Processo e Sviluppo del Software

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Le immagini presenti in questi appunti sono tratte dalle slides del corso e tutti i diritti delle stesse sono da destinarsi ai docenti del corso stesso.

Capitolo 2

Metodi Agili

I *metodi agili* sono stati definiti per rispondere all'esigenza di dover affrontare lo sviluppo di software in continuo cambiamento. Durante lo sviluppo si hanno vari passaggi:

- comprensione dei prerequisiti
- scoperta di nuovi requisiti o cambiamento dei vecchi

Questa situazione rendeva difficile lo sviluppo secondo il vecchio metodo *waterfall* (portando al fallimento di diversi progetti).
I *metodi agili* ammettono che i requisiti cambino in modo "naturale" durante il processo di sviluppo software e per questo assumono un modello di processo *circolare*, con iterazioni della durata di un paio di settimane (figura 2.1). Potenzialmente dopo un'iterazione si può arrivare ad un prodotto che può essere messo in "produzione". Dopo ogni rilascio si raccolgono *feedback* per poter rivalutare i requisiti e migliorare il progetto.
Si hanno quindi aspetti comuni nei metodi agili e nel loro processo:

- enfasi sul team, sulla sua qualità e sulla sua selezione
- il team è *self organizing*, non essendoci un *manager* ma essendo il team a gestire lo sviluppo, dando importanza ai vari membri del team
- enfasi al pragmatismo, evitando di produrre documenti inutili che sarebbero difficili da mantenere, focalizzandosi su una documentazione efficacie
- enfasi sulla comunicazione diretta e non tramite documenti, tramite meeting e riunioni periodiche

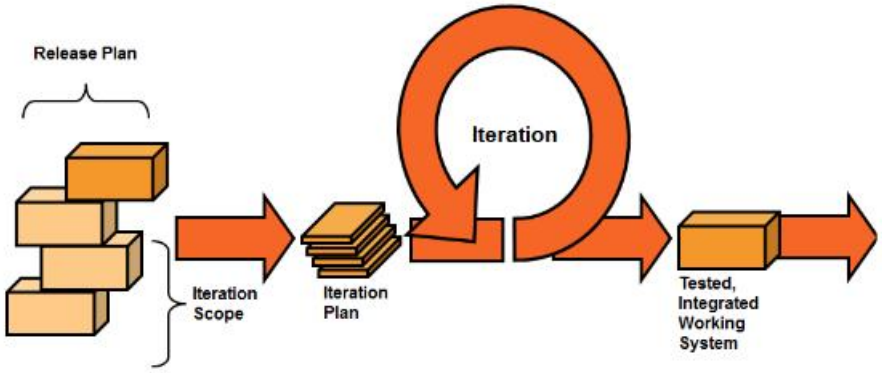

Figura 2.1: Rappresentazione grafica del modello agile. I blocchi a sinistra rappresentano i requisiti (da sviluppare secondo una certa priorità). Lo sviluppo si articola tramite varie iterazioni che porta ogni volta ad aggiungere una parte al prodotto finale (ogni iterazione produce, a partire da un certo requisito, un parte di prodotto di qualità già definitiva, con testing, documentazione etc...)

- enfasi sull'idea che nulla sia definitivo e che non bisogni seguire la perfezione fin da subito ma che, "pezzo per pezzo", ogni step porterà al raggiungimento di una perfezione finale (anche dal punto di vista del design)
- enfasi sul controllo costante della qualità del prodotto, anche tramite pratiche tecniche come quella del *continuous testing*, dove un insieme di test viene eseguita in modo automatico dopo ogni modifica, o anche pratiche di *analisi statica e dinamica* del codice al fine di trovare difetti nello stesso nonché di *refactoring*

I metodi agili sono molto "elastici" e permettono la facile definizione di nuovi metodi facilmente adattabili al singolo progetto.

2.1 Scrum

Tra i vari *metodi agili* uno dei più famosi è **scrum** (figura 2.2).
In questo caso la parte di sviluppo e iterazione prende il nome di *sprint* e ha una durata variabile tra una e quattro settimane, per avere un rilascio frequente e una veloce raccolta di feedback. I requisiti sono raccolti nel cosiddetto *product backlog*, con priorità basata sulla base delle indicazioni del committente. Ad ogni *sprint* si estrae dal *product backlog* lo *sprint backlog*,

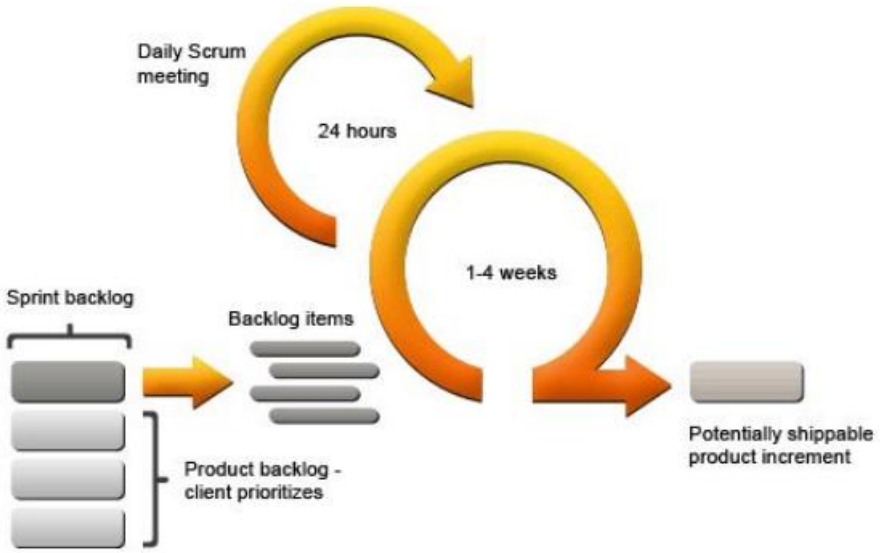

Figura 2.2: Rappresentazione grafica del processo scrum

ovvero il requisito (o i requisiti) da implementare nello *sprint*. Lo *sprint backlog* viene analizzato nel dettaglio producendo i vari *backlog items*, ovvero le singola funzionalità che verranno implementate nello *sprint*. Si ottiene quindi di volta in volta un pezzo di prodotto finale, testato e documentato. Durante le settimane di *sprint* si ha anche un meeting giornaliero utile per mantenere alti i livelli di comunicazione e visibilità dello sviluppo. Durante il meeting ogni dev risponde a tre domande:

1. Cosa è stato fatto dall'ultimo meeting?
2. Cosa farai fino al prossimo meeting?
3. Quali sono le difficoltà incontrate?

l'ultimo punto permette la cooperazione tra team members che sono consci di cosa ciascuno stia facendo.
Durante il processo *scrum* si hanno quindi tre ruoli:

1. il **product owner**, il committente che partecipa tramite feedback e definizione dei requisiti
2. il **team**, che sviluppa
3. lo **scrum master**, che controlla che il processo scrum sia svolto correttamente

Essi collaborano nelle varie fasi:

- product owner e team collaborano nella definzione dei backlog e nella loro selezione ad inizio *sprint*
- durante lo *sprint* lavora solo il team
- nello studio del risultato collaborano tutti coloro che hanno un interesse diretto nel progetto (team, product owner, stakeholders)

Lo scrum master interagisce in ogni fase, fase che viene comunque guidata tramite meeting:

- **sprint planning meeting**, ad inizio *sprint*
- **daily scrum meeting**, il meeting giornaliero
- **sprint review meeting**, in uscita dallo *sprint* per lo studio dei risultati
- **sprint retrospective meeting**, in uscita dallo *sprint* per lo studio tra i membri del team di eventuali migliorie al processo e allo sviluppo del prodotto (anche dal punto di vista delle tecniche e delle tecnologie)

2.2 Extreme Programming

Un altro tipo di metodo agile è l'*extreme programming* (figura 2.3), ormai poco usato. I requisiti prendono i nomi di *stories*, che vengono descritti come l'utente del sistema, detto *attore*, che cerca di fare qualcosa tramite una "narrazione". Vengono scelti quindi *stories* per la prossima iterazione, dove si hanno testing e revisione continua. Le release di ogni iterazione vengono catalogate per importanza (con anche la solita collezione di feedback). In *extreme programming* si hanno davvero molte pratiche:

- The Planning Game
- Short Releases
- Metaphor
- Simple Design
- Refactoring

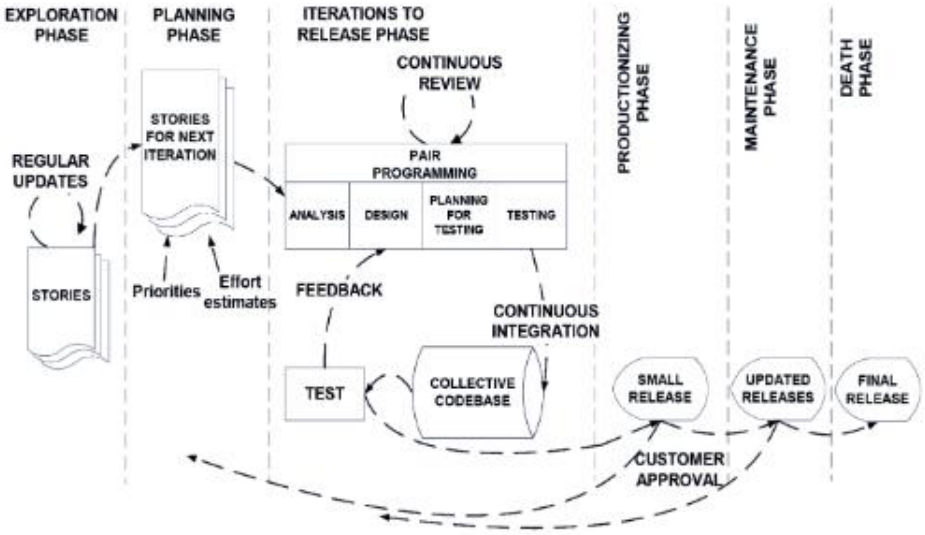

Figura 2.3: Rappresentazione grafica dell'extreme programming

- Test-First Design
- Pair Programming
- Collective Ownership (codice condiviso da tutti senza alcun owner, con tutti in grado di effettuare modifiche)
- Continuous Integration
- 40-Hour Week
- On-Site Customer
- Coding Standard (per avere il collective ownership avendo un solo standard di codifica comune a tutti)
- Open workspace

Capitolo 3

DevOps

Nel tempo si è passato dal *modello waterfall* (dove tutto era definito ad inizio progetto) al *modello agile*. Negli ultimi tempi si è sviluppato un altro metodo, chiamato **DevOps**, dove anche la parte di *operation* deve essere agile, ovvero il rilascio in produzione e il *deployment* devono essere agili quanto lo sviluppo. Amazon, Netflix, Facebook e molte altre società già adottano le pratiche *DevOps.*
Nel *DevOps* i team di sviluppo e operation sono indipendenti tra loro, diminuendo il costo di impegno necessario al team di sviluppo per la parte di deployment (che viene resa anche più sicura grazie alla diminuzione dell'intervento umano in favore di automazioni). Inoltre, avendo i due team dei tempi di lavoro diversi, si riesce a prevenire ritardi causati dalla non organicità delle operazioni.
DevOps promuove la collaborazione tra di due team al fine di ottenere una sorta di team unico che curi sia sviluppo che operation.
DevOps include quindi diversi processi che vengono automatizzati:

- Continuous Development
- Continuous Integration
- Continuous Testing
- Continuous Deployment (anche con tecnologie di virtualizzazione e con l'uso di *container* in *ambiente cloud*)
- Continuous Monitoring

Con DevOps il feedback arriva in primis dal software (i tool di *monitor*) inoltre il focus viene spostato sui processi automatici.
Nel DevOps si introducono nuovi ruoli:

- il **DevOps Evangelist**, simile allo *scrum master*, supervisiona l'intero processo di DevOps
- l'**automation expert**, dedicato a curare gli aspetti di automatismo
- un **Security Engineer**
- un **Software Developer**, nonché **Tester**
- un **Quality Assurance**, che verifica la qualità del prodotto rispetto ai requisiti
- un **Code Release Manager**, che si occupa sull'infrastruttura e sul deploy della release

Il DevOps (figura 3.1) si basa su sei principi base:

1. **Customer-Centric Action**, ovvero il committente è al centro dell'azione
2. **End-To-End Responsibility**, ovvero il team gestisce interamente il prodotto, avendone responsabilità totale
3. **Continuous Improvement**, ovvero cercare continuamente di migliorare senza sprechi il prodotto finale e i servizi
4. **Automate everything**, ovvero cercare di automatizzare l'intera infrastruttura di processo, dalle attività di testing e integrazione fino ad arrivare alla costruzione della release e del deployment
5. **Work as one team**, ovvero unificare tutti gli aspetti sotto un unico team o comunque con due team che collaborano fortemente come se fossero uno
6. **Monitor and test everything**, ovvero testare e monitorare costantemente il prodotto

Il quarto e il sesto punto sono i due punti tecnici principali.

Figura 3.1: Rappresentazione famosa del *lifecycle* di DevOps, con le due parti, con le relative parti fondamentali, di sviluppo e operation distinte dai colori ma unite in un singolo metodo unico

3.0.1 Build, Test e Release

Partiamo dalla parte "dev" di DevOps.
Bisogna pensare a questi step in ottica di automatismo vicina al DevOps.
Innanzitutto bisogna introdurre i sistemi di **version control**, in primis **Git**, un sistema di version control **distribuito**, dove ogni utente ha una copia della repository (con la storia dei cambiamenti), con la quale interagisce tramite *commit* e *update*. Esiste poi una repository lato server per permettere di condividere i vari cambiamenti tramite sincronizzazione.
Git permette anche lo sviluppo *multi-branch* per permettere di lavorare su più branch, pensando allo sviluppo separato dove ogni branch alla fine può essere unito (*merge*) con quello principale (solitamente chiamato *master*). Un altro branch standard è quello di sviluppo, detto *develop*. Un altro ancora è quello detto *hotfix*, per le modifiche più urgenti, che vive in uno stadio intermedio tra i due sopracitati, prima ancora del branch *develop*. Volendo ciascuna feature può essere creata in un branch dedicati. Si procede con le versioni "merge-ndo" quando necessario, "step by step" fino ad un branch *release* per poi andare dopo il testing su *master*. Sul branch *release* possono essere effettuati fix che poi verranno riportati anche in *develop*.
Lo sviluppo su multi-branch si collega alle operazioni di verifica che possono essere attivate automaticamente a seconda dell'evoluzione del codice su ogni branch. Avendo ogni branch una precisa semantica possiamo definire precise attività di verifica, corrispondenti a *pipelines* precise, solitamente innescate da un *push* di codice su un certo branch, in modo sia automatico che manuale. Le pipeline vengono attivate in fase di test di un componente, in fase di creazione di un sottosistema, di assembramento di un sistema intero o di

deployment in produzione. Si hanno quindi quattro fasi:

1. component phase
2. subsystem phase
3. system phase
4. production phase

Spesso le pipelines sono usate come *quality gates* per valutare se un push può essere accettato in un certo branch. Una pipeline può essere anche regolata temporalmente, in modo che avvenga solo ad un certo momento della giornata.

Component phase e Subsystem phase

Dove il fuoco è sulla più piccola unità testabile che viene aggiornata (una classe, un metodo etc...) che non può essere eseguita senza l'intero sistema. In tal caso si può fare:

- code review
- unit testing
- static code analysis

Un cambiamento può anche essere testato nell'ambito del sottosistema di cui fa parte, in tal caso si hanno anche check di prestazioni e sicurezza. Il servizio però potrebbe essere da testare in isolamento rispetto ad altri servizi, usando quindi dei *mocks* o degli *stubs*, ovvero creando degli alter ego dei servizi mancanti in modo che il servizio da testare possa funzionare.

System phase

In questo caso si testa l'intero sistema che viene "deployato" in ambiente di test. Si hanno:

- integration tests
- performance tests
- security tests

Tutti test che richiedono l'interezza del sistema e sono spesso molto dispendiosi e quindi bisogna regolare la frequenza di tali test in molti casi (sfruttando ad esempio la notte).

Production phase

Questa fase è legata alla necessità di creare gli artefatti che andranno direttamente "sul campo", ovvero il deployment in produzione. In tale fase potrebbe essere necessario creare container o macchine virtuali. In tal caso si hanno dei check molto veloci sugli artefatti finali (che non siano per esempio corretti), dando per assodato che la qualità del codice sia già stata testata. Si hanno quindi strategie anche di *deployment incrementale*, per cui esistono più versioni del software contemporaneamente con diversa accessibilità per gli utenti finale (accessibilità che viene man mano scalata). In tal caso si usano anche vari tool di monitor. Si hanno anche eventualmente tecniche di *zero downtime* (dove il software non è mai in uno stato *unstable*).

Fasi diverse corrispondono a branch diversi

3.0.2 Deploy, Operate e Monitor

Studiamo ora la parte "Ops" di DevOps.