Appunti di Sviluppo Software in Gruppi di Lavoro Complessi

Caccaro Sebastiano
A.A.2019/2020

Contents

4 Ottobre 2019

1 Problemi dello sviluppo software

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.

1.1 Modelli organizzativi

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

1.1.1 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.

1.1.2 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

1.1.3 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.

1.1.4 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. Questa è la filosofia adottata da Debian.

1.1.5 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