante** - ogni volta che il codice sorgente viene compilato ed impacchettato, è importante che vengano eseguiti dei test sul sorgente affinché la qualità del codice venga tenuta sotto controllo ed eventuali bug vengano scoperti il prima possibile.
- **Tutti eseguono commit alla baseline tutti i giorni** - è inutile compilare e testare il sorgente se gli sviluppatori sviluppano codice sui propri computer senza sincronizzarsi spesso col codice scritto da altri. È importante però che il codice sia compilabile ed abbia superato tutti i test.
- **Ogni commit fa partire una build** - ogni modifica al codice sorgente condiviso potrebbe generare dei bug e quindi compilare e testare subito dà la possibilità di intervenire immediatamente su eventuali bug.
- **Fai in modo che il build sia veloce** - una build troppo lenta può rendere poco agevole il far partire le build automatici dopo i commit, e potrebbe costringere gli sviluppatori ad aspettare la fine della build prima di effettuare un commit o di verificarne l'esito.
- **Eseguire i test in un clone dell'ambiente di produzione** - se l'ambiente su cui vengono eseguiti i test non è assolutamente identico all'ambiente di produzione, c'è il rischio che qualcosa che è stato testato generi dei bug inspiegabili una volta rilasciato in produzione, con conseguenti danni economici.
- **Fai in modo che sia facile prendere le ultime versioni dei pacchetti** - questo principio serve quando si hanno tanti ambienti che devono essere tenuti allineati.

- **Ognuno può vedere i risultati dell'ultimo build** - la trasparenza permette di stabilire il livello di qualità raggiunto da ogni modulo e capire quali sono i moduli che hanno bisogno di maggiore attenzione.
- **Automatizza i rilasci** - anche il rilascio del software sugli ambienti di test o sull'ambiente di produzione deve essere automatizzato.

11 | Divisione del lavoro

Vogliamo suddividere il lavoro senza la continua necessità di coordinazione. Perché un sottogruppo di lavoro possa procedere in "isolamento" dovrebbe conoscere i componenti sviluppati da altri (o che altri svilupperanno). Cioè il loro comportamento:

- In situazioni fisiologiche (correttezza): il grado in cui un sistema o un componente è esente da difetti nella sua specifica, progettazione e implementazione.
- In situazioni patologiche (robustezza): il grado in cui un sistema o un componente può funzionare correttamente in presenza di input non validi o condizioni ambientali stressanti.

A questo scopo è quindi necessario specificare il funzionamento del sistema. Una specifica è una descrizione delle proprietà del marchingegno/componente utilizzato per risolvere un problema (a sua volta definito dai requisiti di progetto). Le specifiche, perciò, sono una descrizione delle parti che compongono la soluzione: le modalità computazionali però sono lasciate impredicate. (What vs. How)

Le specifiche costituiscono naturalmente l'interfaccia fra gruppi che si suddividono l'implementazione di un sistema complesso. Il coordinamento rimane necessario a livello di specifica: ma accordarsi su "cosa" sembra più facile che sul "come". I sottogruppi avranno la responsabilità di aderire alle specifiche nelle loro implementazioni.

Ci sono una serie di "interface faults" (Perry & Evangelist) che emergono nei sistemi complessi, ad esempio aggiungere funzionalità o problemi di performance.

11.1 Asserzioni

Il meccanismo base per monitorare/verificare l'aderenza di una implementazione alle specifiche (e ridurre gli interface fault): le asserzioni. L'asserzione è un'espressione logica che specifica uno stato del programma che deve esistere o un insieme di condizioni che le variabili del programma devono soddisfare in un punto particolare durante l'esecuzione del programma. Una funzione o una macro che si lamenta a gran voce se un presupposto di progettazione su cui si basa il codice non è vero.

Le asserzioni sono presenti in ogni maggiore linguaggio di programmazione, ci sono dei patter per utilizzare le asserzioni.

12 | Design by Contract

Se usate estensivamente, le asserzioni possono costituire una vera e propria specifica delle componenti del sistema. L'idea della progettazione per contratto (B. Meyer, 1986) è che il linguaggio per descrivere specifiche e implementazioni è lo stesso: la specifica è parte integrante del codice del sistema. La specifica è parte del "contratto" secondo cui ciascun componente fornisce i propri servizi al resto del sistema.

12.1 Tripla di Hoare

$\{P\}S\{Q\}$

Ogni esecuzione di S che parta da uno stato che soddisfa la condizione P (pre-condizione) termina in uno stato che soddisfa la condizione Q (post-condizione). Ogni programma che termina è corretto se e solo se vale la proprietà precedente.

La tripla di Hoare $\{P\}S\{Q\}$ può diventare un contratto fra chi implementa (fornitore) S e chi usa (cliente) S. L'implementatore di S si impegna a garantire Q in tutti gli stati che soddisfano P. L'utilizzatore di S si impegna a chiedere il servizio in un stato che soddisfa P ed è certo che se S termina, si giungerà in uno stato in Q vale. Il lavoro dell'implementatore è particolarmente facile quando: Q è True (vera per ogni risultato!) o quando P è False (l'utilizzatore non riuscirà mai a portare il sistema in uno stato in cui tocchi fare qualcosa!). Weakest precondition (data Q) o strongest postcondition (data P) determinano il ruolo di una feature.

Eiffel è linguaggio object-oriented che introduce i contratti nell'interfaccia delle classi. Il contratto di default per un metodo ("feature") F è TrueFTrue.

12.2 Eiffel

Eiffel è esplicitamente progettato come linguaggio "di progetto", non solo "di programmazione". Ecma standard 367: specify, design, implement and modify quality software. "Programmazione in grande" con oggetti, derivati da classi organizzate in gerarchie di ereditarietà e raggruppate in cluster, che forniscono feature (command o query) ai loro client.

Asserzioni in Eiffel:

- Pre/Post condizioni
- Invarianti di classe

- Asserzioni
- Loop invariant

Le invarianti di classe sono condizioni che devono essere vere in ogni momento "critico", ossia osservabile dall'esterno. In pratica è come se facessero parte di ogni pre- e post-condizione. È possibile avere un supporto run-time alle violazioni: se una condizione non vale viene sollevata un'eccezione. L'eccezione porta il sistema nel precedente stato stabile ed è possibile:

- Terminare con un fallimento
- Riprovare

Spesso si scrivono le "stesse" cose due volte: implementazione e specifica. Il client è responsabile delle precondizioni, il fornitore di postcondizioni e invarianti.

12.3 Design by Contract in Eiffel

Il principio di sostituzione di Liskov stabilisce che, perché un oggetto di una classe derivata soddisfi la relazione 'is-a', ogni suo metodo:

- Deve essere accessibile a pre-condizioni uguali o più deboli del metodo della superclasse
- Deve garantire post-condizioni uguali o più forti del metodo della superclasse
-

Altrimenti il "figlio" non può essere sostituito al "padre" senza alterare il sistema.

Un modo per garantire che le condizioni siano automaticamente vere consiste nell'assumere implicitamente che, se la classe evoluta specifica esplicitamente una precondizione P e una postcondizione Q, le reali pre- e post-condizioni siano:

- PREderived = PREparent P
- POSTderived = POSTparent Q

In Eiffel: 'require else' e 'ensure then'.

Quando una feature accetta dei parametri il tipo del parametro formale è chiaramente una implicita precondizione: infatti limita il dominio di ciò che l'implementatore deve trattare. Il tipo del valore di ritorno costituisce invece una promessa, esattamente come le postcondizioni. Inevitabilmente, quindi, per evitare errori di tipo occorre aderire al principio di Liskov anche nel trattamento dei parametri. C'è un modo facile di ottenerlo, adottato dalla maggior parte dei linguaggi orientati agli oggetto e staticamente tipizzati (come Java e C++): le ridefinizioni (i metodi ereditati) non possono cambiare il tipo dei parametri, sia di quelli in "ingresso", che di quelli in "uscita" come il valore di ritorno: nel gergo dei linguaggi di programmazione questa scelta è detta "novarianza". La novarianza a volte può però essere troppo vincolante per progettisti che

troverebbero in qualche caso più "naturale" restringere il dominio dei parametri di ingresso. Per questo motivo Eiffel fa una scelta piuttosto controcorrente: la ridefinizione può essere covariante cioè ammettere parametri (in ingresso) di sottotipi, infrangendo così il principio di Liskov. Si noti che una varianza in senso opposto (controvarianza) in cui i parametri posso essere sopra-tipi sarebbe invece del tutto accettabile dal punto di vista del principio di Liskov: si tratta in effetti di una possibilità esplorata in alcuni linguaggi (e.g. Sather) ma risultata poco utile e tutto sommato meno naturale anche della novarianza.

Infrangere il principio di Liskov porta a possibili errori, per questo in Eiffel esiste la 'catcall', un'eccezione sollevata quando ci si aspettava un sotto-tipo e si riceve un tipo che ha tutti i medodi che servivano.

12.4 Violazioni del contratto

Nel modello di Eiffel hanno un ruolo importante le eccezioni, che vengono trattate in un modo differente da quello dei più diffusi linguaggi di programmazione. Un'eccezione è un evento di runtime che può causare il fallimento di una chiamata di routine (violazione del contratto). Un errore di una routine provoca un'eccezione nel suo chiamante. Sicuramente c'è un difetto. Se il contratto è corretto:

- Violazione delle precondizioni: responsabile è il client
- Violazione delle postcondizioni o invarianti: responsabile è l'implementatore

Due modalità per trattare le eccezioni:

- Fallimento (panico organizzato): ripulire l'ambiente, terminare la chiamata e segnalare il guasto al chiamante.
- Retry: tenta di modificare le condizioni che hanno portato all'eccezione e di eseguire nuovamente la routine dall'inizio.

Per trattare il secondo caso, Eiffel introduce il costrutto 'rescue/retry'. Se il corpo del 'rescue' non fa 'retry', si ha un failure.