

# Appunti di Sviluppo Software in Gruppi di Lavoro Complessi

Caccaro Sebastiano  
A.A.2019/2020

# Contents

<b>1</b>	<b>Modelli Organizzativi</b>	<b>2</b>
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
<b>2</b>	<b>Metodologie Agili</b>	<b>6</b>
2.1	Principi agile . . . . .	6
2.2	Principi agile nella pratica . . . . .	7
2.2.1	Enfasi sul testing . . . . .	8

# 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 accentrare 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.

### Bazaar

Mercato autogestito, dove chiunque può mettere una bancarella dove vuole, quando vuole



Figure 1: Un bazaar

Nel modello a Bazaar, 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 Bazaar 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 ampia, 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 il 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 Bazaar 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
- **Dipendenze:** 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 pallone, 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 efficace: 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 preoccupa 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 finché non è completamente chiaro che ne ho bisogno. Cerco quindi di evitare l'**over engineering**
- 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 preoccupa 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**.