# Cattedrale, Bazaar e Kibutz

**Lettura obbligatoria:** The mythical man-month (capitoli 1 – 12)

**Non linearità del software:** Se un ponte regge 100 tonnellate, ne regge anche 50, i test sono lineari. Nel software il test con un input non dimostra che il software funzioni anche con un input diverso. Va scritto software “perfetto”.

## Problemi identificati da Brook

**Il software invecchia in fretta:** se passa troppo tempo, un software in sviluppo potrebbe risultare inutile o superate.

Le tecniche di stima sono ancora terribili al giorno d’oggi, anche se sono migliorate.

Si confonde “effort” con “progress”, si tende a monitorare il lavoro in base alle ore. In realtà le ore non contano. L’effort è molto facile da misurare, il progress dipende.

**Man-month:** unità di misura, quanti uomini-mese ci vogliono per realizzare un prodotto. Uomini e mesi sembrano intercambiabili ma in realtà si ritorna alla confusione tra effort e progress.

Esempio: per fare un bambino ci vogliono 9 man-month, se ho due donne ci vogliono sempre 9 man-month.

**Legge di Brooks:** aggiungere personale ad un progetto in ritardo, porterà solo il progetto a ritardare di più.

**Secondo sistema:** quando si scrive una cosa per la seconda volta si pensa di poter evitare gli errori della prima volte e aggiungere anche nuove funzionalità con il tempo risparmiato. FALSO

## La cattedrale / la sala operatoria

Distinzione chiara tra il lavoro architetturale da quello implementativo. La parte architetturale è accentrata, chi implementa non prende decisioni architetturali. C’è un single point of failure nel caso l’architettura presenta dei problemi.

Viene chiamata anche la sala operatoria. C’è un chirurgo e un co-pilota (programmatori bravissimi) e un equipe con ruoli precisi (persone normali).

## Bazaar

Eric Raymond prende come esempio il kernel di Linux e Fetchmail che coinvolgono tantissima gente e sono programmi di successo, ma usano un sistema diverso dalla cattedrale pensata da Brook. Quindi dice che non è vero che l’unico modo di sviluppare software è a cattedrale.

Un bazaar è un mercato in cui chiunque può mettere una bancarella e vendere quello che gli pare. La metafora è che ci si auto- organizza e non c’è nulla di pianificato. Il codice sorgente è condiviso, ogni persona è utente e programmatore. Si perseguono gli obbiettivi dei singoli, e così facendo indirettamente si costruisce l’intero software.

* Una persona contribuisce perché vuole, non perché lavora in azienda
* Gli altri utenti, che hanno altri obbiettivi, sono comunque in grado di interagire al livello dello sviluppo
* Rilascia appena puoi. Rilascia spesso. Si riduce il rischio di aver messo molto effort senza progress
* Linus’ law: “*se mi date abbastanza palle degli occhi, ogni bug viene a galla*”. All’aumentare degli utenti di un sistema, aumenteranno i bug. Se sono utenti consapevoli, loro saranno in grado di spiegarvi perché una funzionalità non va.

Non è semplice definire un bug, senza specifiche ognuno potrebbe avere una sua visione su quali siano bug.

## Il Kibbutz

Diverse organizzazioni si occupano di ambiti diversi della stessa applicazione. Ad esempio kernel Linux + applicazioni.

* Distribuzioni Linux
* Pacchetto → ha le informazioni necessarie per integrarsi in una distribuzione (dipendenze)

I kibbutz sono aziende agricole create con la nascita di Israele il cui scopo è permettere lo sviluppo della terra promessa.

Lavorare in un Kibbutz significa fare uno sforzo collettivo per un ideale condiviso e superiore.

### Debian

Debian utilizza il modello Kibbutz, è diverso dal Bazaar perché c’è un’ organizzazione con delle policy.

Le policy sono decise da un comitato tecnico composto da una decina di persone

Ad ogni policy corrisponde un tool che incarna la policy, quindi non è necessario conoscere e capire al 100% le policy.

### Dpkg e conffiles

Quando si aggiorna un programma da una versione precedente ad una successiva spesso le personalizzazioni fatte dall’utente vengono perse. Questo succede perché l’aggiornamento di un programma è in realtà l’installazione di un nuovo programma.

Debian cerca di risolvere questo problema facendo in modo che le parti del programma personalizzabili siano indicate.

Nel 99% è un file di testo nella cartella /etc.

**Nomenclatura:**

* V → pacchetto vecchio
* V’ → pacchetto nuovo
* Cd → file di configurazione di V
* Cu → file originale del sistema cambiato dall’utente
* C’d → file di configurazione di V’

Quando un pacchetto viene installato sopra uno vecchio si verificano **4 condizioni diverse**:

* Cd = C’d = Cu → niente da configurare, l’utente non ha modificato niente
* Cd = C’d and Cd != Cu → l’utente ha modificato il file di configurazione, posso tenere la configurazione vecchia perché è compatibile con la nuova versione del programma V
* Cd != C’d and Cd = Cu → l’utente non ha modificato il file di configurazione, gli metto il nuovo file di configurazione
* Cd != C’d and Cd != Cu → il file di configurazione della nuova versione di V è cambiato, e l’utente aveva modificato la configurazione, in qualche modo si cerca di adattare il nuovo file di configurazione

# Sviluppo agile

Negli anni 90 nascono una serie di movimenti che propongono di sviluppare software in modo diverso dalla visione classica degli ingegneri del software.

Nel 2001 queste idee sono raccolte nell’agile manifesto. Questo manifesto è astratto e non appartiene a nessun progetto.

### Valori dell’agile manifesto

* Individui e le interazione più che i processi e gli strumenti
* Il software funzionante più che la documentazione esaustiva
* La collaborazione col cliente più che la negoziazione dei contratti
* Rispondere al cambiamento più che seguire un piano

I progetti di solito si sviluppano seguendo un canone che è utilizzato per valutare la correttezza e la qualità di un prodotto. È difficile trovare un’alternativa allo sviluppo secondo un canone. Gli agilisti infatti non propongono un canone ma una serie di principi da seguire.

### Principi dell’agile manifesto

1. Rilasciare software di valore, fin da subito e in maniera continua
2. Cambiamenti nei requisiti anche a stadi avanzati
3. Rilasciare frequentemente software funzionante
4. Il software funzionante è la principale misura di progresso
5. Committenti e sviluppatori devono lavorare insieme quotidianamente
6. Individui motivati e ben supportati
7. Conversazione faccia a faccia
8. Sviluppo sostenibile: mantenere un ritmo costante  no straordinari, indicano pianificazione sbagliata
9. Eccellenza tecnica
10. Semplicità → fare sempre il minimo
11. Auto organizzazione
12. A intervalli regolari il team riflette su come diventare più efficace

### I veri punti chiave

* **Team piccoli e auto-organizzati,** senza manager tradizionali, ma facilitatori. Il manager è una persona che gestisce, il facilitatore tenta di mettere d’accordo i membri del gruppo
* **Rifiuto di azioni e decisioni big upfront →** energie spese nel fare più che nel progettare, se in futuro ci sarà un cambiamento da fare, si spenderanno energie per quel cambiamento. È comunque meglio che immaginarsi tutti i possibili cambiamenti prima di iniziare a sviluppare
* **Misura e controllo del processo di sviluppo**, con pianificazioni con orizzonti temporali ridotti → il processo è sempre importante, si rifiuta una pianificazione totale preferendo una pianificazione giornaliera
* … morta la registrazione

# Scrum

Scrum è descritto in un piccolo manuale di riferimento. Scrum in italiano si traduce come “mischia del rugby”. La metafora è che tutti insieme si superano le avversità.

## I ruoli

I gruppi sono coesi e piccoli, formati da 7 + - 2 persone.

Ci sono 2 ruoli definiti:

* **product owner:** interfaccia col cliente / committente, fissa le priorità in base a opportunità e rischi di buisnes e gestisce il backlog. Nella metodologia agile si cercava di portare il committente nel gruppo di sviluppo, questo però trova l’opposizione del committente che non è interessato a partecipare al lavoro. In Scrum invece si sceglie una persona che svolge il ruolo del committente, ma che è sempre interna al team, e partecipa a tutte le attività del team
* **scrum master:** è il facilitatore del gruppo, non fa veramente parte del gruppo ma controlla che gli altri membri agiscano secondo le metodologie Scrum e che non ci siano impedimenti
* **Gli altri:** stimano la complessità del lavoro, identificano i rischi, dimostrano il progresso del prodotto scrivendo codice.

## Il backlog

Il backlog sono le cose che mancano da fare. Il product owner gestisce il backlog. In un sistema a cascata non c’è motivo di gestire il backlog, perché man mano che faccio qualcosa lo cancello dalla lista. In Scrum non si pianifica all’inizio, ma si risponde al cambiamento ripianificando a piccoli passi.

## Pianificazione

Scrum organizza i requisiti in user stories. Ogni volta che si fa una stima ci si riferisce ad una o più storie. Un insieme di storie si chiama epopea. Il lavoro si divide in periodi di 1 – 3 settimane chiamati sprint. Alla fine dello sprint si rilascia.

Finestra chiusa → è vietato aggiungere nuove funzionalità ad uno sprint in corso. Tutte le eventuali tutte le eventuali negoziazioni sono rimandate allo sprint successivo. In uno sprint al massimo si consegna meno di quello pianificato, mai di più. Se si vuole consegnare di più si inizia un nuovo sprint. La pianificazione quindi diventa “che cosa facciamo nel nuovo sprint?”

Per capire cosa mettere nello sprint servono due cose:

* velocity → quanto si è veloci a fare le cose
* stima del tempo necessario per un requirement → viene trasformato in user stories, per ogni storia si calcola quanta fatica è richiesta

### Stima di una stories

Si scegli la user story più chiara per tutti, sia per il tempo richiesto, sia per effettivamente quello che va fatto. É scelta come user story 1 ed è il riferimento per quantificare tutte le altre stories.

Planning poker → ogni persona dice quant’è lunga secondo lui una storia. Non lo fa dicendolo esplicitamente ma prendendo una carta con un numero segreto. In questo modo una stima non influenza le altre. Si girano tutte le carte, chi ha la più alta/bassa deve argomentare il perché. L’obbiettivo è arrivare ad una stima condivisa da tutto il gruppo.

Altro metodo più semplice è quello delle magliette (S, M, L)

### Riunioni di planning

**regole riunioni:**

* hanno un timer
* si distinguono maiali da polli. I maiali sono direttamente coinvolti e prendono le decisioni, i polli sono solo interessati e danno pareri solo se richiesto dai maiali. Una persona è almeno un pollo perché tutti sono interessati

**stand up meeting** → riunione di 15 minuti fatta in piedi tutte le mattine, si discute cosa si è fatto il giorno prima e cosa sarà fatto oggi, si discute se ci sono impedimenti.

**Planning** → lo si fa ogni 1/5 giorni, più o meno a metà sprint. È l’aggiornamento del backlog.

**Retrospettiva** → alla fine dello sprint, si valuta quello che è stato fatto. È fatta solo con il gruppo

**review** → visto che si rilascia la review coinvolge anche il cliente

## Tecniche di programmazione

### Pair programming

Si programma in coppia, ci sono due ruoli:

Il pilota usa la tastiera, scrive solo quello che gli dice il copilota. L’idea è che il copilota è costretto ad esplicitare quello che vuole fare. Il pilota fa da filtro per quello che dice il copilota. I due ruoli non sono fissi, si cambiano ogni n minuti. Studi dimostrano che la produttività è leggermente minore, ma la qualità del codice è maggiore. C’è anche un livellamento delle skill, quelli più bravi permettono a quelli meno bravi di migliorare.

### Codice condiviso

Solitamente la modularizzazione del software è usata come divisione del lavoro ma anche come rigida distinzione delle responsabilità. Un modulo è solo di una persona, ci mette le mani solo lui, se c’è un bug è colpa sua.

Con il codice condiviso invece tutti possono mettere le mani dappertutto. È prevalente l’interesse del gruppo a far funzionare le cose rispetto a distinguere le responsabilità sul codice. Anche nei gruppi agili comunque normalmente una persona mette le mani solo sulla sua parte, però se il sistema non funziona è consentito andare a risolvere i problema degli altri.

La condivisione del codice è resa tollerabile dalla continuos integration, cioè andare continuamente ad andare a integrare i moduli per controllare se funzionano quando messi insieme.

### Refactoring

Refactoring vuol dire riscrivere un pezzo di codice senza cambiare la sua funzionalità. Non esiste però un modo automatico per sapere se il codice nuovo è davvero equivalente a quello vecchio.

Aggiungendo un criterio di funzionalità come quello del TDD, si può affermare che due codici sono equivalenti se i due codici riescono a superare gli stessi test.

Queste operazioni di refactoring si fanno con l’aiuto di un tool e non a mano. Ad esempio il cambio di nome di una variabile non è un semplice search and replace, ma è una ricerca semantica di quella specifica variabile.

I cataloghi di refactoring servono per future implementazioni nei tool e per la maturazione dei programmatori che andando a leggere il razionale dei refactoring posso imparare modi più eleganti per sistemare il codice

Code smell → stessa riga di codice più volte, e altro…

### TDD

Test driven developement, si traduce con sviluppo guidato dal test. Non è solo una tecnica di verifica, ma una tecnica di specifica, con la quale si definisce che cosa si vuole ottenere. I test li scrive la stessa persona che poi cerca di passare questi test

* Rosso: test non superato, l’obbiettivo è farlo diventarlo verde. Solitamente un test deve essere passato in 10 minuti, se ci vuole più tempo è troppo complicato, spezzo il test in sotto-test.
* Verde: test superato. Si fa refactoring del codice, il test deve essere ancora verde dopo il refactoring.

**Problemi:**

* Il TDD è che è estremamente dipendente dal tool, per esempio se faccio test su database e ogni volta ci vogliono 15 minuti per verificare se un test è rosso o verde allora il TDD non è fattibile.
* Alcuni test possono avere side effect irreversibili (esempio mandare una mail di test).
* Spesso si vogliono fare test su pezzi di codice che non si stanno sviluppando.

Questi problemi si risolvono con i mock object, classi che imitano il comportamento di altre classi reali. I mock non eseguono il codice vero, ma gli viene insegnato a rispondere in un certo modo alle chiamate dei metodi.

### Velocity tracking

La velocità è una caratteristica del gruppo di lavoro, non la si conosce a priori ma si impara a conoscere a seguito di diversi progetti.

Il tracking è la maniera con la quale si cerca di misurare il progresso. È una velocità perché è il progresso rispetto all’effort.

La velocità è inizialmente stimata riferendosi a 1/3 del tempo a disposizione; con 6 programmatori e uno sprint di 2 settimane:

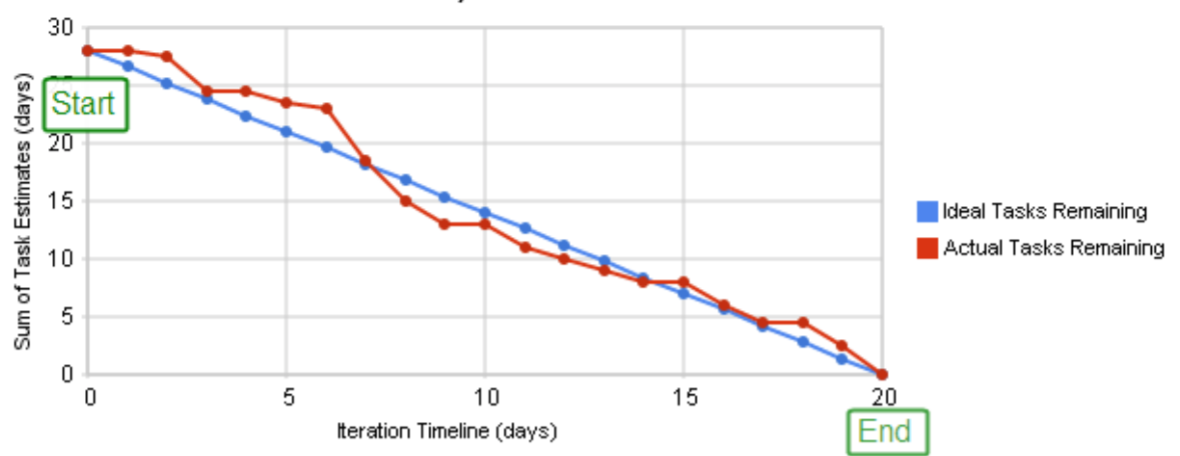
Per le storie si usa una taskboard, una lavagna divisa in 4 colonne:

* story: storie da fare in uno sprint
* not started: storie non iniziate
* in progress: storie in corso
* done: storie finite

Si possono mettere dei limiti su una colonna, ad esempio un massimo di 3 storie in progress. Il numero di storie non è detto che sia fisso, se una storia è troppo complessa potrebbe venir divisa in 2 storie distinte. É sempre vietato introdurre nuove storie.

burndown chart: quanto ci metto a bruciare tutti i miei post-it

* 30 post-it iniziali
* sprint 20 giorni
* alla fine dello sprint idealmente devo avere 0 post-it
* Ogni giorno guardo quanti post-it ho finito, e controllo se sono oltre il pianificato o sotto il pianificato.



# Integrazione continua

Negli ultimi 10 anni gli strumenti di versioning sono cambiati radicalmente. In passato si pensava che dividere il programma in tanti rami fosse un problema. Oggi invece ogni feature ha un ramo diverso, e ogni programmatore lavora su più rami in parallelo.

## Comandi Git

Lo stesso comando git ha molte opzioni che modificano radicalmente il suo comportamento per evitare di avere molti comandi con sintassi diversa. É possibile espandere git in maniera molto semplice aggiungendo comandi personalizzati.

### git init

Inizializza un repository vuota creando una cartella .git che contiene:

* HEAD
* config
* description

### git status

Rappresenta la situazione attuale. I file untracked sono file presenti nel working space ma non aggiunti alla repository. Non cambia nulla nella directory di git.

### git add

I file in add verranno aggiunti alla repository alla prossima commit. La add è utile perché si può usare la copia creata per tornare ad una versione precedente senza però fare commit. Oppure per scegliere un sottoinsieme di file di cui fare la commit. Nella cartella .git viene creato un nuovo file che è l’hash del file aggiunto.

L’hash serve per:

* verificare l’uguaglianza tra file
* risparmiare spazio su file con lo stesso contenuto (stessa versione di più file)
* essere sicuri che il file sia stato salvato nel modo corretto

Nell’hash non c’è il nome del file ma solo il contenuto. Infatti se faccio add di un file con un nome diverso ma contenuto uguale ad un altro file già presente nella repo git, allora non verrà fatto un nuovo hash. Git tiene comunque conto che un nuovo file è stato creato.

Funzionamento git add dettagliato:

* calcola dimensione file
* crea una stringa composta da type + spazio + dimensione + terminatore stringa

type per tutti i file è uguale a BLOB

* concatena il risultato con il contenuto vero e proprio
* calcola lo SHA1 della “stringa” complessiva. Le prime due cifre dell’hash sono usate per accedere al filesystem in maniera efficiente

### git cat-file

Dato un hash ritorna informazioni sul file. Se l’hash è univoco basta specificare i primi quattro caratteri.

* -t → tipo
* -s → dimensione
* -p → stampa il contenuto (non sempre vero)

### File index

é un file fondamentale, essendo molto usato deve essere efficiente, è uno dei pochi file che non è testuale ma binario. Sull’index c‘è tutto quello che poi andrà in commit.

### git commit

Il commit necessita di un commento. Crea un nuovo file di tipo commit che contiene autore, data e commento. Nascosto c’è un puntatore ad un altro hash.