Appunti di Sviluppo Software in Gruppi di Lavoro Complessi

Caccaro Sebastiano
A.A.2019/2020

Contents

1 Modelli Organizzativi **2**
- 1.1 Modelli a Cattedrale e sala Operatoria . . . 2
  - 1.1.1 Critiche . . . 3
- 1.2 Modello a Bazaar . . . 3
- 1.3 Modello a Kibbutz . . . 4
  - 1.3.1 Esempio di Debian . . . 5

2 Metodologie Agili **6**
- 2.1 Principi agile . . . 6
- 2.2 Principi agile nella pratica . . . 7
  - 2.2.1 Enfasi sul testing . . . 8
- 2.3 Scrum . . . 8
  - 2.3.1 Pianificazione . . . 9
  - 2.3.2 Riunioni . . . 9
  - 2.3.3 Tecniche di lavoro . . . 10
    - 2.3.3.1 Pair Programming . . . 10
    - 2.3.3.2 Codice Condiviso . . . 10
    - 2.3.3.3 Refactoring . . . 10
    - 2.3.3.4 Test Driven Development (TDD) . . . 11
    - 2.3.3.5 Velocity Tracking . . . 11

3 Continuos Integration **12**
- 3.1 Capisaldi . . . 13
- 3.2 Configuration Managment . . . 14
  - 3.2.1 Artifact . . . 14
  - 3.2.2 Sincronizzazione . . . 15
  - 3.2.3 Lavoro concorrente . . . 15
  - 3.2.4 Implementazione . . . 16

4 Git **16**

1 Modelli Organizzativi

Lo sviluppo software presenta dei problemi intrinseci:

- **Non linearità del software**: Un errore molto piccolo può avere conseguenze catastrofiche
- Obiettivi poco chiari e mutabili

Questi problemi esistono tutt'oggi e sono difficilmente mitigabili. Esistono invece delle criticità che possono essere risolte.

> **Legge di Brooks**
> Aggiungere personale ad un progetto in ritardo lo farà solo ritardare.

Nello sviluppo software, non tutto è facilmente parallelizzabile. Non posso far nascere un bambino da 9 donne in un mese. Va da se che l'**effort** non corrisponde al **progress**. È molto facile stimare quanto si è lavorato, è meno facile misurare di quanto si è progredito, e questo può causare ulteriori ritardi. La soluzione non è aggiungere personale.

Un progetto deve mantenere sempre la sua **integrità concettuale**.
Per far ciò Brooks propone vari modelli.

1.1 Modelli a Cattedrale e sala Operatoria

- **Cattedrale**: Tenere rigorosamente separata progettazione e implementazione. L'implementatore deve quindi curarsi solamente di seguire quanto progettato. Si mantiene così la visione originale del progetto. Questo modello ha però il difetto di essere molto poco flessibile, e difetti nel progetto comportano problematiche enormi.
- **Sala operatoria:** Solamente il chirurgo (superbravo) si occupa di fare le cose importanti, gli altri nella sala fanno praticamente solo da assistente. Il vero lavoro viene svolto solamente dal chirurgo (una sola persona).

Questi modelli fanno però due grosse supposizioni:

- Che sia possibile accentare lo sforzo creativo in un'unica persona.

- Che sia possibile separare completamente progettazione e implementazione.

La maggior parte delle volte, tuttavia, queste supposizioni non si rivelano corrette.

1.1.1 Critiche

Eric Raymond contrappone il modello a **bazaar** (usato per lo sviluppo di Linux) contro la cattedrale di Brooks, osservando che il modello open source di Linux produca software di qualità, pur non usando i modelli proposti da brooks.

8 Ottobre 2019

1.2 Modello a Bazaar

Usato in Linux, si identifica esplicitamente come l'antitesi del modello a cattedrale. Raymond condivide l'analisi di Brooks, ma arriva alla conclusione che ci possano essere altri modelli in certe situazioni.

> **Baazar**
> Mercato autogestito, dove chiunque può mettere una bancarella dove vuole, quando vuole

Figure 1: Un bazaar

Nel modello a Baazar, ognuno fa i propri interessi e sviluppa ciò che gli interessa sviluppare. Non c'è quindi un obiettivo comune, ma nel perseguire i propri interessi chi sviluppa nel modello a Baazar contribuisce a tenere viva la codebase. Questo processo non segue un modello prefissato, e quindi

produce una sorta di organismo in continua evoluzione, il quale scopo quindi non diventa obsoleto. Ma cosa permette a progetti come questi, che non adottano i modelli di Brooks, di non fallire?

- Le persone non lavorano perchè costrette a farlo per un'azienda. Chi contribuisce lo fa per interessi personali, ed è quindi interessato e motivato ("personal itch")
- Gli utenti sono considerati co-sviluppatori, ciò aiuta a individuare e risolvere bug più velocemente
- Rilasciare presto e frequentemente, in modo tale da avere sempre feedback

> **Legge di Linux**
> Data una base di beta-tester e co-sviluppatori abbastanza amplia, quasi ogni problema può essere scoperto e risolto velocemente da qualcuno.

Per Brooks un numero elevato di utenti porta inevitabilmente ad avere più bug, in quanto ogni utente può vedere problemi diversi. Raymond invece considera gli utenti come collaboratori, che possono aiutare lo sviluppatore. Anche questo modello, in teoria fantastico, in pratica è abbastanza idealistico.

1.3 Modello a Kibbutz

Per poter supportare delle applicazioni, un sistema deve fornire dei servizi adeguati, come kernel, driver, librerie di sistema ecc.
Linux è solamente un kernel, non ci si possono far girare applicazioni. Nasce quindi li concetto di **distribuzione**, ovvero un sistema completo immediatamente utilizzabile. Un programma viene quindi distribuito sotto forma di pacchetto, che è progettato per lavorare con una distribuzione (esempio pacchetto .deb).

> **Kibbutz**
> Fattoria, villaggio, impresa collettiva nata in Israele, con scopo di popolare il nuovo stato

Figure 2: Un kibbutz

Non parliamo quindi più di un bazaar, dove ognuno fa quello che vuole. Ma di un'organizzazione strutturata e organizzata, con uno scopo comune. Sono presenti delle **policy** prestabilite, che hanno dei corrispettivi tool che assicurano il rispetto di tali policy. Queste policy hanno l'effetto di abbassare drasticamente l'effort comunicativo fra i vari contributori al progetto. Praticamente molti progetti open-source sono organizzati a kibbutz, perchè il modello a Baazar non è una strada viabile.
Questa è la filosofia adottata da Debian.

1.3.1 Esempio di Debian

Fin dall'inizio, Debian è openSource (ad oggi circa 1900 sviluppatori, che devono superare degli esami). Supporta più di 10 architetture e tre kernel diversi (una distribuzione non è per forza linux).
In Debian, ogni pacchetto contiene alcune informazioni come:

- Nome
- Architetture supportate
- **Dipedenze**: tutti i pacchetti che servono al corretto funzionamento del mio pacchetto. Il solo codice sorgente del mio pacchetto è inutile senza queste informazioni

Nel pacchetto sono contenuti anche i propri file di configurazione. Questi file sono specificati nei pacchetti, quindi un upgrade non va a intaccare la configurazione. In caso ci siano problemi, chiedo all'utente di fare il merge.

10 Ottobre

2 Metodologie Agili

Negli anni 90 si pensa che il modo per aumentare la qualità di processi debba dipendere da una gran enfasi sulla produzione di documentazione, UML ecc. Secondo gli agilisti, tutto il focus che viene posto sui processi non risulta effettivamente in software di qualità.
Nascono quindi dei nuovi modelli organizzativi che reagiscono a queste cose pallose, come:

- eXtreme Programming
- Scrum

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 un 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.

15 Ottobre 2019

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.

2.1 Principi agile

Il manifesto agile espone 12 principi. Quelli più generali sono:

- Rilasciare software di valore, fin da subito e in maniera continua: la prima parte è un po' ovvia, La parte interessante è **fin da subito** e **in maniera continua**
- Consegnare **frequentemente** software funzionante

- Il **software funzionante** è la principale misura di progresso: attenzione che funzionante non vuol dire che soddisfa per forza le esigenze dell'utente
- **Cambiamenti nei requisiti** anche a stadi avanzati
- Committenti e sviluppatori devono lavorare insieme **quotidianamente**: è un po' più uno **scazzo** per il committente, e attribuisce meno responsabilità allo sviluppatore, evitando conflitti
- Conversazione faccia a faccia: ci si capisce meglio parlandosi e vedendosi

Sono presenti inoltre dei principi riguardanti il gruppo di lavoro:

- Individui motivati e ben supportati (abbastanza ovvio)
- Sviluppo sostenibile, ovvero essere grado di **mantenere indefinitamente un ritmo costante**: non faccio la tirata dell'ultimo secondo, perchè poi avrò delle ripercussioni
- Eccellenza tecnica: contrapposta alla qualità astratta nei modelli di Brooks
- Team che si auto-organizzano: sia per mantenere la motivazione, sia per riuscire a seguire dei ritmi sostenibili per il team
- A intervalli regolari il team riflette su come diventare più efficacie: è la stessa cosa di un controllo sui processi, ma implementato in modo più libero

L'ultimo principio invece è abbastanza poetico:

- La semplicità - l'arte di massimizzare la quantità di lavoro non svolto - è essenziale

2.2 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.

- 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.

2.2.1 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**.

17 Ottobre 2019

2.3 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.

2.3.1 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.

2.3.2 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).

2.3.3 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.

2.3.3.1 Pair Programming

`Programmazione in coppia, lit.`
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.

2.3.3.2 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**.

22 Ottobre 2019

2.3.3.3 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.

2.3.3.4 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.

2.3.3.5 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**. È la tipica board divisa in colonne (ex. Da Fare, In Lavorazione, Fatto). Si possono prevedere anche limiti di numerosità per colonna.
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**.

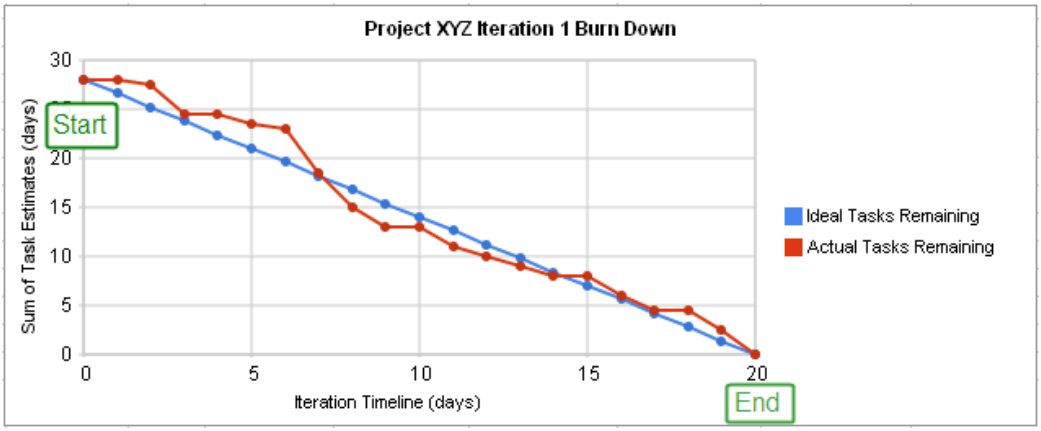

Figure 3: Burndown Chart

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.

24 Ottobre 2019

3 Continuos Integration

Nei primi anni dello sviluppo software si volevano seguire i modelli **Bottom Up** e **Top Down**. In realtà, sopratutto con l'object orientation, faccio tutto un po' alla cazzo, e non ho un chiaro modello di sviluppo.

> **Continuos Integration**
> Allineamento frequente (molte volte al giorno) dagli ambienti di lavoro degli sviluppatori verso l'ambiente condiviso (mainline)

Questo modello porta alcune conseguenze:

- Non esiste una fase di integrazione definita, ma si integra via via per piccole parti. Questo però comporta che ogni volta che sviluppo una feature, essa debba essere integrabile.

3.1 Capisaldi

Martin Fowler nel 2006 formula alcuni principi:

- **Mantenere una singola repo**: ci deve essere tutto, dal codice alla documentazione. Devo avere anche una chiara visione di tutte le vecchie versioni del codice.
- **Automatizzare la build**: già con make negli anni 70 era un problema. Ora con maven, gradle ecc siamo al top. In questo modo risparmio tempo e ho un processo più facile e affidabile.
- **Build auto-testante**: Non scrivo parti di codice senza un minimo di test, quantomeno per non regredire
- **Chiunque committa ogni giorno**: ad ogni commit di altre persone, integro i cambiamenti 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
- **Sistemo le build rotte subito**: se non riesco a sistemarla subito, faccio un revert e sistemo sulla mia macchina.
- **Keep the build fast**: una build veloce permette di testare ad ogni commit
- **Eseguire i test su un clone dell'ambiente di deployment**

- **Rendere facile ottenere l'ultimo eseguibile**: il vecchio approccio era fornire solo le parti non generabili, quindi solo il sorgente. Potrebbe essere che con ambienti di sviluppo diversi il codice non compili. Quindi, bisogna fornire anche gli eseguibili.
- **Tutti devono poter vedere cosa succede**: ovvero a che punto sono arrivato con lo sviluppo
- **Automatic Deployment**

3.2 Configuration Managment

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

> **Configuration Managment**
> 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.

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.

29 Ottobre 2019

È meglio configurare l'insieme dei file e il loro stato, in modo tale da avere sempre una configurazione funzionale.
Quando lavoro sui file, calcolo le differenze in base alle linee di testo.
Dagli anni 2000, con l'avvento dell'open source e di internet, si sono resi necessari nuovi metodi di collaborazione **distribuiti** e **flessibili**, come git e mercurial. Questo è necessario a permettere una collaborazione da parte di molte persone e anche saltuaria.

3.2.1 Artifact

Ogni SCM deve affrontare alcune scelte:

- **Archiviazione degli artifact**: è molto costoso e occupa molto spazio, ma permette di avere esattamente il prodotto compilato. In alternativa, si può salvare meticolosamente la configurazione di ambiente per avere un processo di compilazione **replicabile**

3.2.2 Sincronizzazione

Il meccanismo base negli SCM è quello di **check-in** (commit) e **check-out**, che vanno a versionare un set di cambiamenti su uno o più file. Non vado quindi a versionare immediatamente ogni linea di codice. 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.

3.2.3 Lavoro concorrente

Si può gestire in due modi:

- **Lock:** ogni file ha un lock, e può essere modificato da una persona alla volta. Non il migliore approccio.
- **Merge:** più persone possono modificare lo stesso file contemporaneamente, la prima persona che effettua il merge lo esegue, le altre risolvono i conflitti. È possibile lavorare anche senza connessione, e poi posso sincronizzare in seguito.

È chiaro che il **merge** è la tecnica superiore, anche se comunque a volte può essere difficoltoso. Posso avere varie situazioni:

- Lavoro parallelo su file diversi: non ho molti problemi, a meno che non faccio cazzate cambiando l'interfaccia.
- Lavoro parallelo su hunk diversi: nessun problema, non creo conflitti.
- Lavoro parallelo sullo stesso hunk: confitto :(, lo devo risolvere (ovviamente in locale) prima di pushare. Per capire chi abbia la versione modificata del file, vado a cercare un antenato comune fra la codebase A e la codebase B. Se A non presenta differenze e B si, allora vuol dire che è B che ha modificato il file.

> **Hunk**
> Insieme di righe adiacenti in file che hanno subito modifiche

3.2.4 Implementazione

I vecchi sistemi di versioning salvavano ogni volta i **delta** dalla versione predente o successiva. Questo risparmia spazio, ma andare ad accedere a varie versioni è molto costoso. Git, invece, ogni volta che viene modificato un file lo salva nuovamente (compresso magari). Se invece un file non viene modificato, il file non viene aggiornato, basta aggiungere un nuovo puntatore. Per capire se due file sono uguali, posso tranquillamente confrontare il loro hash, che mantengo sempre salvato come nome del file. Anche due file diversi ma in locazioni differenti sono salvati come puntatori allo stesso hash.

4 Git

Ci sono vari comandi base, non riportati qua, dei quali non verrà spiegato il funzionamento. Il file fondamentale di `.git` è `HEAD` che mi dice qual'è la versione sulla quale sto lavorando.
I file memorizzati sono salvati con il loro hash dento `.git/objects`. Per velocizzare le perfomance, si usa il bucketing, ovvero si usano le prime due lettere del nome per raggruppare i file in cartelle. Ovviamente questi file sono compressi. I file più vecchi sono invece spostati nella cartella pack, dove non rompono le palle. Posso stampare e accedere a questi file con `git cat-file`.

Quando faccio add e poi modifico un file, la prima versione del file può rimanere memorizzata finchè git non decide di fare garbage collection. 31 Ottobre
Ogni file viene compresso con:

- **Header** comprendente tipo del file e dimensione
- **Contenuto** del file

Git in realtà traccia i file in objects anche senza fare l'add. Se non vengono aggiunti, restano come dangling blobs, e vengono anche loro eliminati alla successiva garbage collection. I file che invece sono "indicizzati" sono visualizzabili da `git ls-files -s`.
Attenzione che le directory vuote non vengono tracciate! È meglio pensare al nome dei file come al loro intero percorso, ex `dir1/dir2/file.txt`, e sono questi che vengono committati.
Ogni commit ha:

- Uno o più genitori (zero nel caso sono la radice)
- Un `tree` con i file che ha linkato

- Dei metadati, come committer ecc.

Quando committo, non vado a svuotare lo stage. Lo stage contiene tutto ciò che andrà nel prossimo commit (compresi i file già presenti).

5 Novembre

HEAD è il puntatore al punto di partenza alla working directory. Attraverso il comando `git tag`, posso taggare il branch corrente con un nome a mia scelta, in modo tale da poterlo trovare più velocemente. Posso usare il comando `git checkout nomeTag` per entrare all'interno del branch. Posso riferirmi in vari modi a dei commit, come `tag`, `hash`, `HEAD` o `branch`. Posso anche riferirmi al padre (`cappuccio n`) o ad un antenato (`~n`).
Posso **riscrivere un commit** con `git commit --amend`, dimenticandomi così di mio padre.
`git reset [ref][--][files]` toglie le modifiche (fatte con `git add`) dallo stage, ma non le toglie dalla working directory. Se aggiungo `--files`, non vado a modificare la working dir, senza files sposto anche il branch puntato. Se uso `--hard`, tolgo i file anche dalla working dir. Con `--soft` non butto via ne la working dir, ne lo stage. Quindi, se puoi committo, sto praticamente facendo uno squash.

`git revert` prende un commit precedente a quello attuale, e 7 Novembre
crea un nuovo commit che riporta al codice allo stato del suddetto

commit. `git bisect` mi permette di effettuare la bisezione per cercare la comparsa di una modifica all'interno del codice