Sviluppo software in gruppi di lavoro complessi

Federico Piscitelli - Omar Zaher - Simone Malesardi

Matricole 970949 - 970944 - 978989

A.A. 2020/2021

Ultimo aggiornamento 25 febbraio 2021

Indice

1 Introduzione

Il corso presenta gli aspetti più organizzativi dello sviluppo software, ma mira a formare figure professionali che contribuiscono allo sviluppo più che alla pura gestione dei progetti. Gli argomenti che verranno trattati sono:

- Cenni a modelli organizzativi di sviluppo (Cattedrale, Bazaar, Kibbutz e gruppi di lavoro agili)
- Supporto offerto dai tool di configuration management e versioning
- Continuous integration & delivery
- DevOps
- Documentazione e specifica mirata al lavoro collaborativo (Design By Contract e linguaggi per la separation of concern)

2 La cattedrale, il bazaar e altri modelli

Nel libro *The mythical man-month* (lettura obbligatoria fino al capitolo 7) Fred Brooks racconta la sua esperienza all'interno di IBM, durante il progetto di OS/360.

"Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it." [1] - Fred Brooks, The mythical man-month

2.1 I problemi identificati da Brooks

Nonostante l'insuccesso di questo progetto, Brooks è riuscito a trarre ottime osservazioni, utili ancora oggi nel mondo dell'ingegneria del software. Le più importanti:

- le tecniche di stima sono poco sviluppate e si tende ad assumere che tutto andrà bene (il modello COCOMO è la tecnica di stima più famosa in ambito software)
- si confonde "effort" con "progress", ma questi non corrispondono. È molto facile stimare quanto si è lavorato, è meno facile misurare di quanto si è progredito, e questo può causare ulteriori ritardi
- il progredire dello sviluppo viene controllato in maniera molto superficiale
- si risponde ai ritardi aggiungendo personale

Tra tutte queste osservazioni, l'ultima è forse la più importante, perchè è proprio da questa che deriva la famosa **legge di Brooks**. Nello sviluppo software, non tutto è facilmente parallelizzabile. Se una donna ci mette 9 mesi a far nascere un bambino, mettendone insieme due non ci vorranno 4 mesi e mezzo.

2.2 La legge di Brooks

"Adding manpower to a late software project makes it later"

Aggiungere forza lavoro ad un progetto software in ritardo, lo fa ritardare ancora di più. Può sembrare un paradosso, eppure è così.

Ogni lavoratore deve essere formato e deve essere a conoscienza delle tecnologie utilizzate, degli obiettivi, della strategia, di quello che è stato fatto fino ad ora. Questa parte di formazione non può essere divisa e varia in base al numero di lavoratori. Inoltre tutto ciò peggiora la comunicazione tra i vari membri dei team. Se ogni parte del lavoro

deve essere cordinata in maniera separata dalle altre parti l'effort aumenta a $\frac{n(n-1)}{2}$. Tre lavoratori richiedono tre volte tanto il lavoro di comunicazione che avviene tra una coppia di lavoratori. Quattro ne richiedono sei volte tanto.

$$\text{costo del coordinamento} \propto n^2$$

2.3 Modelli organizzativi

Secondo Brooks è fondamentale preservare l'integrità concettuale di un progetto. I modelli organizzativi citati sono:

- La Cattedrale: rigorosa separazione fra lavoro architetturale (accentrato) e implementativo (distribuito). Un modello gerarchico in cui si distribuisce il lavoro in maniera piramidale.
- La sala operatoria (H. Mills): un chirurgo e un co-pilota, contornati da una equipe con ruoli precisi, ma tutti giocano per loro. Il vero lavoro viene svolto solamente dal chirurgo.

3 Modelli a bazaar

Il modello a bazaar, a differenza di quello a cattedrale è portato avanti in parallelo da diversi sviluppatori che, ognuno col suo stile, ognuno col suo metodo, arricchiscono di funzionalità il progetto iniziale. Si passa quindi da un modello accentrato, quale il modello a cattedrale, ad un modello estremamente decentrato (il bazaar). Il modello di sviluppo è in buona parte indipendente dal modello di gestione della proprietà intellettuale.

Nell'articolo di Raymond [3] l'analisi è concentrata su due tipi di software open-source:

- GCC (a cattedrale)
- Linux (a bazaar)

L'esempio di Linux è sicuramente il più lampante, infatti, con le sue 27mln di righe di codice, il suo primo rilascio il 17 settembre 1991, i suoi 1991 sviluppatori (di cui 304 nuovi) che hanno lavorato all'ultima release di agosto 2020, è la dimostrazione che anche i progetti a bazaar possono avere successo (contrariamente a quanto si potesse pensare).

3.1 Un buon progetto bazaar

Un buon progetto bazaar per essere tale deve avere 4 caratteristiche fondamentali:

- Il progetto deve nascere da un "fastidio" (itch) di un programmatore
- Bisogna trattare gli utenti come co-sviluppatori per poter effettuare rapidi miglioramenti al codice e per rendere effettivo il processo di debug.
- Bisogna rilasciare presto, spesso e ascoltare i "clienti"
- Bisogna avere un grande numero di tester e sviluppatori in modo che la quasi totalità dei problemi potrà essere riconosciuto e sistemato da qualcuno

Proprio quest'ultimo punto ci introduce la Legge di Linus (Torvalds) che sembra essere in completa antitesi con la Legge di Brooks:

"Given enough eyeballs, all bugs are shallow"

Secondo Raymond [3], la Legge di Brooks sembra non valere più: "Provided the development coordinator has a medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one."

3.2 Dal Bazaar al Kibbutz

Fino ad ora abbiamo parlato di due stili che, metaforicamente, descrivono il processo di sviluppo di un software:

- il bazaar: chiunque può contribuire con del codice al sorgente messo a disposizione dal "promoter", che si occuperà di integrare tutte le aggiunte nel codice principale del progetto
- la cattedrale: un architetto comanda un piccolo gruppo di lavoratori specializzati con un preciso scheduling dei tasks e ognuno con le sue responsabilità

Tuttavia questi due modelli non descrivono ancora tutte le casistiche di stili possibili. Infatti, come riportato anche nell'articolo del Prof. Monga [2], alcuni studi hanno evidenziato come molti progetti open source (web server Apache, il Kernel di Linux, il browser Mozilla) stanno nel mezzo dei due estremi: il progetto è portato avanti da un gruppo di 10-15 persone che controllano la base del codice e sono responsabili per l'80% del codice scritto, ma data l'apertura del codice ci possono essere diversi contributor che correggono bugs o aggiungono features.
Come esempio da analizzare prenderemo il progetto Debian che come scopo ha quello di produrre una coerente distribuzione di software gratuito a partire dal lavoro di volontari indipendenti.
Come si diceva in precedenza l'open source era visto come un bazaar, un magico calderone dove i contribuenti buttavano il loro codice. In realtà però, per creare distribuzioni coerenti ci vuole grande organizzazione e collaborazione tra le parti coinvolte, quindi la metafora del Bazaar sembra non andare più bene, come non va bene neanche la metafora della Cattedrale visto che non c'è nessun architetto e il lavoro è portato avanti solo da volontari.
Il Prof. Monga introduce quindi la metafora del Kibbutz, una comunità cooperativa che persegue uno stesso obiettivo. Le proprietà fondamentali che caratterizzano questa comunità sono:

- le persone entrano a far parte della comunità su base volontaria e non si aspettano nessun tipo di remunerazione per il loro lavoro
- i membri sono tutti d'accordo e perseguono un obiettivo comune
- i membri condividono una coscienza civica e accettano che il loro lavoro sia regolato da esplicite regole stabilite tramite una democrazia diretta

3.3 L'esempio di Debian

3.3.1 Le motivazioni

Il progetto Debian nasce nell'agosto 1993 da un'idea di Ian Murdock che aveva lo scopo di mantenere e assemblare una distribuzione di software Linux lavorando in maniera "aperta", seguendo lo spirito di Linux e GNU.

3.3.2 La struttura

Chiunque può candidarsi per diventare un membro della Debian community. Per essere accettato deve dimostrare di possedere capacità base di trattamento dei software e dei packages e deve conoscere "Debian Free Software Guidelines" e "Debian Social Contract". Accettando di prendere parte al progetto si accetta anche la Debian Constitution che definisce un'organizzazione snella con queste figure presenti:

- Project Leader (DL)
- Project Secretary (DS)
- Technical Committee (TC): formato da 8 memebri
- Individual Developers

DL, DS, TC devono essere diverse persone. **I lavoro è tutto su base volontaria: nessuno è obbligato a fare nulla e ognuno può decidere se candidarsi per un lavoro o meno.**
Ogni anno viene eletto un DL da tutti gli Individual Developers. Un DL può prendere decisioni urgenti, può nominare il DS e insieme al DS e al TC può rinnovare il TC stesso. Il TC è composto da 8 membri (con un minimo di 4) e decide le policy a livello tecnico. Il DS è eletto dal DL ogni anno e si occupa di gestire ogni sorta di elezione ed eventuali distpute sulla costituzione.
Le proprietà e le attività finanziarie sono amministrate da "Software in the Public Interest, Inc" (SPI) dove ogni membro della community può essere membro votante.

Le conseguenze che derivano da una struttura del genere è che nessun individuo può prendere il controllo dell'intero progetto e siccome la coerenza del prodotto finale è l'obiettivo su cui tutti i membri sono d'accordo, questa libertà deve essere controllata da un numero di regole che sono definite dal TC, ma devono comunque avere un alto grado di consenso generale.

Ovviamente nell'organizzazione spiegata fin'ora mancano diversi attori:

- Upstream Authors: contribuiscono a Debian scrivendo software open source. In linea teorica non dovrebbero essere al corrente del progetto Debian, ma in realtà sono in contatto con gli sviluppatori Debian siccome all'interno di Debian gran parte del lavoro è scoprire e correggere bugs.
- Users: la soddisfazione degli utenti è sicuramente una caratteristica che direziona il progetto. Debian propone per i suoi utenti un sofisticato sistema di tracking dei bug (Debian Bug Tracking System, DBTS) dove gli utenti possono non solo segnalare un bug, ma anche proporre una soluzione.
- Ultima categoria (molto in ascesa) sono soloro che utilizzano Debian come base di partenza per creare delle distribuzioni personalizzate.

3.3.3 Le distribuzioni

Una distribuzione è composta da un programma di installazione e un set di packages. Il programma di installazione è in grado di impostare il sistema da zero su diversi tipi di hardware (questo rende l'installazione un processo abbastanza complicato). I packages possono essere recuperati in una serie di CD, dal hard disk o dalla rete.
Tutto l'effort di sviluppo è incentrato sulla produzione di packages. Ma cos'è un package?
Un package è l'unità minima che può essere installata o disinstallata da un sistema.
Ogni DD è responsabile di uno o più packages e ha il compito di svilupparlo e/o mantenerlo. Una volta che questo è completo viene caricato nella repository pubblica da dove i Debian user possono scaricarlo e provarlo sui loro sistemi. Siccome il pacchetto è stato testato solo dallo sviluppatore verrà considerato come alpha-testing e la repository verrà chiamata unstable distribution. Un pacchetto che rimane in unstable-distribution per dieci giorni senza che vengano segnalati bug, viene automaticamente caricato in un'altra repository (più stabile) e prende lo stato di beta-testing. Questa repository è conosciuta come distribuzione di testing.

3.3.4 Coordinamento

L'obiettivo di ottenere una distribuzione coerente, dove ogni programma interagisce linearmente con gli altri, è molto complesso. Infattu, lo sforzo principale portato avanti dai DDs è diretto a garantire che i loro pacchetti siano completamente conformi alle politiche Debian.
Le policies sono la chiave nell'approccio Debian nella distribuzione software. I DDs sono liberi fino a che seguono le policies approvate da tutti. Spesso queste sono basate su standard internazionali o della community e riguardano tutti i problemi di incoerenza in un sistema.

3.4 Conclusioni

Abbiamo già evidenziato più volte la complicatezza nell'obiettivo perseguito dal progetto Debian. Questa sembra essere stata annullata grazie all'approccio utilizzato che non segue nè il modello a cattedrale (con un singolo architetto con tutti i poteri) nè segue il modello a bazaar (dove la coordinazione sta nell'interazione delle parti). Il modello adottato da Debian preserva la libertà di azione e decide democraticamente la coerenza.

4 Gruppi di lavoro agili

Prima di addentrarci nella spiegazione del perchè sono nati i gruppi di lavoro agili facciamo un riassunto di quello che è il pensiero elaborato fino ad ora:

- Sviluppare software in gruppo è difficile (The Tar Pit): poichè le attività sono difficilmente separabili la complessità del coordinamento aumenta col quadrato del numero di componenti (Legge di Brooks)
- La soluzione gerarchica: la squadra di lavoro gioca per il progettista (cattedrale/sala operatoria)
- L'esistenza di progetti open source con migliaia di partecipanti sembra mettere in dubbio la Legge di Brooks
- L'organizzazione di questi gruppi è molto decentrata e le idee progettuali vengono da più parti (bazaar)
- Legge di Linus: le attività di verifica e convalida sono parallelizzabili
- In realtà però la Tar Pit e il caso sono sempre in agguato: più che dei bazaar disordinati, i progetti distribuiti tendono più ad assomigliare a dei kibbutz, con valori tecnologici condivisi e regole di partecipazione e sviluppo forzate da strumenti software

Negli anni '90 (dove UML era padrone) nascono molti approcci legati soprattutto dal fastidio per l'eccessiva enfasi data alla documentazione del processo di sviluppo. Nascono così:

- eXtreme Programming (XP)
- Scrum
- DSDM
- Adaptive Software Development
- ...

I ricercatori dell'ingegneria del software hanno sempre messo in evidenza la crucialità del processo di sviluppo e studiato cicli di vita iterativi.
Il manifesto della **Programmazione Agile** viene pubblicato nel 2001, ed esprime i seguenti concetti:

1. Individui e interazioni > Processi e strumenti
2. Software funzionante > Documentazione esaustiva
3. Collaborazione cliente > Negoziazione contratti
4. Rispondere al cambiamento > Seguire un piano

Tutti questi punti sono però **molto generici e idealistici**, e sono sì belli, ma poco applicabili. Il più concreto di questi è forse il punto 4, perchè riconosce il fatto che è molto facile sbagliare la pianificazione. È meglio quindi salvare risorse per potersi adattare a nuove situazioni.
Al contrario di quanto si è portati a pensare, agile non significa il rifiuto dei processi, ma piuttosto il rifiuto di continue verifiche e benchmark come misura della qualità di quanto prodotto. Quello che fanno gli agilisti è sostituire al canone fatto di processi e misurazioni un canone con dei principi astratti più o meno condivisibili.
Tra i più importanti principi agili troviamo:

- Rilasciare software di valore, fin da subito e in maniera continua (non c'è quindi solo una consegna finale)
- Consegniamo frequentemente software funzionante
- Il software funzionante è la principale misura di progresso
- Cambiamenti nei requisiti anche negli stadi avanzati
- Committenti e sviluppatori devono lavorare insieme quotidianamente
- Conversazione faccia a faccia
- Individui motivati e ben supportati
- Sviluppo sostenibile: essere in grado di mantenere indefinitamente un ritmo costante
- Eccellenza tecnica
- Team che si auto-organizzano
- A intervalli regolari il team riflette su come diventare più efficace
- La semplicità - l'arte di massimizzare la quantità di lavoro non svolto - è essenziale

Il manifesto non è un metodo, nè veramente un 'programma': parla ai cuori più che alle teste degli sviluppatori. È la dichiarazione di un disagio (non voglio essere pagato per scrivere documenti che nessuno legge) e la prefigurazione di un'utopia (il paradiso dei programmatori). I metodi agili (principalmente XP e Scrum) fanno invece proposte concrete (non sempre facilmente identificabili come in linea col manifesto. . .)

4.1 Principi agile nella pratica

Questi principi nell'implementazione in canoni agile si traducono nelle seguenti prescrizioni:

- Team **piccoli** e **auto-organizzati**, senza manager tradizionali, ma facilitatori (evitano che l'auto-organizzazione diventi anarchia)

- Rifiuto di azioni e decisioni **big upfront**, sviluppo interattivo aperto alla variazioni in corso d'opera: cerco di non prendere decisioni troppo importati a meno che non posso fare altrimenti. Mi preoccupo di possibili cambiamenti solo quando il problema si pone effettivamente. Spesso no big upfront è noto con YAGNI (you aren't gonna need it), quindi non sviluppo una cosa finchè non è completamente chiaro che ne ho bisogno. Cerco quindi di evitare l'**over engeneering**

- Misura e controllo del processo di sviluppo, con pianificazioni con orizzonti temporali e funzionali ridotti: ovvero mi concentro molto su quello che farò oggi, non mi preoccupo di quello che farò fra 2 settimane

- Enfasi su testing, intesa come tecnica di sviluppo.

La parte più problematica è la partecipazione della committenza, che infatti è interpretata in maniera molto diversa dai vari approcci agili.

4.1.1 Enfasi sul testing

I requisiti sono sostituiti dalle **User Stories**, ovvero dei template di frasi che il committente compila.

As a `USER TYPE` *I want* `FUNCTIONALITY` *so that* `MOTIVATION`

Queste frasi vengono usate per capire cosa bisogna fare e per valutare se vale davvero la pena farlo.
La specifica che l'ingegneria classica produce è invece sostituita da **casi di test**, che vengono associati alle user stories. Il mio scopo è quindi quello di far passare i test, ovvero sto facendo **Test Driven Development (TDD)**.
Deve quindi essere chiaro che il **test non è un elemento di verifica**.

4.2 Scrum

Scrum vuol dire mischia (tipo Rugby). Tutta la metodologia Scrum è descritta in un piccolo manuale di 17 pagine, molto schematico e sintetico.
Questo framework prescrive alcune regole:

- **Team piccoli**: 7+-2 persone. Il team è auto-organizzato, ma ci devono essere un **product owner** e uno **scrum master**.

- **Presenza di un product owner**: è il membro del team che funge da rappresentate del committente (fa comunque anche gli interessi della propria azienda), in modo tale da non dover scomodare il committente vero e proprio. Funge anche da interfaccia con il committente. Gestisce il backlog. Ciò è necessario perchè Scrum prescrive la necessità di ripianificare spesso.

- **Presenza di uno Scrum Master**: è il facilitatore del gruppo. Cerca di ovviare a problemi di sviluppo, ma anche logistici ecc. Ma esiste anche per fare rispettare i principi dello Scrum, in modo da far funzionare meglio lo Scrum stesso.

- **Membri del team**: oltre che a programmare, devono fare anche le stime, che sono fondamentali.

4.2.1 Pianificazione

Non serve niente pianificare a lunghi periodi, perchè si rischia di sbagliare. È meglio pianificare a brevi intervalli, in modo tale da avere una pianificazione corretta e quindi Utile.
Nello sviluppo agile si creano delle **epopee** (insieme di user story) che si sviluppano in sprint con **lunghezza prefissata** di 1-3 settimane. Queste permette di avere una velocità costante e di pianificare quindi correttezza. So che comunque alla fine di questo sprint dovrò rilasciare qualcosa, anche se non del tutto conforme a quanto pianificato.
Durante lo sprint non è possibile rinegoziare o aggiungere features, al limite si ricomincia lo sprint. Si chiama **closed window rule**.
Per pianificare, scelgo una user story che è sicuramente chiara a tutti. A questa user story assegnerò un punteggio di 1. Esprimerò la complessità di tutte le altre user stories in relazione alla story di valore 1. Ogni membro del team fa una stima in segreto delle user story e la tiene segreta. Si scoprono le stime e poi si discute **insieme** sulla stima definitiva. Questa metodologia si chiama **planning poker**.
Le riunioni sono **timeboxed**, ovvero hanno una lunghezza prefissata. Quando si discute, sono presenti dei **pigs**, ovvero delle persone direttamente interessate che hanno voce in capitolo su una decisione, e dei **chicken**, ovvero persone che possono dare solamente un'opinione.

4.2.2 Riunioni

Sono prescritte le seguenti riunioni:

- **Daily stand up**: (15 min ogni giorno) Cosa abbiamo fatto ieri, cosa facciamo oggi, ci sono impedimenti?
- **Planning**: (ogni 1-5 giorni). Pianificazione di uno sprint, definizione dello sprint backlog con stima per ogni epopea/storia
- **Retrospettiva**: (circa 30 min) Alla fine di uno sprint, per migliorare i processi.
- **Review**: (1 ora) Alla fine di uno sprint, presentazione del lavoro agli stakeholder (quindi anche al cliente).

4.2.3 Tecniche di lavoro

Si usano alcune tecniche di programmazione, provenienti sopratutto dall'eXtreme programming.
Molte di queste tecniche sono utilizzabili e hanno senso solamente se implementate tramite tools e **framework automatici**, come Git, Junit, ecc.

Pair Programming È presente un **pilota** (colui che ha la tastiera) e un **copilota** (che dice cosa cosa fare e cosa scrivere). In questo modo, si è forzati a esplicitare cosa si vuole fare, e il codice scritto deve passare per almeno 2 persone. Ovviamente ci si da il cambio con un certo intervallo di tempo. Può sembrare una perdita di tempo e soldi, ma studi dimostrano che la produttiva è solo leggermente inferiore, a fronte di una qualità del codice più alta. In questo modo, poi, ci sono più persone che conoscono una certa codebase.

Codice Condiviso Tutto il codice deve essere accessibile e modificabile da ogni membro del team. Questo può essere estremante dannoso, ma i metodi agile pongono numerose protezioni per accorgersi di eventuali danni, come la **continuos integration** e il **TDD**.
La ragione dietro ciò è dare la priorità al software funzionante per avanzare nel progetto, piuttosto al dare a ogni persona la "colpa" per ogni singolo malfunzionamento.
Attenzione a non confondere la condivisione del codice come ragione per rinunciare all'**Information Hiding**.

Refactoring Riscrittura di un pezzo di codice senza cambiarne le funzionalità. Per essere sicuro di avere le stesse funzionalità, mi basta controllare che passi gli stessi test (vedi TDD). Molte delle operazioni di refactoring (come cambiare il nome una variabile) hanno dei metodi standard per essere implementati, anche in modo automatico (come per il cambio di nome).
Il refactoring ha l'obiettivo di generalizzare il codice scritto, anche mediante l'ausilio del catalogo dei **code smell**, ovvero delle bad practise di programmazione (come la duplicazione del codice) che andrebbero rimosse.

Test Driven Development (TDD) Non è assolutamente una tecnica di verifica, ma una tecnica di progettazione.
Funziona più o meno così:

1. Aggiungo un test
2. Ripeto tutti i test di assicurandomi che il test fallisca
3. Scrivo il codice per fare passar il test
4. Ripeto i test, che dovrebbero passare
5. **Refactoring**, mantenendo il funzionamento del test
6. Da capo

Eventuali bug vanno esaminati e trasformati in caso di test.
Si svolgono test di unità tramite appositi framework, che metto a disposizione suite con mock ecc.

Velocity Tracking La velocità è una caratteristica del mio gruppo di lavoro, che riesco a conoscere con il tempo, facendo progetti. Il gruppo deve quindi costantemente monitorare il progresso, per vedere se in linea con quanto ci si aspetta.
Il meccanismo principale che mi permette di misurare il progresso è la **task board** come ad esempio il **kanban**. E la tipica board divisa in colonne (ex. Da Fare, In Lavorazione, Fatto). Si possono prevedere anche limiti di numerosità per colonna.

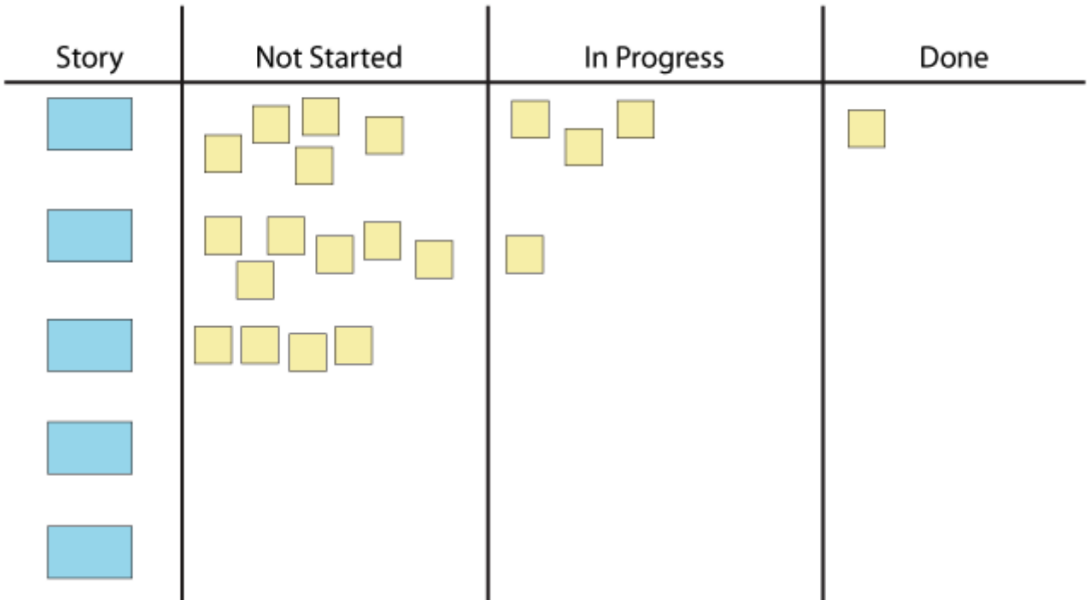


Figura 1: Task board

L'idea è di tenera traccia di cosa si è fatto e di cosa manca fare. Tendenzialmente, nelle ascisse si tende a mettere una misura temporale, e si va a comporre il **burndown chart**.

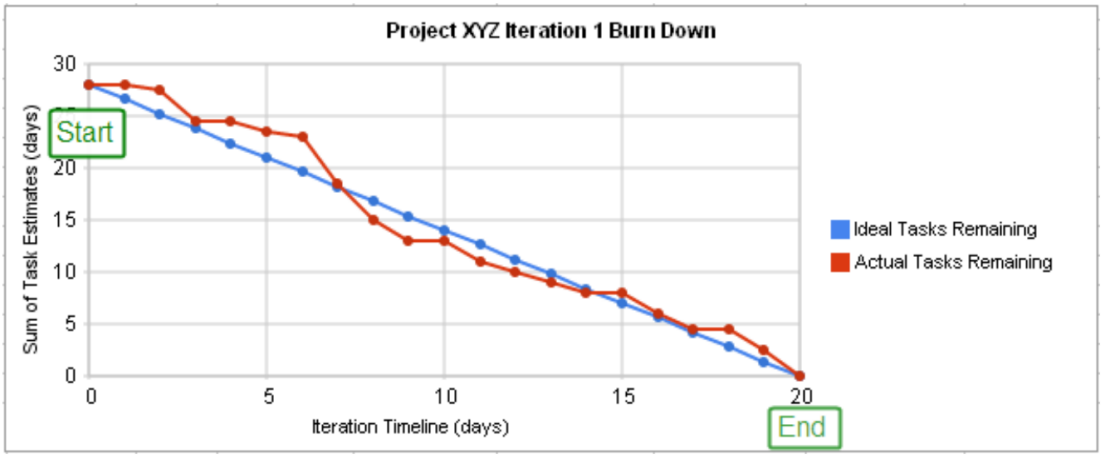


Figura 2: Burndown Chart

Quello che faccio solitamente è tenere traccia dello scostamento rispetto alla pianificazione lineare.
Una approssimazione per calcolare il tempo nelle ascisse è usare i giorni uomo, ad esempio per un sprint di 2 settimane con sei programmatori abbiamo `6 programmatori x 5 giorni a settimana x 2 settimane / 3`. Il diviso 3 serve perchè i meeting ecc portano via circa un terzo della giornata.

5 Continuous Integration

Nel modello a cascata, l'integrazione rappresenta una fase a sé stante. Nei primi anni dello sviluppo software si volevano seguire i modelli di integrazione:

- Bottom Up: partendo dalle foglie viene ricomposto il sistema generale.
- Top Down: partendo dallo schema generico vengono implementate man mano le funzionalità.

In realtà, sopratutto con l'object orientation, vbiene fatto tutto un po' alla rinfusa, senza un chiaro modello di sviluppo.

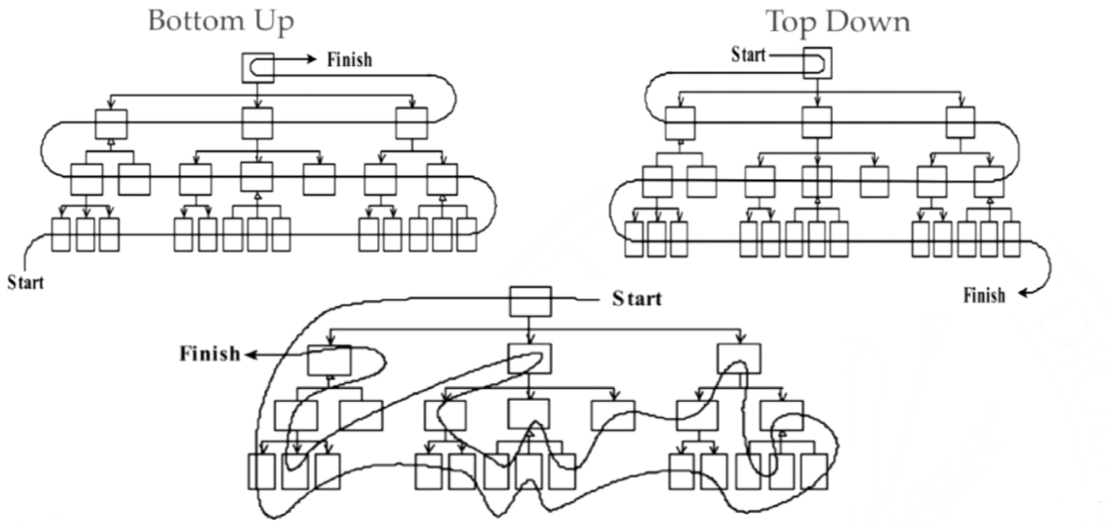


Figura 3: Diversi tipi di approccio durante gli anni

Continuos Integration: allineamento frequente (molte volte al giorno) dagli ambienti di lavoro degli sviluppatori verso l'ambiente condiviso (mainline).

A questo punto l'integrazione non è più fase. Essa viene incorporata nello sviluppo normale. Terminata una parte (una feature, una patch, ...) si integra. Questo però comporta che ogni volta che sviluppo una feature, essa debba essere integrabile. Inoltre, l'integrazione diventa un NONEVENT. Ogni episodio di integrazione ha una durata semplice.

5.1 Capisaldi (Martin Fowler 2006)

Martin Fowler nel 2006 formula alcuni principi:

- **Mantenere una singola repo**: tutti devono fare riferimento ad un'unica sorgente in cui ci deve essere tutto, dal codice alla documentazione. Devo avere anche una chiara visione di tutte le vecchie versioni del codice e di quella più recente.

- **Automatizzare la build**: l'obiettivo è quello di automatizzare e rendere facilmente ricostruibile il progetto, utilizzando strumenti di automatizzazione di alcune operazioni (quali compilazione, ecc.), tenendo conto anche delle dipendenze. In questo modo risparmio tempo e ho un processo più facile e affidabile. Problematiche:
  - rischiamo di esagerare nel voler automatizzare a tutti i costi
  - automatizzare un task piccolo e poco frequente fa perdere tempo
  - automatizzare un task può richiedere più tempo del previsto (a causa di debugging, ecc.), il che potrebbe non costituire un vantaggio, ma può renderlo più affidabile.
- **Build auto-testante**: test automatici eseguiti ogni volta che integro, per controllare che le cose funzionino e che quindi la fase di integrazione abbia avuto successo. Possono essere test di unità e test end-to-end che garantiscono la non regressione, ossia mantenga le funzionalità precedenti.
- **Chiunque committa sulla mainline ogni giorno**: ad ogni commit di altre persone, integro i cambiamenti sulla mainline in modo tale da non avere problemi dopo. Siccome posso pushare solo dopo aver risolto i conflitti, l'integrazione avviene sulla mia macchina.
- **Ogni commit dovrebbe buildare sulla macchina di integrazione**: le build sono automatizzate. Se falliscono dei test, non consento il push. Il programma non deve funzionare solo sulla propria macchina, ma anche sulla macchina di integrazione.
- **Sistemare le build rotte subito**: se i controlli falliscono, quindi l'ultima versione sulla mainline non funziona, va risolto il più velocemente possibile. Si possono adottare tre tipi di soluzione:
  - correggere l'errore
  - tornare ad una versione precedente (revert)
  - utilizzare una "guardia" che determina l'idoneità rispetto ad una certa serie di test, altrimenti rimane in pending (Git hooks, Gerritt, ecc.).
- **Mantenere veloce la build**: una build veloce permette di testare ad ogni commit.
- **Eseguire i test su un clone dell'ambiente di deployment**: avere una macchina completamente uguale a quella della produzione è indispensabile durante lo staging (per comprendere problemi di retro-compatibilità delle dipendenza, ecc.), in modo da poter verificare che funzioni da entrambe le parti. Docker, permette di isolare i singoli processi e l'ambiente in cui viene eseguito.
- **Rendere facile ottenere l'ultimo eseguibile**: fornendo solo il sorgente potrebbe essere che con ambienti di sviluppo diversi il codice non compili. Quindi, bisogna fornire anche gli eseguibili, non solo dell'ultima versione, ma anche delle precedenti.

- **Tutti devono poter vedere cosa succede**: bisogna garantire trasparenza, il che vuol dire consentire la visibilità del punto in cui si trova il codice sulla mainline, quali e quanti test ha passato, etc.
- **Automate Deployment**: se garantiamo che ci sia sempre nella head della macchina di integrazione la versione più nuova e corretta possiamo automatizzare anche il meccanismo di passaggio in produzione.

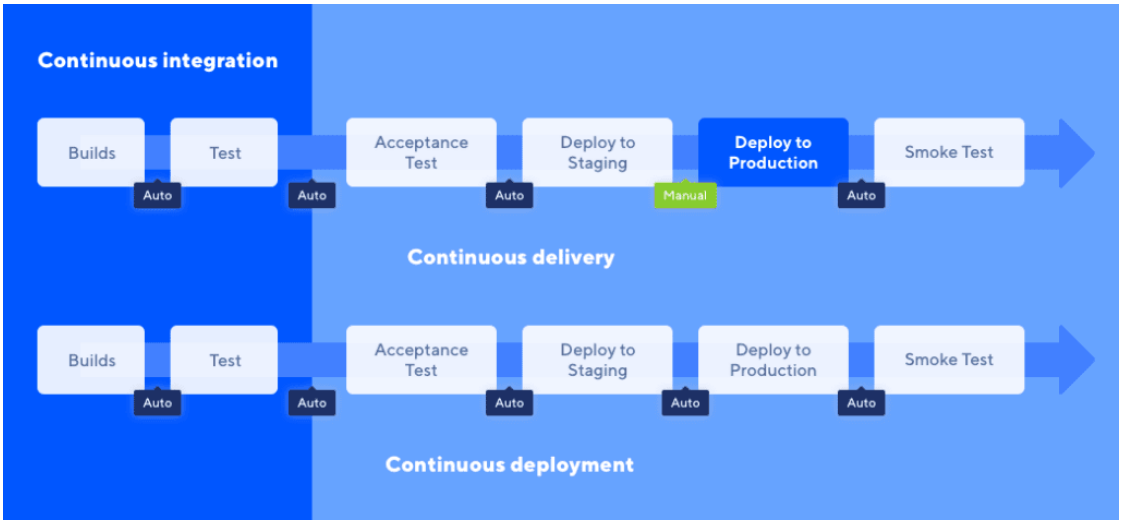


Figura 4: Continuous Delivery vs Continuous Deployment

Non tutto ciò che Fowler riporta in questo articolo è oro colato, essendo un articolo ormai datato. Ad esempio, invita a "tenere i branch al minimo", poiché la gestione dei branch inizialmente era molto complicata.

5.2 Configuration Management

Dagli anni 70 comincia ad essere utilizzato nel mondo del software.

Configuration Management: pratiche che hanno l'obiettivo di rendere sistematico il processo di sviluppo, **tenendo traccia dei cambiamenti** in modo che il prodotto sia in ogni istante in uno stato (configurazione) ben definito.

Gli oggetti di cui si controlla l'evoluzione sono detti configuration item o artifact (manufatto, NON artefatto). Tre scenari di esempio:

- Programmatore solo, utile ad esempio per mappare somiglianze tra le versioni
- Collaborazione studente professore

- Blaming, in modo da sapere per ogni riga di codice chi è stato l'ultimo a modificarla (annotare il listato).

L'approccio iniziale era quello di versionare i singoli file, e non l'insieme. Inoltre, per la collaborazione venivano usati sistemi di write lock piuttosto scomodi.
Gli SCM (Source Code Management) sono per lo più indipendenti da linguaggi di programmazione e applicazioni. Lavorano genericamente su file, preferibilmente fatti di righe di testo e calcolando le differenze in base a queste.

- anni '80: strumenti locali (SCCS, rcs, ...)
- anni '90: strumenti client-server centralizzati (cvs, subversion, ...)
- anni 2000: con l'avvento dell'open source e di internet, si sono resi necessari nuovi metodi di collaborazione **distribuiti** (peer-to-peer), come git e mercurial. Questo è necessario a permettere una collaborazione da parte di molte persone, anche saltuaria.

5.2.1 Artifact

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

5.3 Sincronizzazione

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 (o commit) dichiara la volontà di registrare un determinato change-set

Queste vanno a versionare un set di cambiamenti su uno o più file. Non si va quindi a versionare immediatamente ogni linea di codice. Queste operazioni vengono attivate rispetto a un'applicazione di repository. In teoria, non servirebbe un nodo centrale, perchè ogni peer possiede l'intera copia della codebase, che può essere sincronizzata con gli altri. In pratica, si usa un **repository centrale** per sincronizzare più velocemente e facilmente tutte le modifiche.

5.3.1 Lavoro concorrente

L'accesso concorrente si può gestire in due modi:

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

Il merge rimane un'operazione delicata che generalmente viene trattata con strategie diverse:

- Lavoro parallelo su artifact diversi
- Lavoro parallelo sullo stesso artifact, ma su hunk diversi
- Lavoro parallelo sullo stesso artifact e sullo stesso hunk

Un hunk è un insieme di righe adiacenti che voglio trattare omogeneamente rispetto al mio tool di versioning.

L'ultimo caso necessita sempre di lavoro intelligente. Nel resto dei casi, dipende. Nel caso di lavoro parallelo sullo stesso artifact, quando le due revisioni hanno un antenato comune (per esempio la revisione da cui entrambi sono partiti) si può facilitare il lavoro di merge (3-way merge). Siano 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 Git

6.1 Comandi base

```
$ git init
```

Inizializza una nuova repository creando una directory .git che al suo interno contiene:

- HEAD: è il file che contiene il puntatore o il riferimento alla WD o al branch e il corrispettivo ultimo commit
- refs: è una cartella che contiene i riferimenti agli oggetti nella WD
- config: è una cartella che contiene le configurazioni
- objects: è una directory nella quale sono salvati tree, commit e blob. Ogni oggetto ha la sua sotto cartella.
- index: è un file che viene utilizzato quando aggiungiamo files allo stage per un commit.
- ...

```
$ git add
```

Aggiunge uno "snapshot" dei file in quel momento nell'index. Git non versiona le directory vuote.

```
$ git status
```

Visualizza: i path che presentano differenze tra il file index e il commit corrente a cui punta HEAD, i path che presentano differenze tra l'albero di lavoro e il file index e i path nell'albero di lavoro che non sono tracciati da Git

```
$ git log [--all]
```

Mostra i log di commit del branch in cui ci troviamo includendo hash, autore, data e messaggio. Includendo –all mostra tutti i commit di tutti i branch.

```
$ git hist [--all]
```

È un alias per git log definito come:

```
$ git config --global alias.hist "log --pretty=format:'%C(yellow)[%ad]%C(reset)
    %C(green)[%h]%C(reset) | %C(red)%s %C(bold red){{%an}}%C(reset) %C(blue)
    %d%C(reset)' --graph --date=short"
```

e non fa altro che un "pretty print" di git log

```
$ rm -rf .git
```

Rimuovo la repository: più in generale è il comando per eliminare una cartella.

```
$ git cat-file [-s -t -p] hash
```

È il comando utilizzato per interpretare gli objects: ritorna la dimensione, il tipo o il contenuto del file corrispondente all'hash (basta fornire i primi n caratteri dell'hash dell'obj che lo rendono univoco e distinguibile). Si possono avere tre tipi di objects:

- commit
- tree (che sono la rappresentazione delle directory)
- blob (che sono la rappresentazione dei file)

```
$ git ls-files [-s]
```

Mostra i file sull'index o nel repository...files che sono nello stage che saranno aggiunti al prossimo commit, questo perchè nell'index ci sono sempre i puntatori a ciò che verra messo nel prossimo commit.

```
$ git restore
```

Recupera la versione che si aveva nell'index a discapito di quella nella WD

```
$ git commit [-a] [-m messaggio] [--amend]
```

Si apre una nuova scheda nel terminale dove ci viene chiesto di inserire un messaggio per il commit (deve esserci almeno una linea decommentata). Una volta eseguito questo comando viene creato un oggetto di tipo commit (nella cartella objects) che ha al suo interno un oggetto di tipo tree (con il suo hash), l'autore, il committer, il messaggio e il puntatore al commit padre.
Andando a vedere (tramite il comando git cat-file -p) il contenuto del tree è quello che avevamo nello stage (quindi nel file index) in questo caso visto che c'è un solo livello di innestamento. All'interno del file del tree c'è scritto il tipo (tree) e poi gli hash dei file in formato binario per questioni di ottimizzazione.
Aggiungendo il –amend si può andare a modificare il commit più recente. Può essere utilizzato in diversi modi a seconda delle nostre necessità:

- possiamo combinare i cambiamenti nello stage con il commit precedente al posto che crearne uno nuovo
- possiamo utilizzarlo invece per modificare il messaggio di commit precedente senza cambiare il suo snapshot

Usando il comando amend non solo alteriamo l'ultimo commit, ma ne andiamo a creare direttamente un altro. Per Git sarà come un nuovo commit.
Usando -a prima di -m posso evitare di fare la add per tutti quei file che sono stati tracciati almeno una volta in passato (ossia per cui è già stata fatta la add una volta)

```
$ git branch nome
```

Crea un nuovo branch ossia aggiunge un file in refs/head/ chiamato con il nome del branch (sul file HEAD il puntatore rimane al branch che stavamo puntando prima)

```
$ git checkout [-b] nome
```

Esegue il checkout (o crea il branch ed esegue il checkout) spostando il puntatore di HEAD.

HEAD punta ad un nome di un ramo (un file) che troviamo sotto refs/head/ e in situazioni standard è master che contiene l'hash del commit corrente. Facendo git checkout nome_branch spostiamo il puntatore nel file HEAD facendolo corrispondere al file che si chiama come il nome_branch. Se il commit del nuovo branch non corrisponde al commit del vecchio il comando checkout (e se non ci sono conflitti) andrebbe a cambiare i riferimenti nell'index e file nella WD.

```
$ git tag [-a] nome [-m messaggio]
```

Crea un tag sul commit corrente. Abbiamo due modi in cui possiamo creare tag:

- annotated: aggiungiamo un messaggio al nome del tag
- lightweight: specifichiamo solo il nome del tag

```
$ git tag -l
```

Mostra tutti i tag

```
$ git show nome
```

Mostra i dati del tag e del commit ad esso associato.

Abbiamo citato i branch e i tag. Erroneamente si potrebbe pensare che siano simili, in realtà sono molto diversi, infatti i tag rimangono fermi su una commit, mentre i branch evolvono. (?)

```
$ git hash-object nome_file
```

Ritorna l'hash dell'oggetto senza aggiungerlo alla lista degli obj.

```
$ git hash-object -w nome_file
```

Crea un blob a partire da un file, aggiunge l'hash dell'oggetto agli obj senza però aggiungere il tracciamento nel file index.

```
$ git fsck
```

Dove fsck sta per file system check, fa un controllo sugli obj

```
$ git checkout -- .
```

Ripristina la situazione della WD a partire dall'index

```
$ git fetch
```

Recupera dalla repository remota quello che non ho nella wd.

```
$ git reflog [nome_branch]
```

Mostro i log del file HEAD

```
$ git diff [--cached] [ref] [ref1] [--] [files]
```

Mostra le differenze tra la wd e l'index. Se includiamo –cached mostra le differenze tra la repository e la index.

```
$ git checkout [ref] [--] [files]
```

Vengono copiati i files nella versione presente in ref dentro a index e wd. Se non sono indicati files allora viene spostata la head.

```
$ git reset [ref] [--] [files]
```

Simile a checkout ma oltre a spostare il puntatore dell'HEAD può modificare anche l'index e la wd.
Abbiamo diverse possibilità di utilizzo:

- includendo –hard l'index and la wd vengono resettati fino a diventare uguali a quelli del commit specificato
- includendo –soft i ref pointer vengono aggiornati, mentre l'index e la wd rimangono inalterate
- includendo –mixed (o non includendo nulla) vengono aggiornati i ref pointer, l'index è resettato allo stato del commit specificato. Ogni cambiamento che è stato rimosso dall'index viene spostato nella wd.

```
$ git merge [ref]
```

Integra i cambiamenti dal commit a cui si riferisce nel branch corrente, partendo dal punto in cui le loro storie si sono divise. In parole povere unisce le versioni e, se non ci sono conflitti, crea il commit del merge. Se ci sono conflitti invece, questi vengono salvati nella wd, mentre nell'index verrà salvato ciò che non ha creato conflitti.

```
$ git merge --no-ff [ref]
```

A differenza del merge normale, che tende ad appiattire la storia, questo tiene visibile la separazione tra i branch.

```
$ git revert [ref]
```

È un comando safe, non distrugge o cambia la storia ma crea un commit che inverte gli effetti del commit citato (che potrebbe anche non essere l'ultimo)

```
$ git rebase [-i] ...
```

Ribasare è il processo che muove o combina una sequenza di commit, cambiandone il commit base. In pratica modifico la base di origine del mio branch facendo sembrare il suo inizio su un altro commit. Internamente Git crea nuove commit e le applica alla nuova base.
Includendo -i richiamiamo il rebase interattivo che va a modificare ogni singolo commit eseguito fino a quel momento in maniera interattiva ossia dovremo specificare noi quale operazione eseguire su quale commit.

```
$ git bisect
```

Serve per eseguire la ricerca dicotomica tra le varie commit quando scopriamo un problema che siamo sicuri non fosse presente ad un punto della nostra storia, ma non abbiamo idea di quando sia stato inserito.

```
$ git bisect start #per cominciare il bisect
$ git bisect reset #per fermare il bisect
$ git bisect bad   #se dove sono ora è errato
$ git bisect good  #se dove sono ora è corretto
```

```
$ git bisect run sh -c "shell script"
```

Esegue uno script della shell che ritorna 0 o 1 che vengono interpretati dalla git bisect come bad e good. In questo modo riusciamo a rendere più "indipendente" e automatica la bisect

```
$ git stash
```

Mette da parte wd e index e poi fa reset. Può essere utilizzato in diverse casistiche:

- Interrupted workflow: il mio capo mi chiede qualcosa di urgente (emergency fix), ma io stavo già lavorando ad altro nel mio wd

  ```
  $  git stash
  #lavoro lavoro
  $  git stash pop [--index]
  ```

  Includendo –index alla pop porto sia il file index che la wd, altrimenti avrei portato solo la wd.

- Pulling into a dirty tree

  ```
  $ git stash
  $ git pull
  $ git stash pop
  ```

- Testing partial commits: risolve il problema di testare una index diversa da wd che si vuole committare

  ```
  $ git add --patch foo #aggiungi la prima parte all'index
  $ git stash push --keep-index #salva gli altri cambiamenti nello stash
  # edit/build/test first part
  $ git commit -m 'First part' #commit cambiamenti testati
  $  git stash pop #lavora agli altri cambiamenti
  #ripeti finchè rimane solo un commit
  # edit/build/test remaining parts
  $ git commit foo -m 'Remaining parts'
  ```

```
$ git filter-branch
```

Permette di riscrivere la storia in maniera più completa:

```
$ git filter-branch --index-filter 'git rm --cached --ignore-unmatch filename
```

Permette di fare come se un file non fosse mai versionato

```
$ git filter-branch --subdirectory-filter foodir -- --all
```

Permette di ricavare un repository con solo il contenuto di una subdirectory

```
$ git filter-branch -f --tree-filter 'mkdir ._p ; mv * ._p; mv ._p core;' -- --all
```

Permette di fare come se alcuni file fossero sempre stati in una sottodirectory

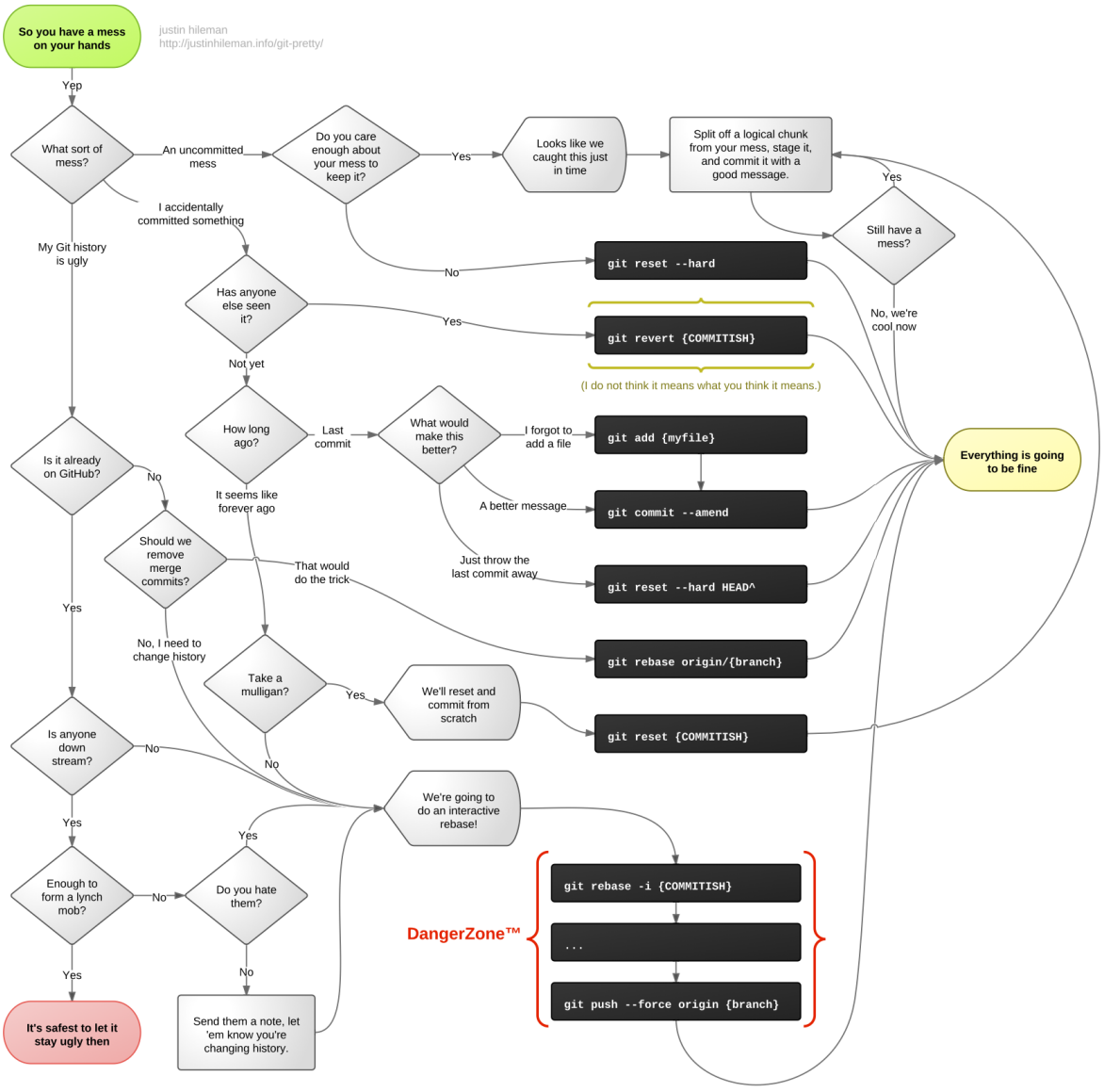


Figura 5: Cambiare la storia con git

6.2 Git Hooks

Come molti VCS anche Git può lanciare custom scripts al verificarsi di determinate azioni. Esistono due gruppi di hooks:

- client-side: sono triggerati quando vengono eseguite operazioni come commit e merge
- server-side: vengono lanciati quando si eseguono operazioni sulla rete come push e pull

Client-side	Server-side
pre-commit	pre-receive
prepare-commit-msg	update
commit-msg	post-receive
post-commit	
pre-rebase	
post-rewrite	
post-checkout	
post-merge	
pre-push	

6.2.1 pre-commit hook

Viene lanciato subito dopo un comando commit (prima di editare il messaggio) a meno che non si sia usata l'opzione –no-verify. Può determinare il fallimento di un commit.

6.2.2 post-commit hook

Viene lanciato dopo il commit (dopo che è stato accettato), quindi non ne può cambiare lo stato. Viene utilizzato solitamente per mandare delle notifiche o per lolcommits

6.2.3 update (server-side)

Possono essere utilizzati per rifiutare un push (CI con capacità di bloccare un commit) e differiscono nel come vengono chiamati.

6.2.4 post-receive (server-side)

Può essere usato per notificare

6.3 Git Flow

Fino ad ora abbiamo utilizzato i branch in maniera piuttosto "casuale". AVH ha generato un set di script (prima ad uso interno e successivamente reso pubblico) per regolamentare l'utilizzo dei branch, introducendo una tipizzazione, nuove operazioni guidate eincludendo anche i remote. Per poter cominciare ad utilizzare git flow va lanciato il comando

```
$ git flow init
```

A questo punto la repo sarà gestita da git flow che si occuperà dei comandi a basso livello (git branch, git checkout) mentre a noi basterà tenere a mente le ”regole” dei branch:

- **master**: ramo con vita infinita che contiene le versioni stabili e pronte alla consegna
- **develop**: ramo con vita infinita che è considerato quello di integrazione

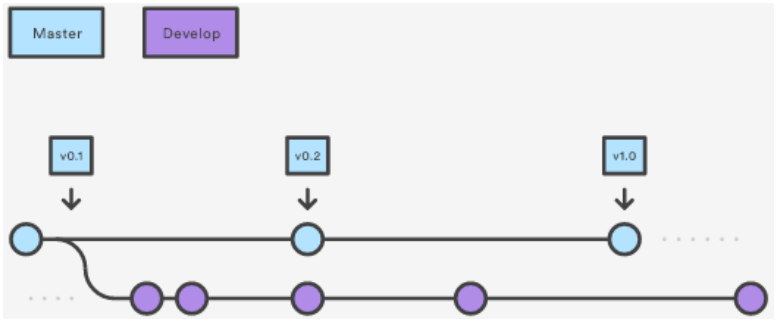


Figura 6: I rami master e develop non hanno fine

- **feature**: ramo che contiene lo sviluppo di una feature che poi andrà a confluire nel ramo develop. Non possono esserci più feature aperte con lo stesso nome

```
$ git flow feature start feat1
#hack hack hack
$ git flow feature finish feat1
```

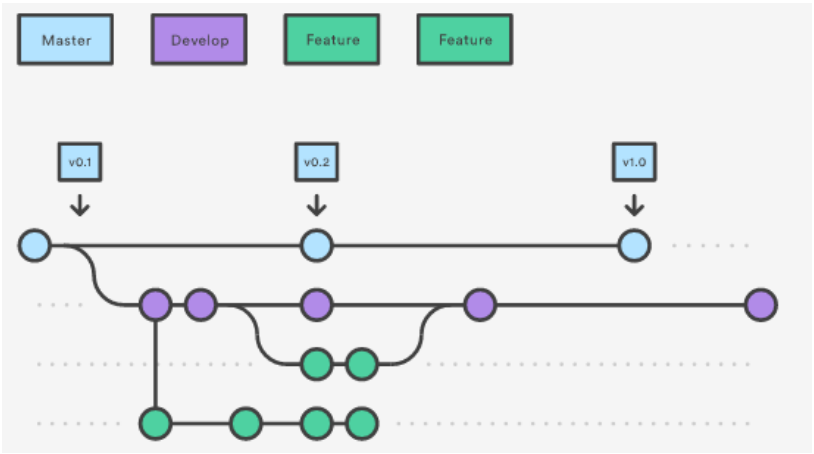


Figura 7: Il ramo feature confluisce nel ramo develop

- **release**: è il ramo che si posiziona tra develop e produzione. Viene creata una release quando si vuole "congelare" lo stato del mio prodotto che verrà mandato in produzione.

```
$ git flow release start ver1
#test fix test fix
$ git flow release finish ver1
```

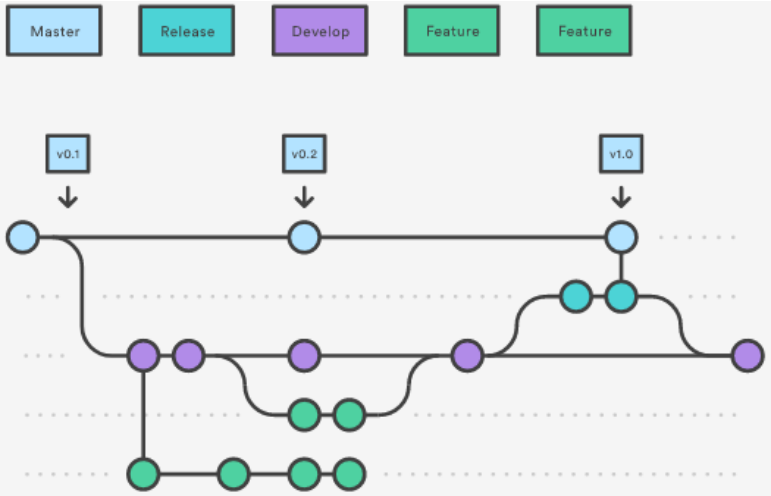


Figura 8: Ramo release

- **hotfix**: ramo dove ci sono le riparazioni veloci dei difetti urgenti che non possono aspettare una prossima release

```
$ git flow hotfix start hotfix1
#hack hack hack
$ git flow hotfix finish hotfix1
```

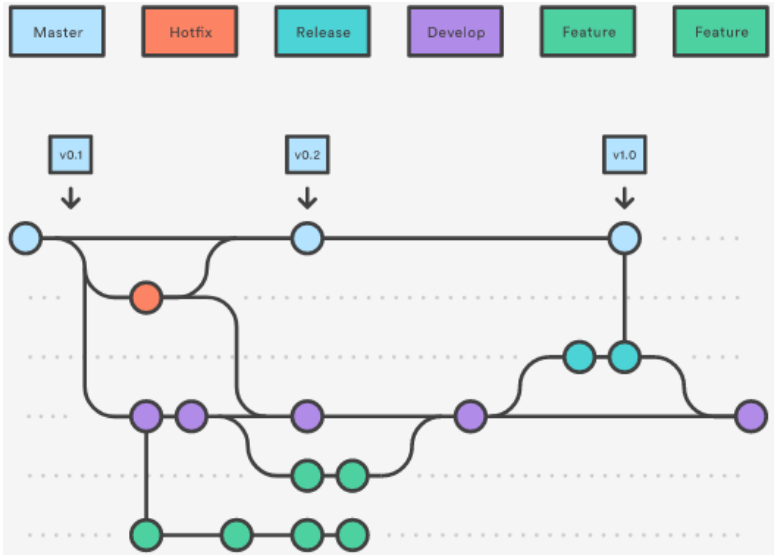


Figura 9: Ramo hotfix

- **support**: ramo che serve per fornire supporto a versioni vecchie del sw (non è molto usato)

6.4 Autorizzazioni e revisioni

Possiamo accorgerci di alcuni limiti di git: abbiamo un singolo livello di autorizzazione e nessun livello di review

```
$ git request-pull <start> <url> [<end>]
```

È il comando che viene utilizzato per generare un riassunto del lavoro fatto su una determinata repo, utilizzato in passato per convincere altri a pullare il codice (ti faccio vedere cosa ho fatto quindi sei più motivato a scaricare il mio codice).

La maggior parte dei progetti open source soffre di questi limiti. Gli ambienti di hosting hanno cercato di trovare soluzioni alternative inventandosi nuovi meccanismi e cercando di imporre nuovi workflow.

Fork: risolve un primo problema di autorizzazioni perchè permette di mantenere i legami tra la repository originale e la mia forkata, ma con owner e autorizzazioni diverse. Sulla mia repo forkata ho permessi di scrittura, ma non posso scrivere sulla repo originale. I file che non vengono modificati (ossia di cui non cambia il blob) non vengono duplicati, ma viene mantenuto un puntatore alla repo originale.
Tra creazione e deploy è prevista una fase di review: rimane il controllo della repo originale. Ho forkato una repo, ho fatto un bug fix, mando una pull (merge) request, dove

viene revisionato il mio codice.

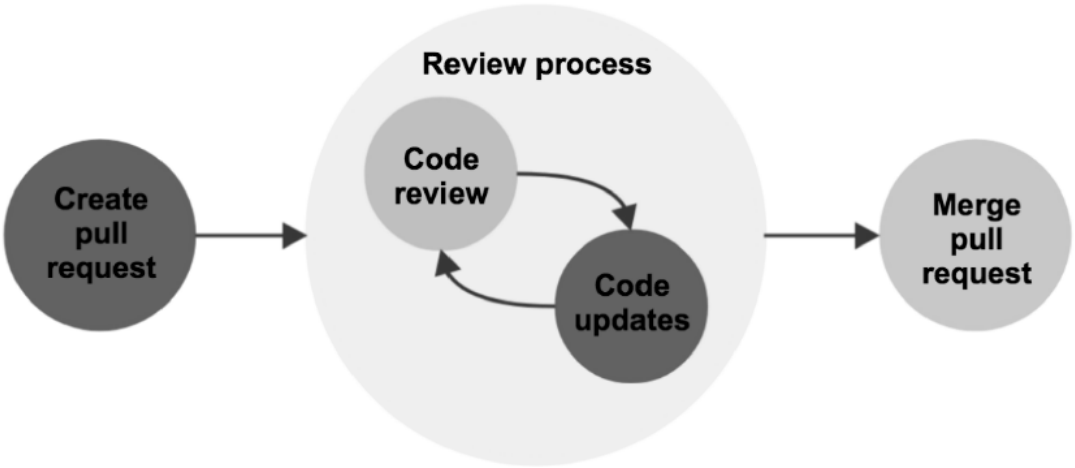


Figura 10: Processo di revisione di una pull/merge request

Pull/merge request: permette di gestire interazioni lasche tra sviluppatori mediate dal sito di hosting. Per creare una pull/merge request sono necessari:

- titolo della pull request
- target branch
- source branch

Sotto la richiesta si genera un thread di discussione tra sviluppatori che possono fare da garanti o discutere sulla pull/merge request.

6.4.1 Gerrit

È un progetto di google AOSP: l'obiettivo è quello di rendere decentralizzata anche la fase di review tramite delle peer review di sottomissioni.
Ci sono due ruoli principali:

- Verifier: scarica il codice da verificare sulla propria macchina, esegue la build e i test e comunica allo sviluppatore il risultato aggiungendo un messaggio di spiegazione
- Approver: deve verificare che la modifica proposta passi in maniera positiva un set di domande e poi la può marcare come LGTM (Looks Good To Me)

La sua architettura è abbastanza semplice e può essere pensata come due git server: facciamo un push su un git server (dove hanno accesso in lettura i reviewer e dove gli

altri possono solo scrivere), i reviewer approvano e "spostano" sul git server pubblico, che tutti possono vedere e leggere.
L'hash non è più del file ma del changeset, quindi si fa un versionamento della modifica.

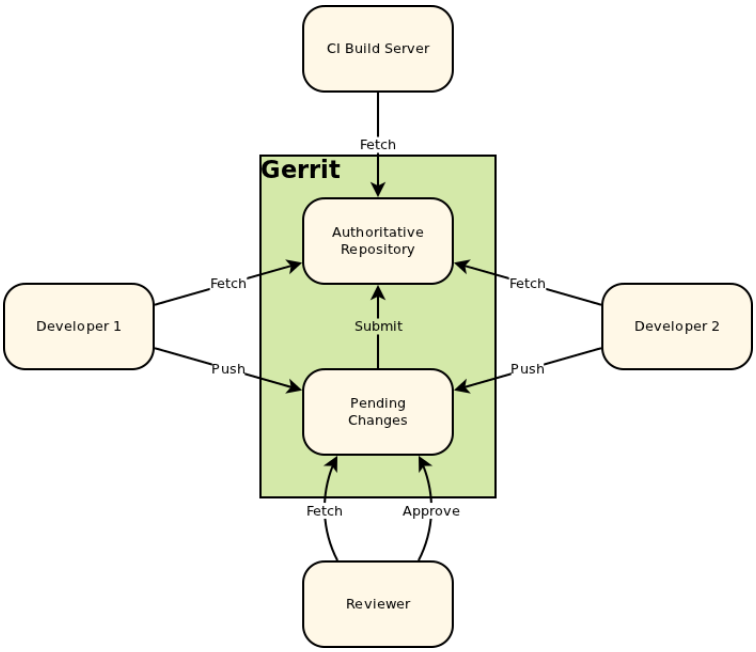


Figura 11: Processo di revisione di una pull/merge request

7 Dependency Hell

Il software che scriviamo ha un sacco di dipendenze (libreria di sistema, ecc.). Il software che si scrive non è mai tutto sotto il nostro controllo (non è solo il codice che scriviamo); le nostre istruzioni si appoggiano a moltissime altre che non sono sotto al nostro controllo.

Qualsiasi applicazione dipende da componenti software fuori dal controllo del produttore: kernel, device driver, librerie di sistema, librerie di supporto.
Anche una semplice applicazione deve interagire con le librerie messe a disposizione dal S.O. Ad esempio la calcolatrice di gnome ha 83 dipendenze.

Le dipendenze creano problemi, vogliamo cercare di essere indipendenti.

Troviamo il concetto di **dependency hell**: le dipendenze rischiano di sfuggirmi di mano, ovvero perdo il controllo delle dipendenze. Io dipendo da cose che cambiano senza che me ne accorga e l'applicazione non funziona più.

Questo concetto va sicuramente meglio con i linguaggi interpretati: alle dipendenze di sistema generalmente sopperisce l'interprete (ma non sempre: con la macchina virtuale Java per esempio può essere piuttosto faticoso utilizzare specifiche librerie grafiche).

- un'applicazione usa librerie per non "reinventare la ruota"
- evitare la sindrome NIH (Not invented here): i programmatori diffidano da cose che non hanno sviluppato loro. Sovraccarica troppo il gruppo di lavoro e nasce dall'esigenza di non dipendere troppo.
- evitare le dipendenze inutili

Le dipendenze vanno il più possibile esplicitamente documentate e motivate.

Abbiamo già discusso che distributori come Debian devono gran parte del loro successo alla ricca documentazione delle dipendenze:

- ogni pacchetto è regolato da un control file, che specifica le caratteristiche
- le dipendenze: Depends, Recommends, Suggests, Enhances, Pre-Depends
- gli script da eseguire per mantenere l'integrità del sistema: preinst, postinst, prerm, postrm
- la priorità: Required, Important, Standard, Optional, Extra

Il problema esiste non solo a livello di sistema, ma anche di singola applicazione (DLL hell):

- Riproducibilità
- Ambienti di "scripting" per i quali non sono possibili compilazioni "statiche"
- Gestione di installazioni concorrenti di diverse versioni

7.0.1 Python

Esaminiamo il caso di Python, ma considerazioni analoghe valgono ormai per moltissime piattaforme di sviluppo (npm, stack, ...) Onnipresenti poi i sistemi di distribuzione centralizzata:

- PHP Pear
- CPAN Perl
- CTAN Tex
- MELPA Emacs
- ....

Python fornisce un meccanismo standard per documentare le dipendenze di un'applicazione:

```
from setuptools import setup

setup(
    name="MyLibrary",
    version="1.0",
    install\_requires=[
    "requests",
    "bcrypt",
    ],
    # ...
)
```

Esistono poi dei punti di distribuzione centralizzata: per esempio PYPI e naturalmente un package manager: **pip install requests**, installa la libreria request nel mio sistema (sovrascrivendo quella vecchia se presente, creando problemi ai programmi che usavano la versione vecchia).

Cosa succede se c'era già la libreria request? Viene fatto un upgrade, ma posso avere un'applicazione che dipendeva dalla versione vecchia e che ora non funziona più.

Come posso fare per installare più versioni delle librerie? Con sistemi come virtualenv non viene più installata la libreria a livello di sistema ma può essere installata a livello di utente, a livello della singola directory o a livello di sistema.

Sempre più spesso non vogliamo installazioni "system-wide", ma "user-wide" o addirittura "application specific".

```
$ cd ~/usr/local/src/app/
$ virtualenv env
New python executable in env/bin/python
Installing setuptools.............done.
```

```
Installing pip..............done.
$ pip ....
$ python ....
```

(Con source ./env/activate si pu'o semplificare la chiamata dei programmi)

virtualenv funziona nel seguente modo (guardando il codice precedente:

- richiama virtualenv che crea una directory di nome "env" dove mette tutto ciò che serve per permettere all'applicazione di funzionare
- virtualenv va attivato, quindi mettere nel path le directory dove ha installato i vari programmi in modo tale che poi quando viene eseguito un comando python, non viene eseguito il python del sistema, ma quello dello specifico virtualenv

Come garantisco l'esplicitazione delle dipendenze?

```
$ pip install pippo
$ pip freeze > requirements.txt
$ pip install -r requirements.txt
```

pip tiene traccia anche delle cose che ho installato. pip freeze > file.txt : salva in file.txt le librerie che sono già installate nella macchina. Esempio:
python3 -m virtualenv PLUTO : installa nella dir PLUTO il virtual enviroment.
pip install non scarica solo la libreria ma anche le dipendenze della libreria stessa a cui fa riferimento.
Problemi

- Può non essere banale tenere aggiornato un virtualenv/pipenv
- Source distribution vs. Wheel (egg)
- Moltissime duplicazioni virtualenvwrapper
- Sistemi più generali, cross-platform: conda

Utilizziamo il versionamento semantico: uno standard per dare i nomi di versioni. Si basa su 3 token: MAJOR.MINOR.PATCH

- MAJOR cambia quando ci sono cambiamenti incompatibili nelle API
- MINOR cambia con nuove funzionalità (backwards-compatibile)
- PATCH solo bugfix

L'uso di questo meccanismo aiuta i package manager e permette di fare cose più sofisticate.

8 Sistemi di build automation

Costruire 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

8.1 Make

Make (1977) è nato per ottimizzare tempi di compilazione. Permette di specificare dipendenze fra processi di generazione.
Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
helloworld.o: helloworld.c
        cc -c -o helloworld.o helloworld.c
helloworld: helloworld.o
        cc -o \$@ \$<
.PHONY: clean
clean:
    rm helloworld.o helloworld
```

Le regole (o ricette) del make file sono composte da:

- target (in blu prima dei :)
- sorgente (dopo i :)
- comando

.PHONY è un nome fasullo di target che mi serve per poter lanciare il comando corrispondente tramite make clean.
Make è un gestore di dipendenze a livello di processi. 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 (ricetta) sono eseguiti seguendo l'ordinamento topologico.
Nei make moderni è possibile eseguire processi di generazione indipendenti in parallelo (make -j).
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.
Ad esempio in C si hanno problemi di non portabilità dovuti a funzioni che non esistono ovunque, hanno nomi/prototipi/comportamenti diversi ecc...
Le soluzioni adottate sono:

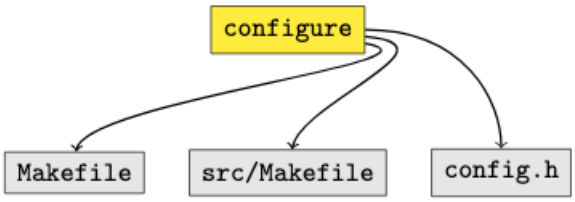
- #if/#else
- substitution macros

- substitution functions

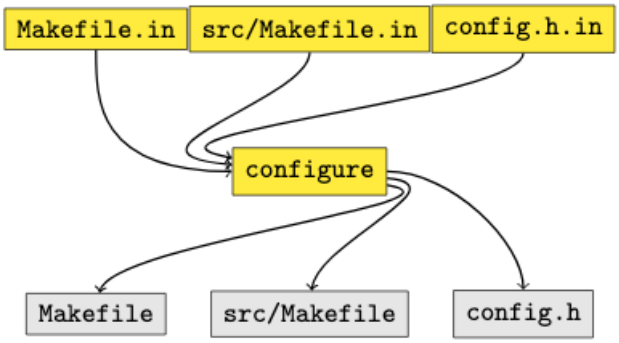
```
#if !defined(CODE_EXECUTABLE)
    static long pagesize = 0;
#if defined(EXECUTABLE_VIA_MMAP_DEVZERO)
    static int zero_fd;
#endif
    if (!pagesize) {
#if defined(HAVE_MACH_VM)
        pagesize = vm_page_size;
#else
        pagesize = getpagesize();
#endif
#if defined(EXECUTABLE_VIA_MMAP_DEVZERO)
        zero_fd = open("/dev/zero", O_RDONLY,0644);
        if (zero_fd < 0) {
            fprintf(stderr, "trampoline: Cannot open /dev/zero!\n");
            abort();
        }
#endif
    }
#endif
```

8.2 Configure

Il passo successivo è autoconf: è uno strumento che serve a limitare i difetti di make (si usa con make). Autoconf è una grande repository di regole tutte diverse. Produce uno script che si chiama configure, che non è altro che un tool di configurazione che prepara l'ambiente di sviluppo in modo da poter eseguire la build in un sistema diverso da quello in cui è stato creato, modificando il makefile.



Con autoconf quindi produco uno script configure che se faccio andare prima di make, modifica Makefile nella maniera adeguata che possa poi essere compilato dal sistema. Make da solo è legato ad un ambiente fisso, uno scenario in cui tutte le persone che compilano il mio programma sullo stesso sitema, make funziona benissimo. Ho bisogno quindi che adatti la build all'ecosistema di sviluppo in cui mi trovo. Make gestisce quindi le dipendenze fra i processi di generazione, autoconf adatta le ricette all'ambiente di sviluppo in cui sto facendo la build.



Autoconf e make fanno insieme un sistema di build automation. La build è composta da:

- gestione delle dipendenze
- processi di generazione (make)
- riconfigurazione all'ambiente di build

8.2.1 Ant

In Java la build automation è organizzata tramite Ant. Ant è basato su XML.

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```

Prendiamo ad esempio il target "compile" le sue entità. Le entità sono l'equivalente dei comandi, infatti prendiamo ad esempio mkdir: a differenza del makefile, non abbiamo il comando shell ma un'entità.

Evita di dover adattare i processi di generazione per più sistemi. Se ho ant su Linux e su Windows sono sicuro che mkdir funziona sia su Linux che su Windows. È pensato così per essere facilmente letto dalla macchina.

8.3 Gradle

È un sistema che usa un linguaggio di Java chiamato Groovy. L'obiettivo è descrivere i processi di generazione di un software. Gradle è una descrizione in cui si mette il meno possibile, devo descrivere le peculiarità del mio progetto. Non dico che il compilatore di Java si chiama javac perchè una cosa normale, ma se il compilatore si dovesse chiamare pippo, allora devo dire questa cosa. Supporta il test e fa anche della reportistica: non solo genera il software ma fa dei log che ci dicono cos'è stato fatto.

8.4 Prospettiva

- **Make:** l'ambiente di sviluppo è implicito: ho solo le dipendenze tra gli artifact sotto controllo.
  Tiene quindi traccia delle dipendenze tra i file di generazione e la limitazione è che assume un ambiente di sviluppo fisso. A make devo associare altre cose come autoconf, se voglio fare una vera build automation

- **Ant**: l'ambiente non è più dato per scontato ma costruisce i mattoni necessari ai processi di generazione. Nasce con in mente un ambiente di sviluppo che è quello di Java. Può sostituire gli elementi del processo di sviluppo con dei plugin. Non fa più l'assunzione di essere in un ambiente fisso

- **Gradle**: oltre a fare ciò che fa ant, permette di descrivere l'ambiente di sviluppo stesso. Quindi non solo ho la descrizione dei processi di generazione (come in make e ant), descrive anche sé stesso

9 Documentazione dei componenti

Strumenti che permettono di descrivere tutto il sapere implicito del singolo. Nell'ambito dell'ingegneria del software si parla di separation of concern: separazione/isolamento di problematiche diverse, in modo da riuscire a pensare ad una cosa alla volta e ad affidare una problematica ad un persona diversa.
Come si può suddividere il lavoro, senza la continua necessità di coordinazione? Perchè un sottogruppo di lavoro possa procedere in "isolamento" dovrebbe conoscere i componenti sviluppati da altri (o che altri svilupperanno), cioè conoscere il loro comportamento:

- in situazioni fisiologiche(correttezza)
- in situazioni patologiche(robustezza)

A questo scopo è quindi necessario specificare il funzionamento del sistema.
Abbiamo parlato di correttezza e robustezza, ma esattamente cosa sono:

Correttezza:

> "The degree to which a system or component is free from faults in its specification, design, and implementation"

Robustezza:

> "The degree to which a system or component can function correctly in the presenceof invalid inputs or stressful environmental conditions."

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.

Le specifiche costituiscono naturalmente l'interfaccia tra 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.

Perry & Evangelist (nel 1985) identificano una serie di"Interface Fault" cher imangono sostanzialmente comuni anche nei sistemi complessi di oggi.

- Construction (mismatchinterface/implementation)
- Inadequate functionality

- Disagreements on functionality
- Misuse of interface
- Data structure alteration
- Violation of data constraints
- Initialization/value errors
- Inadequate error processing
- Inadequate postprocessing (resource deallocation)
- Inadequate interface support
- Changes/Added functionality
- Coordination of changes
- Timing/performance problems

Introduciamo quindi il meccanismo dell'asserzione che verifica che le ipotesi confermino il reale funzionamento del componente che sto sviluppando.

Assert:

> "(1) a logical expression specifying a program state that must exist or a set ofconditions that program variables must satisfy at a particular point during programexecution. (2) a function or macro that complains loudly if a design assumption onwhich the code is based is not true."

In pratica l'asserzione è l'esplicitazione di un requisito sotto forma di codice.
È utile ragionare su "pattern" di asserzioni, spesso codificati in assertion languages/libraries.
Descrive un preprocessore (APP) per produrre asserzioni: il preprocessore lavora suspeciali "commenti" /*@@*/:

- assume
- promise
- return
- assert

```
int square_root(int x);
/*@
    assume x >= 0;
    return y where y >= 0;
    return y where y*y <= x
    &&  x < (y+1)*(y+1);
@*/

void swap(int* x, int* y);
/*@
    assume x && y && x != y;
    promise *x == in *y;
    promise *y == in *x;
@*/

void swap(int* x, int* y)
{
    *x = *x + *y;
    *y = *x - *y;
    /*@
        assert *y == in *x;
    @*/
    *x = *x - *y;
}
```

Queste asserzioni possono essere classificate in:

- Consistency between arguments
- Dependency of return value on arguments
- Effect on global state/Frame specifications
- The context in which a function is called
- Subrange membership of data/Enumeration membership of data
- Non-null pointers
- Condition of the else part of complex if (and switch)
- Consistency between related data
- Intermediate summary of processing

10 Docker

Con la crescita della dimensione degli oggetti da sviluppare e del numero di persone concorrenti, è nata la necessità di introdurre nuovi strumenti per poter comunque lavorare in maniera efficace. Quando si sviluppano software complessi, possono essere impiegati diversi:

- linguaggi (polyglot)
- librerie (versioni multiple)
- servizi (dbms, httpd, cache, log, ...)
- persone (onboarding)
- testing (fixtures)

Per risolvere questo problema si è pensato di adottare un approccio DevOps, scaricando alcune responsabilità sul developer. Si vuole:

- Dal punto di vista dei Dev:
  - predicibilità: eseguendo due volte lo stesso codice, sullo stesso ambiente di sviluppo, si ottiene lo stesso risultato.
  - replicabilità: spostandolo su un altro ambiente viene mantenuto il funzionamento.
  - versionamento: sistema di gestione delle configurazioni (il software funziona con un certo linguaggio, certe librerie, ecc.)
  - leggero: tutto questo dev'essere ottenuto attraverso un meccanismo efficiente e leggero.
- Dal punto di visto degli Ops:
  - tolleranza al fallimento: replicare le parti in maniera tale che se una parte smette di funzionare, le altre continuano ad andare.
  - portabilità: funzioni anche sul proprio server, oltre a quello di sviluppo.
  - scalabilità: possibilità di aumentare le risorse con l'aumentare dei clienti. Più risorse, più performance.
  - componibilità, attraverso moduli.
  - schedulazione & orchestrazione, in maniera automatica.

Possibili soluzioni sono:

- Configuration management: strumenti che permettono di dare una descrizione, in alcune casi imperativa, per configurare macchine hardware. Ansible, Chef, Puppet, Saltstack, Terraform.

- Virtualizzazione: strumenti che virtualizzano l'hardware, permettendo di avere più sistemi operativi sulla stessa macchina. LXC, KVM, Vagrant.
- Containers: strumenti basati sull'idea di conteinerizzazione. Docker, Singularity.

10.1 Virtualizzazione in stile Docker

In Docker si ha una virtualizzazione fortemente dipendente dal sistema operativo. Necessita, quindi, di supporto da parte del kernel per:

- separazione dei processi: permette la virtualizzazione dello spazio dei processi. Mantiene separati processi di gruppi diversi.
- supporto sul filesystem: una sola copia del filesystem che serve a più immagini diverse. Il kernel controlla che vengano allocate solo le modifiche da parte dei diversi container rispetto all'immagine standard (copy-on-write).
- networking: meccanismi di astrazione del networking. Container sulla stessa macchina o su macchine distinte possono lavorare insieme.

10.2 Immagini e Container

Un'immagine, nel gergo di Docker, è un grafo aciclico (DAG) di filesystem read-only e copy-on-write. Queste possono essere ottenute attraverso: pull (da un Docker Hub, su cui è anche possibile effettuare una push), build o commit.

```
$ docker pull hello-world
```

Un container è ciò che si ottiene prendendo un'immagine read-only, tramutando il filesystem in read-write e mettendo in esecuzione un processo distinto. Un container si può ottenere:

- esplicitamente attraverso una create

  ```
  $ docker create --name dt-image hello-world
  ```

- implicitamente attraverso una run (= create + start)

  ```
  $ docker run hello-world
  ```

Comandi per la gestione delle immagini:

```
docker images (elenco immagini), docker inspect, docker rmi (rimuovere immagine),
docker pull, docker push, docker export, docker import, docker create, docker run
```

Comandi per la gestione dei container:

```
docker ps [-a] (elenco container; l'opzione -a visualizza anche i processi terminati),
docker inspect, docker rm (rimuovere container), docker start, docker stop,
docker kill, docker pause, docker unpause
```

In che modo è possibile produrre un'immagine? Di seguito due possibili approcci:

- worst practice: eseguire un'immagine esistente, fare delle modifiche ed infine effettuare una commit (registrandola con una tag simbolica). Questo approccio non è riproducibile.
- best practice: utilizzare un processo di build che si basa su un Dockerfile, un file che specifica i passi di creazione e di configurazione dell'immagine.

È poi ragionevole rendere l'immagine pubblica. Per farlo, ci si può registrare su Docker ed inviare l'immagine a Docker Hub utilizzando il comando docker push, che pusha solo i nodi che non sono già noti e tutto il necessario per poter porre in esecuzione l'applicazione in un secondo momento.

Un container Docker può essere posto in esecuzione in diversi modi:

- ephemeral

  ```
  $ docker run --rm busybox date
  ```

  Eseguo un container specificando il comando da eseguire al suo interno. L'opzione –rm indica che quando il container termina viene eliminato.

- attached

  ```
  $ docker run -it busybox
  ```

  Eseguo un container allocando una TUI in cui poter eseguire i comandi.

- daemonized

  ```
  $ docker run --name dt-daemon -d busybox \
      sh -c 'while true; do date; sleep 10; done'
  ```

  Eseguo un container in background. Usualmente nei container vengono messi in esecuzione dei server, che rimangono costantemente in esecuzione, per cui questo è uno dei modi più ragionevoli.

Ho diversi modi per poter porre in esecuzione qualcosa e ritornare in contatto una volta staccato:

- ottenere i log delle istanze in esecuzione e terminate (metodo standard).

```
$ docker logs dt-daemon
```

- attaccarsi allo stdin/out del (processo in un) container in esecuzione.

```
$ docker attach --sig-proxy=false dt-daemon
```

- eseguire un altro processo in un container in esecuzione.

```
$ docker exec dt-daemon ps -a
```

10.3 Persistenza e comunicazione

Il modo più ragionevole per rendere persistenti le informazioni in Docker è l'utilizzo di un volume. Un volume è un'astrazione di un volume fisico (un piccolo filesystem).

```
$ docker volume create dt-volume
```

Il volume è nel filesystem della macchina ospite, per cui è molto difficile condividerlo tra container su macchine diverse. Può essere letto e scritto (anche con esecuzioni ephemeral) nel seguente modo:

```
$ docker run --rm --volume dt-volume:/data busybox sh -c 'date > /data/file.txt'
$ docker run --rm --volume dt-volume:/data busybox sh -c 'cat /data/file.txt'
```

L'opzione –volume monta il volume dt-volume nel mount point /data.

I comandi per la gestione dei volumi sono:

```
docker volume ls, docker volume rm, docker volume inspect
```

Per poter operare sul filesystem host, si può, anche se sconsigliato, mappare una directory del filesystem host (quello su cui viene eseguito Docker) all'interno del container.

```
$ docker run -it --rm -v $(pwd):/data busybox sh -c 'date > /data/file.txt'
```

Esiste inoltre il comando cp per copiare i file dai volumi di Docker al filesystem e viceversa. Questo ha come target un container.

```
$ CID=$(docker create --volume dt-volume:/data busybox)
$ docker cp $CID:/data/file.txt file2.txt
$ docker cp file2.txt $CID:/data/file3.txt
$ docker rm $CID
$ docker run --volume dt-volume:/data busybox cat /data/file3.txt
```

Per porre in comunicazione container diversi si utilizzano le reti, con cui creiamo un segmento di rete al quale diamo un nome e in cui mettiamo dei container.

```
$ docker network create dt-network
$ docker run --network dt-network --name dt-nc --rm -it busybox nc -l -p 80
```

Il nome che ha un container sulla rete è il suo stesso nome. È possibile comunicare con un container a partire da un container sulla stessa rete.

```
$ docker run -it --rm --network dt-network busybox sh -c 'date | nc dt-nc 80'
```

Per rendere pubblico un servizio di rete in un container, è possibile pubblicare una porta (-p) nel seguente modo:

```
$ docker run --name dt-nc --rm -it -p 80 busybox nc -l -p 80
```

I comandi per la gestione delle reti sono:

```
docker network ls, docker network rm, docker network inspect
```

10.4 Docker dal punto di vista dei developer

Dal punto di vista del developer possiamo:

- utilizzare Docker per testare la compilazione/esecuzione con development kit diversi, in maniera veloce.
- utilizzare l'IDE all'interno di un container, mettendolo in esercizio con tutto quel che serve e velocizzando quindi l'onboarding.

11 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.
Tentativo di avere progettazione e scrittura con lo stesso linguaggio (What e How con lo stesso linguaggio).
Il contratto è la descrizione di cosa voglio che mi permette di controllare il funzionamento del sistema.

11.1 Tripla di Hoare

L'idea di contratto si basa su un concetto più antico: la tripla di Hoare. Quando si parla di questa tripla si fa riferimento a degli articoli (scritti da Hoare e Floyd) in cui si cercava di definire una logica formale assiomatica che permettesse di dedurre assiomaticamente la semantica di un programma a partire dalla semantica delle singole istruzioni. Con un sistema del genere non si riesce a dimostrare la correttezza del programma, ma si riesce a descrivere in maniera dichiarativa che cosa fa.

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

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 sse vale la proprietà precedente.
La tripla di Hoare PSQ 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 uno 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.

11.2 Eiffel

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

Altri linguaggi di programmazione offrono delle librerie per applicare il design by contract, ma queste, tuttavia, sono solo delle approssimazioni.

```
feature
 decrement
 -- Decrease counter by one.
     require
      item > 0 -- pre-condition
     do
      item := item - 1 -- implementation
     ensure
      item = old item - 1 -- post-condition
     end
```

:= assegnamento in eiffel (equivalente a = in java)
= predicato logico in eiffel (equivalente a == in java)

Eiffel è esplicitamente progettato come linguaggio "di progetto", non solo "di programmazione":

> "specify, design, implement and modify quality software"

"Programmazione in grande" (programmo un sistema enorme che non può essere pensato e realizzato da una sola persona) con oggetti, derivati da classi organizzate in gerarchie di ereditarietà e raggruppate in cluster, che forniscono feature (command o query) ai loro client. Ogni singola feature è una S di cui posso specificare P e Q.
La violazione del contratto diventa un modo per stabilire un errore nell'accordo o un difetto nell'implementazione.
In un senso formale, si può parlare di design by contract solo con Eiffel.
I contratti, se usati bene, possono diventare lo strumento con il quale gli sviluppatori si accordano.
Per scrivere dei contratti scriveremo del codice, ovvero dei metodi. Le pre e post condizioni non sono fatte per aggiungere funzionalità, ma servono solo per asserire e per verificare lo stato del mondo prima o dopo l'esecuzione della feature.
Le **feature** sono attributi e metodi della classe. Possono avere una visibilità (private, public) che viene definita come feature NONE o feature ANY.
Le asserzioni possono essere suddivise in:

- **pre/post-condizioni** sulla feature: *require, ensure*
- **invarianti di classe**: *invariant*
- **asserzioni**: *check*
- **inviarianti di ciclo**: *from ..... invariant ..... until .... variant .... loop....end*

L'**invariante di una classe** sono condizioni che devono essere vere in ogni momento "critico", ossia osservabile dall'esterno. È una proprietà che tutti gli oggetti di quella

classe mantengono durante la loro vita.
Ad esempio: per la classe persona che ha una feature età, un'invariante di classe potrebbe essere età maggiore di zero. Si possono tradurre in pre e post condizioni che devono valere sempre per tutti i metodi, ad eccezione dei costruttori: nel costruttore l'invariante deve valere solo dopo la creazione dell'oggetto (poichè prima non c'era e l'ho dovuto creare).
Nell'invariante si può fare tutto, l'importante è che il risultato sia un valore logico. Di solito non conviene mettere if e else, ma si usano gli implies.
L' **invariante di ciclo**: rende decidibile un ciclo (posso sapere quando termina).
È buona norma dare sempre nomi alle invarianti, alle pre e post condizioni per risalire più facilmente alla causa dell'errore.

Con Eiffel è possibile avere un supporto run-time alle violazioni: se una condizione non vale viene sollevata un'eccezione (al posto che avere il catch come in Java abbiamo la rescue). L'eccezione porta il sistema nel precedente stato stabile ed è possibile:

- terminare con un fallimento
- riprovare

Spesso potrebbe sembrare di scrivere le stesse cose due volte:

```
do
 balance := balance - x
-----------
ensure
 balance = old balance - x
```

Una è l'implementazione, l'altra è la specifica; una mi serve per dire cosa voglio, l'altra per dire come lo voglio.
Il client è responsabile delle precondizioni, il fornitore di postcondizioni e invarianti.
Old indica lo stato dell'oggetto prima dell'esecuzione.

In Eiffel ci sono altri operatori logici:

- *implies*: equivale a not a or a and b. se a, b, sennò qualsiasi cosa. ??????
- *and* e *or* non cortocircuitate.

In Eiffel ci sono le classi astratte(*defer*).
Nelle pre-condizioni non si possono usare feature private, nelle post-condizioni sì, siccome le pre-condizioni le deve poter valutare il client.

11.3 Eiffel Studio

Eiffel Studio è l'unico IDE integrato con il linguaggio e con l'idea di design by contract. Esistono diverse viste che mostrano ognuna il programma da prospettive diverse:

- Text è quella che si usa quando si deve programmare. Eiffel ragiona sui file e questa è la visione per file.
- Contract è quella che si usa per ragionare a contratto. Fa vedere solo le feature senza implementazione (che in questo caso è irrilevante)
- Flat fa vedere tutte le feature, anche quelle ereditate
- Interface è la visione flat, ma solo dei contratti
- Clickable mostra i diagrammi delle classi (?)

11.3.1 Contratti ed ereditarietà

Il principio di sostituzione di Liskov stabilisce che, perchè 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.

Principio di sostituibilità: il compilatore dovrebbe dimostrare automaticamente che

$$PRE_{parent} \Longrightarrow PRE_{derivata}$$

$$POST_{derivata} \Longrightarrow POST_{parent}$$

1. in un programma corretto non può succedere che PRE_{parent} valga e $PRE_{derivata}$ no; l'oggetto evoluto deve funzionare in ogni stato in cui funzionava l'originale: non può avere obbligazioni più stringenti, semmai più lasche.
2. in un programma corretto non può succedere che valga $POST_{derivata}$ ma non $POST_{parent}$; un stato corretto dell'oggetto evoluto deve essere corretto anche quando ci si attende i benefici dell'originale.

Un modo per garantire che le due 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:

$$PRE_{derivata} = PRE_{parent} \vee P$$

$$POST_{derivata} = POST_{parent} \wedge Q$$

Eiffel introduce quindi un meccanismo sintattico che obbliga a rispettare il principio di Liskov. Il compilatore dovrebbe dimostrare che le pre-condizioni della classe base implicano quelle della derivata. Dualmente vale la stessa cosa sulle post-condizioni. Nel caso generale questo è indecidibile e impossibile. Eiffel costringe il programmatore a scrivere

solo cose dimostrabili.
Le pre-condizioni derivate sono sempre obbligatoriamente in OR con quelle del parent. Per fare questa cosa si utilizza la parola chiave *require else*, cioè sono le pre-condizioni di prima, altrimenti quelle di adesso. Stesso ragionamento duale vale per post-condizioni. Le post-codizioni derivate sono in AND quelle di prima (parola chiave *ensure then*).

Nell'esempio del programma ANIMAL visto a lezione:

- Animale mangia Cibo (is_a Cosa)
- Mucca (is_a Animale) mangia Erba (is_a Cibo)

Questa **covarianza** è contraria al principio di Liskov perchè restringe le precondizioni. La controvarianza (Mucca mangia Cosa, Sather) e l'invarianza (Mucca mangia Cibo, Java) vanno bene. Eiffel invece è covariante il che, impedendo un controllo di conformità statico, introduce parecchie complicazioni come ad esempio CATcall, runtime type identification...

11.3.2 Eccezioni

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.

> An exception is a run-time event that may cause a routine call to fail (contract violation). A failure of a routine causes an exception in its caller.

Abbiamo la possibilità di gestirle in due modi:

1. **Failure** (panico organizzato): pulisce l'ambiente, termina la chiamata e riporta il fallimento al chiamante
2. **Retry**: cerca di cambiare le condizioni che hanno portato all'eccezione e riesegue 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.

11.3.3 Correttezza

Per ogni feature (pubblica) f:

- $\{PRE_f \wedge INV\} body_f \{POST_f \wedge INV\}$
- $\{True\} rescue_f \{INV\}$
- $\{True\} retry_f \{INV \wedge PRE_f\}$

12 Stime dei costi

Stimare i costi è considerata un'attività molto complicata. I costi possono provenire da diverse fonti:

- strutture: uffici, energia
- risorse informatiche: hw e sw di sviluppo
- materiali di consumo: carta, dischi
- personale di supporto: segreteria, amministrazione
- personale tecnico: programmatori, tecnici, manager

Questi sono tutti costi di diverse entità (più grandi e più piccoli), che vengono poi ammortizzati. Alcuni costi sono fissi (non dipendono dal progetto), altri invece, come il personale tecnico, variano in base al progetto.
Il costo del tempo non è un costo assoluto, ma lo devo relazionare alla produttività del personale.
I costi sono influenzati dalla produttività, ma cosa influenza la produttività?

> "A multitude of factor influence developer productivity"

Nelson-Jones identifica 250 aspetti che possono influenzare la produttività di un team di sviluppo. Tra i principali:

- Numero di istruzioni da codificare: la dimensione e la complessità del progetto influenzano il tempo
- Capacità, motivazioni, coordinamento personale
- Complessità dell'applicazione
- Stabilità di requisiti: cambiare requisiti diminuisce la produttività, anche perché è lavoro inutile che non può essere fatturato
- Prestazioni o qualità richieste
- Strumenti di sviluppo usati
- ...

Dobbiamo quindi migliore la produttività, diminuire i tempi e i costi.
I modi con cui posso stimare i costi possono essere:

- Stime umane, in cui applico dei ragionamenti
  - Legge di Parkinson sul lavoro
  - Price to win

  - Esperti
  - Per analogia

- Stime automatizzate
  - algoritmiche: applicazione di qualche algoritmo
  - machine learning: tecniche di intelligenza artificiale in cui si passa in input una serie di dati e si spera che tiri fuori una stima più o meno azzeccata.

12.1 Legge di Parkinson

Il costo dipende dalle risorse disponibili. Come per i gas anche "Il lavoro si espanderà fino a riempire tutto il volume". In termini di stima dei costi vuol dire prendere ciò che ho disposizione e occuparlo tutto. È una visione pessimista, poiché serve tutto, ma allo stesso tempo ottimista, poiché so che basta. Una stima di questo genere, però, rende difficile fare una previsione ragionevole.

12.2 Price to win

Il costo è quanto il cliente è disposto a spendere (sia soldi che tempo). Vantaggio è che si ottiene il lavoro (facendo il prezzo migliore tra i concorrenti, prezzo che il cliente è disposto a pagare), svantaggio è che non si ha nessuna garanzia sul lavoro (potrei farlo male).
Potrebbe anche andare bene se è possibile ricontrattare i requisiti a posteriori e se è possibile lo sviluppo incrementale

12.3 Esperti esterni

Molto spesso quello che si fa è chiedere a esperti esterni. Si fa un primo studio di fattibilità esternamente alla ditta, per non perdere risorse e avere costi precisi. Non è una tecnica, ma una prassi comune.
Si delega la responsabilità ad altri per diversi motivi: sono più bravi, maggiore conoscenza del dominio, non tolgono persone interne da attività più importanti, ci danno il risultato entro una certa scadenza.
Ma come fanno gli esperti a fare le stime?

12.4 Stima per analogia

Si cerca di confrontarlo con un progetto precedente di cui la quantità di lavoro necessaria (mesi-uomo) e i tempi di sviluppo (mesi) sono noti. Si usano coefficienti per gli scostamenti (più grande di, maggiore esperienza di, riuso..). È un metodo top-down dove si prende il progetto nel suo insieme e successivamente si suddivide la spesa.

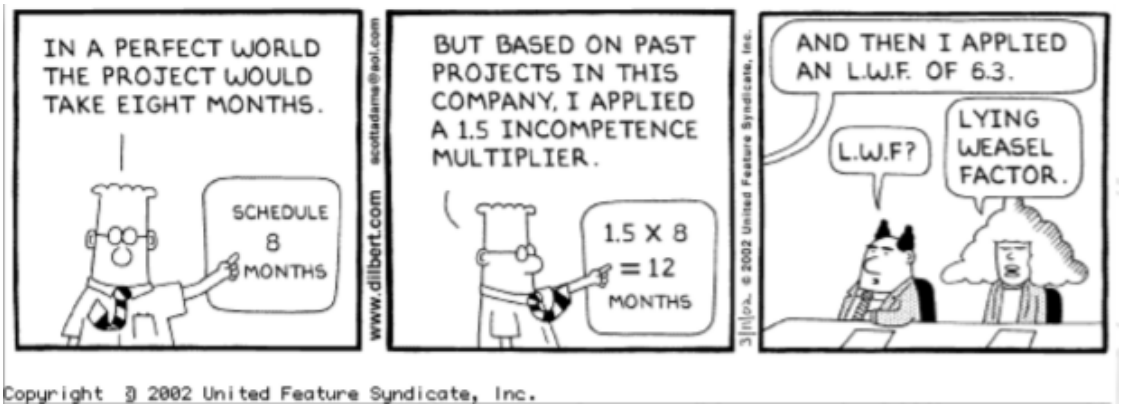


Figura 12: Legge di DeMarco-Glass: Most cost estimates tend to be low

Nel momento in cui usiamo analogia possiamo basare le nostre stime su diversi fattori:

- **Stima basata su dimensione del codice**: si basa sulla portata del progetto, ossia quante cose voglio che ci siano. Più cose ci sono, più devo scrivere. Piuttosto che parlare di righe di codice (che possono essere imprecise per diversi fattori), sarebbe più utile parlare di funzionalità implementate. Quindi si scompone funzionalmente il progetto e si identificano i componenti da sviluppare. Di questi si effettua una stima della dimensione. È una tecnica bottom-up: ricomponiamo la spesa totale a partire dalle singole funzionalità. Maggiore è la scomposizione (e maggiore è il dettaglio in cui si scende), minore è la difficoltà di stima dei singoli componenti
- **Stima basata su quantità di lavoro**: si scompone il progetto secondo una logica operativa (per fasi) e funzionale (componenti), si stima il tempo necessario per svolgere le singole fasi per il determinato componente, tecnica bottom-up.

Questi sono dei metodi umani, se vogliamo cercare di automatizzare dobbiamo cercare di trasformare in numeri:

- **Produttività**: LineeCodice/MeseUomo
- **Qualità**: Errori/LineeCodice
- **Costo Unitario**: CostoTotale/LineeCodice

Queste possono essere misurate su progetti passati, posso quindi avere una base di conoscenza per ragionare sul futuro.
Tra le principali tecniche di stime automatizzate (algoritmiche) troviamo:

- Bohem: COCOMO, COCOMO2
- Putnam: SLIM (Software Life Cycle Management)

12.5 COCOMO

COCOMO è una tecnica degli anni 80, sviluppata quando il software veniva scritto da aziende organizzate col modello a cascata. Al giorno d'oggi non rispetta quella che è stata l'evoluzione del software, per cui c'è stata anche una sua evoluzione: COCOMO 2. COCOMO assume un modello cascata a 4 fasi sequenziali:

1. pianificazione e analisi requisiti
2. progetto
3. sviluppo
4. integrazione e testing

Parte dalla misurazione di casi reali e cerca di estrarre delle formule che vadano a mappare quelli che sono i risultati. Stima i nuovi input, applica la formula e ottiene i risultati.

COCOMO caratterizza i progetti in base a una dimensione e ad ogni dimensione corrisponde una tipologia di formule. Le applicazioni possono essere:

- **semplici**: piccole applicazioni per cui il team è già ferrato e sa quali sono le problematiche che possono presentarsi
- **intermedie**: applicazioni in cui ci possono essere parti di codice in cui non ho nessuna esperienza e posso fare scelte (anche architetturali) sbagliate, accorgendomene tardi
- **complesse**: vincoli real-time, hardware, specifiche... Fattori esterni che complicano la scrittura dell'applicazioni

E a partire da queste COCOMO fornisce tre modelli:

- **Base**: dipende solo dalla stima delle linee di codice
- **Intermedio**: aggiunge 15 fattori correttivi legati alle caratteristiche del progetto e permette di effettuare una stima dei singoli componenti
- **Avanzato**

12.5.1 Modello base

MesiUomo = $aLOC^b$
MesiSviluppo = $cMesiUomo^d$
dove *a, b, c, d* sono parametri che variano in base alla complessità dell'applicazione (con b ¿ 1, d ¡ 1)

Parametro	Semplice	Intermedia	Complessa
a	2.40	3.00	3.60
b	1.05	1.12	1.20
c	2.50	2.50	2.50
d	0.38	0.35	0.32

Al crescere del progetto ha senso inserire nuove persone e impiegarci meno tempo.

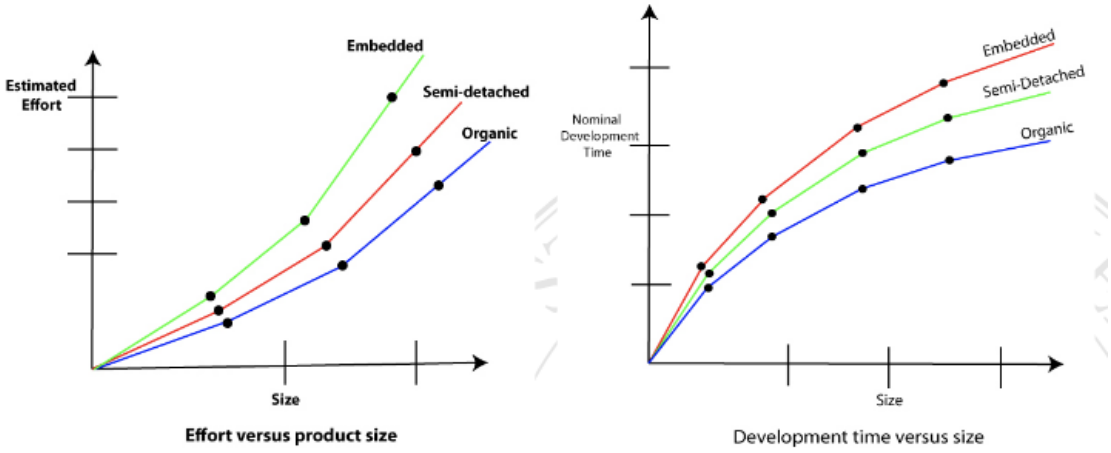


Figura 13: COCOMO andamenti

Con effort (MesiUomo) e development time (MesiSviluppo) ottengo una nuova misura: il dimensionamento del team. Quante persone devo metterci a lavorare?

$$\text{Dimensione} = \frac{MesiUomo}{MesiSviluppo}.$$

COCOMO non vede quindi il numero di componenti del team come fattore su cui giocare per accelerare il progetto, ma come misura derivata.

12.5.2 Modello intermedio

Aggiunge 15 fattori correttivi da considerare:

- Legati al prodotto
  - affidabilità richiesta
  - dimensione database
  - complessità
- Legati alle macchine
  - efficienza richiesta
  - memoria richiesta
  - variabilità ambiente sviluppo
  - tempi di risposta

- Legati al progetto
  - tecniche di programmazione avanzate
  - strumenti avanzati
  - tempi di sviluppo stretti
- Legati al personale
  - esperienza analisi
  - esperienza di dominio
  - esperienza sviluppo

MesiUomo = $aLOC^b \Pi c_j$
MesiSviluppo = $cMesiUomo^d$

dove Πc_j è la produttoria dei coefficienti correttivi e a,b non sono più quelli di prima. LOC dovrebbe essere calcolato a partire dai componenti e non solo dalla stima totale. Questo consente di ottenere risultati più fini.

12.6 COCOMO 2

Tiene conto dei nuovi modo di sviluppare software (diversi dal modello a cascata) e usa formule diverse. In particolare tiene conto di:

- Prototipizzazione
- Riuso
- COTS (commercial off the shell)

Ha diverse formule a seconda del momento della stima:

- prime fasi e prototipizzazione
- fase di progettazione
- fase successiva alla definizione dell'architettura

12.7 Stime automatizzate: Machine Learning

Diamo in pasto ad un algoritmo una grande quantità di dati anziché fare un fitting su una funzione, che è una grossa approssimazione, ed è il motivo per cui si vanno ad inserire tutti i fattori correttivi.
Queste tecniche sono più efficienti avendo però una base di dati sufficiente.

- Neural Networks
- Fuzzy Logic

- Case-Based Reasoning
- Analogy
- Rule-Based
- Regression Trees
- Hybrid System

12.8 Stime collettive/condivise

Fino ad ora si è parlato di un solo stimatore, ma nella pratica per diversi motivi si cerca di ottenere delle stime:

- collettive: tanti individui che decidono separatamente danno una stima migliore di quella di un singolo esperto, perché diverse persona lasciano sfuggire meno considerazioni
- condivise: si fanno mettere d'accordo le diverse stime ed iterativamente si arriva ad una stima più o meno condivisa.

Nei modelli agili la responsabilità della stima passa dai manager (imposizione dall'alto) a team sviluppatori (autodeterminazione). Il risultato finale dev'essere condiviso e accettato da tutti gli elementi del team, indipendentemente da chi la farà.
Abbiamo diverse tecniche che si occupano di stime condivise e collettive:

- Planning Poker
- Team Estimation Game
- #NoEstimate

Ci sono due problemi principali che un metodo di stima condivisa deve affrontare:

- effetto folla, si perde tempo per troppa comunicazione
- effetto ancora, quando arrivano i primi pezzi di informazione (ad esempio stime da un'altra persona), non si riesce più a ragionare per termini assoluti, ma per termini relativi a quel primo elemento.

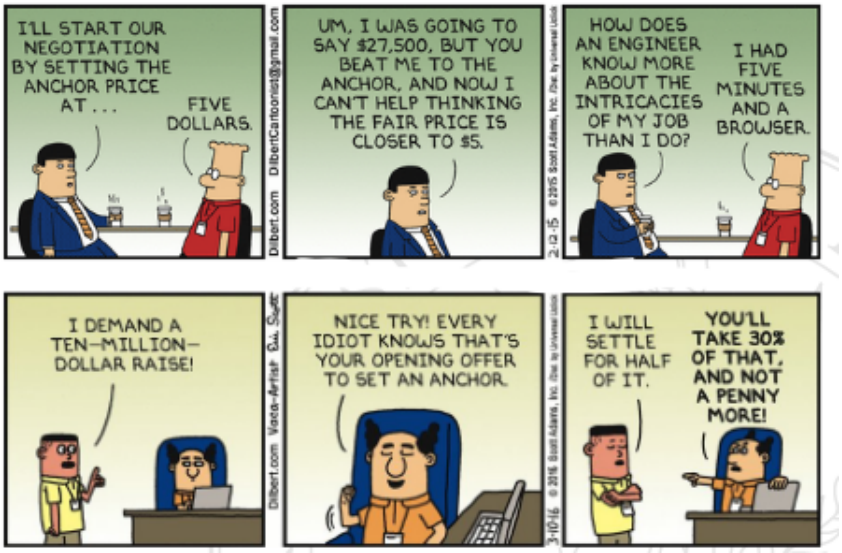


Figura 14: Effetto ancora

12.8.1 Planning Poker

1. Si presentano brevemente le carte
2. Il team può fare domande, richiedere chiarimenti e discutere per chiarire assunzioni e rischi
3. Ogni componente sceglie una carta del poker rappresentante la propria stima. L'unità di misura di queste carte può variare e sicuramente non è lineare. Le unità di misura sono unità di misura ideali, sono numeri, non grandezze dimensionate.
4. Le carte vengono girate contemporaneamente (per eliminare l'effetto ancora)
5. Chi ha espresso le stime più basse e più alte ha tot tempo per spiegare le sue motivazioni e cercare di convincere gli altri.
6. Si ripete la votazione fino a quando si raggiunge l'unanimità. In 3-4 iterazioni, la cosa dovrebbe convergere. Se non converge, vuol dire che la fase di chiarificazione iniziale non è stata sufficiente. A questo punto il coordinatore gestisce la cosa in qualche modo.

Nelle metodologie si parla sempre e solo di storie. Non si vende un prodotto, si vende tempo. Rendo il più produttivo possibile quel periodo di tempo per far quelle cose che per il cliente sono importanti. Poi o si interrompe e si consegna (ad ogni consegna c'è qualcosa che ha valore per l'utente), o si va avanti a fare un'altra cosa, vendendo altro tempo.
Questa tipologia di contratto ha più probabilità di non fallire, rispetto ad un contratto a prodotto. È più difficile contestarlo.

12.8.2 Team Estimation Game

Composto da 3 fasi coglie di più gli aspetti critici di ancoraggio e comunicazione.
È molto più efficiente per stimare tante carte.

1. **Valutazione comparativa**: ad un tavolo si mettono in fila i developer. C'è una pila delle stories. Il primo prende la prima carta, la legge ad alta voce, la piazza al centro del tavolo e a questo punto si mette in coda. Il (nuovo) primo prende la carta dalla pila e decide lui in autonomia se è più semplice, più complicata o equivalente a quella che c'è già sul tavolo, mettendola a sinistra, destra o sotto. Il prossimo può prendere una nuova carta e posizionarla rispetto a tutto il gruppo o tra due gruppi, oppure ancora sotto. Altrimenti può spostare una carta già posizionata, motivando. Se non ci sono carte in pila e non ce n'è nessuna che vuole spostare, passa. Itera finché non ci sono più carte e non c'è nessun developer che vuole spostare una carta. C'è un effetto ancora diverso, perché non è su un valore, ma su una relazione. Non è significativo. La comunicazione non è globale, ma è solo di colui che sposta.

2. **Quantificazione delle distanze**: di nuovo ci si mette in coda davanti al tavolo. Carte disposte a colonne. Ho un mazzo tipo quello del planning poker che ha i valori possibili. Queste carte sono ordinate. Si prende la prima (di solito 2, per avere margine sotto). Ognuno pesca una carta e decide su quale colonna metterla. Arriva un altro dev e può o prendere una carta e metterla su una colonna (alcune colonne possono anche essere saltate), oppure propone uno spostamento in una nuova posizione di una delle carte oppure ancora, se va tutto bene, passa. Si ripete finché son finite le carte. Si può anche decidere che non ci sia alcuna storia che corrisponda a quel valore. Le colonne senza valore vengono assimilate alla colonna alla loro sinistra. A questo punto ogni colonna ha un valore. Non ha ancora una scala effettiva.

3. **Scala assoluta**:Prendo una delle carte a cui ho dato valore più semplice (2) e stimo a quante ore/uomo corrisponde. Dopodiché si fa una proporzione con tutte le altre. A questo punto abbiamo trovato una scala.

La terza fase dopo la prima volta che si esegue team estimation game si perde.
Si lavora, avendo iterazioni corte, basate sul concetto di velocity: capacità osservata nell'iterazione precedente di completare lavori da parte del team.
Nell'iterazione dopo si danno le stime con gli stessi valori, non importa quale sia la scala.
Nell'iterazione successiva posso anche adattare le stime fino a convergere con quella che è la velocità effettiva delle persone.
Ogni team ha un suo numero che non è confrontabile con quello stimato dagli altri team.
Non dev'essere usato come metodo di valutazione, poiché possono corrispondere a scale diverse.
Non si considerano storie non finite.
Non deve nemmeno essere imposta dall'esterno.

12.8.3 #NoEstimate

Nei metodi agili non c'è un commitment totale, è un'indicazione di ciò che riesco fare. È ammesso sbagliare, si adatta alla velocity pian piano.
Non essendo così rigidi, perchè non eliminare la stima? Non è particolarmente importante per l'utente. Non perdiamo tempo a fare stime.
Questo funziona per team maturi perché splittando le storie, si ottengono storie della stessa dimensione, aiutando l'omogeneità. In un certo senso si fanno stime in maniera inconscia.

13 Retrospettive agili

"Un ritrovo rituale di una comunità alla fine di un progetto per riguardare gli eventi e imparare dall'esperienza. Nessuno ha conoscenza dell'intero progetto, ogni persona ha solo un pezzo della storia. Questo rito collettivo ci aiuta a raccontare la storia e a estrarre esperienza per la saggezza."

Il concetto di retrospettiva è presente in tutte le metodologie agili. Questo è il mezzo principe con cui il team ragiona su cosa sta accadendo e su come potersi migliorare. Finita l'iterazione classica, prima di proseguire, occorre fare una retrospettiva (relativamente di breve durata), al fine di adattarsi a ciò che è successo nell'iterazione precedente:

- capire diversi punti di vista
- cercare di forzare un modo di pensare
- favorire una discussione su cosa occorre fare e focalizzarsi su azioni specifiche (non generiche)

Le retrospettive si compngono di 5 fasi che si integrano con l'iterazione classica delle metodologie agili:

- setup
- gather data (raccogliere dati)
- make insights (generare comprensioni migliori)
- decide what to do
- close

13.1 Setup

Si introducono al gruppo lo scopo, l'argomento e il tempo a disposizione. Mira a creare un'atmosfera dove le persone si sentono a loro agio a discutere le questioni.

Le attività svolte in questa fase sono:

- checkin, fare una domanda semplice a cui ognuno deve rispondere con una o due parole (come ti senti rispetto a questa retrospettiva? arrabbiato, felice, ottimista, triste, preoccupato), cominciando in questo modo a raccogliere l'umore del gruppo.
- working agreements: mettersi d'accordo su alcune regole. Non troppe, ma ben chiare, che possono cambiare da una riunione all'altra. Si divide in gruppi e ognuno propone delle regole e dei comportamenti da seguire. Quando si discute, bisogna assumere che ognuno abbia fatto il massimo.

Un'altra possibile attività è ESDP, in cui ciascuno si deve caratterizzare, in maniera anonima, rispetto a ciò di cui si sta parlando come:

- esploratore: ha voglia di capire
- shopper: attende che esca qualcosa di utile
- in vacanza: finge di far presenza
- prigioniero: avrebbe preferito fare altro, ma è stato costretto a venire

13.2 Gather data

Non tutti hanno visto tutto. Bisogna creare un vista collettiva di cosa è successo, considerando due tipi di dati:

- hard data
  - eventi: riunioni, decisioni importanti, assunzione/licenziamento di membri del team, adozione di nuove tecnologie, ecc...
  - metriche: dati come velocity, numero di segnalazioni di bug, quanto refactoring è stato fatto, schema che indica a che punto si è del progetto, ecc..
- soft data: sentimenti, emozioni, umore. Quello che uno ritiene importante o che ha colpito. Non serve che tutti lo considerino importante, ma il fatto che molti non lo considerino importante potrebbe essere un dato aggiuntivo.

Possibili attività svolte in questa fase sono:

- timeline: funziona bene per rilevare hard data. Ognuno propone con delle carte eventi hard per lui significativi e poi vengono posizionate su una timeline in ordine cronologico. Dopodiché ognuno riguarda tutti gli eventi per vedere ciò che hanno scritto gli altri e cogliere ciò che è successo e di cui non ci si è resi conto. Questo può essere utilizzato anche per le emozioni, ma è meglio separare le due cose.
- color code dots: dopo aver raccolto hard data, vengono associati soft data mediante una votazione, su come ci si sente rispetto a quegli eventi. Ogni partecipante riceve un certo numero di gettoni colorati da posizionare sulle schede. Colori che rappresentano le emozioni o l'aspetto che si vuole mettere in evidenza.

13.3 Make insights

Ci si domanda "perchè?" per riuscire a trovare comprensioni maggiori di quello che è accaduto, elaborando i dati e traendone informazioni. Trovare situazioni ricorrenti, cambiamenti improvvisi di umore. È importante affrontare questa riunione con la mente molto aperta, per non lasciarsi condizionare dai propri pregiudizi.

Possibili attività svolte in questa fase sono:

- Patterns & Shifts: si discute lo schema ricavato precedentemente e si cercano connessioni tra eventi, situazioni ricorrenti, similitudini e cambiamenti improvvisi.
- Five Whys: serve a identificare diverse cause più profonde. Ci si divide a coppie, si analizza, ci si chiede reciprocamente il perché più volte (5), andando a cercare le cause prime, e dopodiché si uniscono. Il fatto di farlo a coppie e ricongiungere le questioni, fa emergere più effetti scatenanti, non fermandosi alla prima causa.

Possono essere analizzate anche le cose positive, non solo quelle negative, cercando di capire perché qualcosa ha funzionato per far in modo che la volta successiva funzioni ancora.

13.4 Decide what to do

Sui temi identificati precedentemente si cerca di trovare delle idee di intervento, da adottare nel prossimo progetto o nella prossima iterazione, selezionando quelle più importanti e creando dei piani di sperimentazione, preferibilmente misurabili.

Possibili attività svolte in questa fase sono:

- triple nickels: si sviluppano idee raccogliendo diversi contributi. In maniera circolare ognuno (3 persone) scrive la propria idea su un foglietto e lo passa alla persona successiva. Questa la guarda ed aggiunge qualche particolare, e così via. In questo modo si affronta da diversi punti di vista la stessa idea, in diverse fasi di elaborazione. Le idee devono poi essere giudicate e votate.
- SMART goals: le idee venute fuori per essere pianificate devono essere trasformate in SMART goal che soddisfano queste caratteristiche:
  - Specific: non generico
  - Measurable: e quindi controllabile
  - Attainable: effettuabile
  - Relevant: significativo
  - Timely: opportuno... adesso

13.5 Close

- Occorre decidere come documentare quanto fatto (fare foto alla lavagna, raccogliere foglietti, ecc...)
- Fare una breve retrospettiva della retrospettiva stessa. Cosa è piaciuto? Cosa ha funzionato? Per migliorarsi.
- Ci deve essere un momento per fare apprezzamenti, per finire in maniera positiva.

13.6 Logistica

Quanto dura?

Non ha una durata fissa. Di solito tra le 2 e le 4 ore, ma può durare da mezz'ora a 4 giorni (magari nel caso di un progetto biennale), con le seguenti percentuali:

- Set the stage: 5%
- Gather data: 30-50%
- Generate insights: 20-30%
- Decide what to do: 15-20%
- Close the retrospective: 10%
- Shuffle time: 10-15%

Dove si svolge?

Svolgerla nell'ambiente di lavoro permette di mantenere il legame con il lavoro, avendo già a disposizione i dati. Andare da un'altra parte consente di distaccarsi dall'iterazione stessa e ragionare da un punto di vista fresco, soprattutto se quell'ambiente ricorda problemi e fallimenti.

Riferimenti bibliografici

[1] Fred Brooks. *The mythical man-month.*

[2] Mattia Monga. *From Bazaarto Kibbutz:How Freedom Deals with Coherence in the Debian Project.* URL: `https://homes.di.unimi.it/monga/lib/oss-icse04.pdf`.

[3] Eric Steven Raymond. *The Cathedral and the Bazaar.* URL: `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`.