Appunti di Sviluppo Software in Gruppi di Lavoro Complessi

Caccaro Sebastiano
A.A.2019/2020

Contents

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.

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**.
Attenzione a non confondere la condivisione del codice come ragione per rinunciare all'**Information Hiding**.