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* (a cascata) 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*, si da importanza ai vari membri del team dato che non esiste un *manager* ma è il team stesso a gestire lo sviluppo
- enfasi al pragmatismo, focalizzandosi su una documentazione efficace evitando di produrre documenti inutili e difficili da mantenere
- enfasi sulla comunicazione diretta, sostituendo i documenti suddetti con meeting e riunioni periodiche

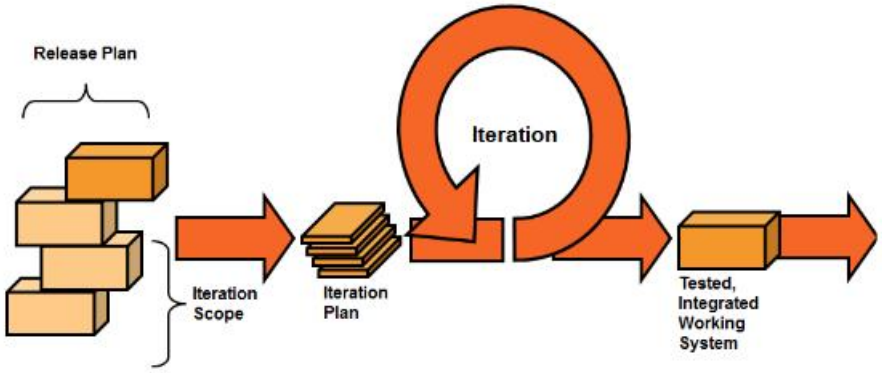

Figura 2.1: Rappresentazione grafica del modello agile. I blocchi a sinistra rappresentano i requisiti (da sviluppare secondo una certa priorità). Lo sviluppo si articola tramite varie iterazioni che porta ogni volta ad aggiungere una parte al prodotto finale (ogni iterazione produce, a partire da un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

- enfasi sull'idea che nulla sia definitivo: la perfezione non deve essere seguita fin da subito ma saranno gli step a portare al raggiungimento di una perfezione finale (anche dal punto di vista del design)
- enfasi sul controllo costante della qualità del prodotto, anche tramite
  - *continuous testing* grazie al quale un insieme di test viene eseguito in modo automatico dopo ogni modifica
  - *analisi statica e dinamica* del codice al fine di trovare difetti nello stesso
  - *refactoring*

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

2.1 Scrum

Uno dei più famosi, tra i vari *metodi agili*, è **scrum** (figura 2.2).
In questo caso la parte di sviluppo e iterazione prende il nome di *sprint* ed

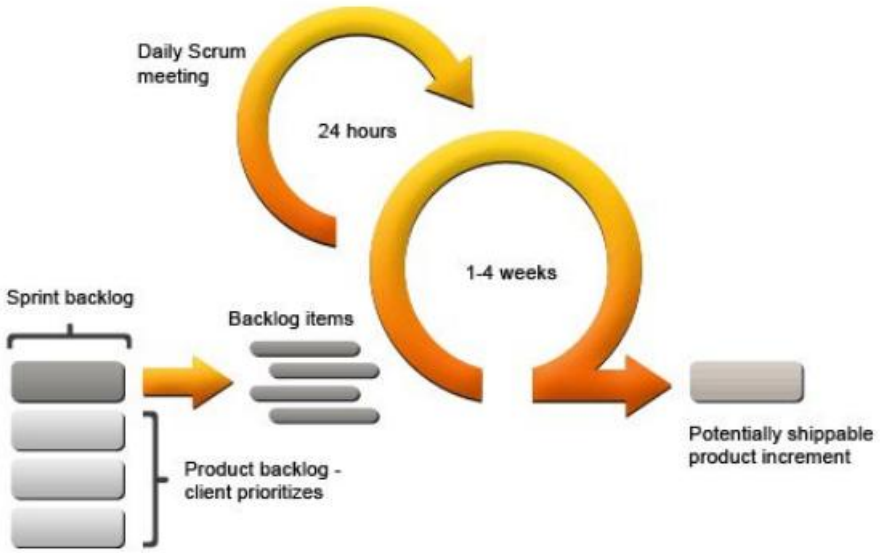

Figura 2.2: Rappresentazione grafica del processo scrum

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*, ovvero il requisito (o i requisiti) da implementare nello *sprint*. Lo *sprint backlog* viene analizzato nel dettaglio producendo i vari *backlog items*, ovvero le singole 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 effettua anche un meeting giornaliero utile per mantenere alti i livelli di comunicazione e visibilità dello sviluppo. Durante il meeting ogni sviluppatore 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*, 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 la correttezza di svolgimento del processo scrum

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 e 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*, delle narrazioni in cui l'attore (futuro utente del sistema) cerca di svolgere un compito. 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

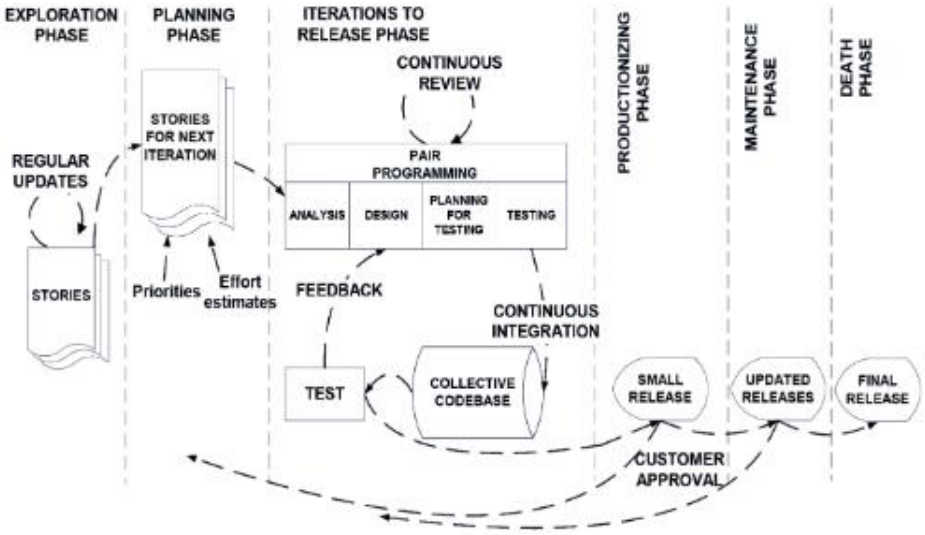

Figura 2.3: Rappresentazione grafica dell'extreme programming

- Simple Design
- Refactoring
- 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: 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

3.1 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 dedicato. 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. 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 finali (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

3.2 Deploy, Operate e Monitor

Studiamo ora la parte "Ops" di DevOps.
Si studia l'evoluzione automatica del software da una versione all'altra in produzione. Avanzare di versione in modo *naive* e istantaneo è troppo rischioso (qualora la nuova versione fosse corrotta non si avrebbe controllo sull'update) e quindi spesso non attuabile (spesso anche per ragioni tecniche). Si ha quindi un insieme di tecniche che si basano in primis sull'*evoluzione incrementale*. Tali tecniche si distinguono in base alla dimensione su cui sono incrementati:

- **Incremental wrt users:** *Dark launching, Canary releases (and User Experimentation)*, ovvero legata agli utenti esposti alla nuova release
- **Incremental wrt requests:** *Gradual upgrades/rollout*, ovvero legata alle richieste per la nuova release
- **Incremental wrt components/replicas:** *Rolling upgrade*, incentrata sulle componenti che vengono aggiornate
- **Non-incremental with backups:** *Green/blue deployment, Rainbow deployment*, non incrementali ma che offrono comunque un backup di sicurezza

Tali schemi possono essere usati in un contesto DevOps.
Per studiare la prima tipologia (*Incremental wrt users*) abbiamo:

- **Dark launching** che arriva dal mondo di Facebook dal 2011. In tale schema l'update è esposto solo ad una parte della popolazione, per la quale viene effettuato il deployment per studiare gli effetti (tramite continuous monitoring) ed eventuali modifiche e migliorie al software, che infine verrà deployato per il resto della popolazione in modo comunque incrementale fino a che l'intera popolazione godrà della feature. Spesso usata per front-end
- **Canary releases**, che studia l'impatto di update relativi al back-end

Tali schemi spesso sono usati di pari passo per le varie sezioni del software, nonché possono essere usati in modo intercambiabile.
Collegato a questi schemi si ha l'approccio basato sull'**user experimentation**, che non è un reale schema di gestione dell'evoluzione del software ma è comunque correlato agli schemi sopra descritti. In questo approccio si studiano diverse varianti del sistema e il loro impatto esponendole agli utenti (perlomeno ad una sottoparte degli stessi in modo incrementale), cercando di capire per l'utente cosa sia meglio e come (si ispira alla *sperimentazione scientifico*). Si hanno quindi più release diverse, per parti di popolazione comparabili, tra le quali si sceglierà la migliore.
Per la seconda tipologia (*Incremental wrt requests*) si ha una divisione a seconda delle richieste fatte dagli utenti (esempio un momento d'uso diverso porta all'uso di componenti diverse), detto *gradual rollout*. Si ha quindi un *load balancer* che permette la coesistenza di due versioni, una nuova e una vecchia, dello stesso servizio. In modo graduale, partendo da pochissime, si passano le richieste alla versione nuova per poter studiare e testare la nuova versione (in caso di problemi il load balancer dirotterà tutte le richieste alla vecchia versione). Alla fine tutto il traffico sarà diretto verso la nuova versione, mentre la vecchia verrà dismessa.
Per la terza tipologia (*Incremental wrt components/replicas*), si ha lo schema del *rolling upgrade*, dove l'upgrade non riguarda un singolo upgrade ma tanti componenti di un sistema distribuito, verificando efficacia di ogni singolo update tramite il continuous monitoring prima di effettuare l'upgrade di un'altra componente. La stessa idea su applica anche a diverse versioni dello stesso prodotto, aggiornandone una prima e poi le altre progressivamente. Le nuove versioni delle componenti "upgradate" devono essere compatibili con quelle ancora prive di upgrade.
Per la quarta tipologia (*Non-incremental with backups*) si ha il **blue/green deployment**, dove vengono isolate due copie della stessa infrastruttura (anche hardware), dove una ospita la versione nuove l'altra la vecchia. Un router ridireziona le richieste degli utenti verso le due unità e quella che ospita la

nuova versione subirà le solite operazioni di test che, se superate, porteranno il router a direzionare verso quella unità, ignorando la vecchia. Se ci sono problemi si fa rollback alla vecchia unità che rimane come backup. Questo schema può essere generalizzato nel **rainbow deployment** dove il momento di coesistenza tra le due versioni (se non più versioni) viene prolungato al fine che vecchie richieste che richiedono una lunga elaborazione vengano elaborate dall'unità vecchia mentre le nuove dall'unità nuova.
In ogni caso le applicazioni devono essere costruite per supportare tutti questi schemi di deployment (a causa di stati delle applicazioni, backward compatibility etc...)

3.2.1 Deployable units

Il caso più tipico in merito alle unità dove fare deployment è il mondo del **cloud**, con **unità virtualizzate e virtual machine (VM)**, dove magari ogni servizio vive in una diversa VM. Si hanno diversi casi in merito a questo tipo di deployment:

- **cloud** basato su **VMs**, dove si ha un'infrastruttura gestita dal cloud provider che gestisce l'hardware e l'hypervisor. Ogni VM, che sono le nostre unità di deployment, ha un sistema operativo arbitrario che lavora con l'hardware mostrato dall'hypervisor. Ogni VM avrà una o più applicazioni e fare deployment porterà all'update di una o più VM. In alcuni casi si fa deployment di intere VM e in altri si modifica il software di una VM già in esecuzione. L'ambiente cloud solitamente è **multi-tenant** (ovvero su una piattaforma unica di un provider si hanno più VM di diverse organizzazioni). Una VM è grossa in quanto contiene un sistema operativo intero e la loro gestione può quindi essere difficoltosa

- **cloud** basato su **containers** che risolvono il problema della grandezza delle VM. In questo caso lo schema è il medesimo ma si ha un **container engine** al posto dell'hypervisor e ogni container non contiene l'intero sistema operativo ma solo il minimo necessario al funzionamento dell'applicazione (il sistema operativo viene condiviso dalla macchina sottostante, riducendo il volume dei singoli containers). In questo caso lo schema di update spesso consiste nel distruggere e ricreare i singoli containers. Anche qui si ha un contesto *multi-tenant*

- **bare metal**, dove i provider offrono direttamente risorse hardware, guadagnando prestazioni ma aumentano anche i costi eco-

nomici , che vengono comunque gestite dal cloud provider. Non si ha virtualizzazione ma accesso diretto alle risorse su cui fare deployment. Questa è una soluzione tipicamente **single-tenant** (sulla macchina gira il software di una sola organizzazione)

- **server dedicati**, un metodo ormai superato con difficoltà causate dall'uso di script, shell e connessione *ftp* completamente autogestiti dall'organizzazione e non da un provider

Il deployment "stile cloud" non è comunque l'unico possibile. Un esempio quotidiano è il deployment di app mobile sui vari store, dove il back-end probabilmente sarà gestito come sopra spiegato mentre l'app in sé viene rilasciata negli store e sarà l'utente finale a fare il deployment installando l'app sul proprio device. Le forme di monitoraggio e di feedback sono spesso diverse e provengono dagli utenti finali stessi.

3.2.2 Monitor

In ambiente cloud ci sono tante soluzioni per il **monitoring**, ad esempio lo **stack di ELK**, formato da:

- **Elasticsearch**
- **Logstash**
- **Kibana**

I dati, ad esempio log o metriche d'uso hardware, vengono raccolti e passano da **Logstash**, finendo in un database, per la memorizzazione di *time series* (serie temporali) di dati (questo in primis per le metriche d'uso che per i log), gestito da **Elasticsearch** e venendo visualizzati da una dashboard grafica, gestita da **Kibana**. Si ha quindi un ambiente di **continuous monitoring**.

3.2.3 DevOps tools

Ogni step del DevOps è gestito tramite moderne tecnologie e tools , con varie alternative per ogni fase (per questo servono figure esperte per ogni step). Vediamo qualche esempio (in ottica più spinta ad un progetto in Java):

- code: Git, Svn, Jira, Eclipse
- build: Apache Ant, Maven, Gradle
- test: JUnit

- release: Jenkins, Bamboo
- deploy: Pupper, Chef, Ansible, SaltStack
- monitor: New Relic, Sensu, Splunk, Nagios

Capitolo 4

Risk management

Partiamo da un semplice esempio:

Esempio 1. *Durante lo sviluppo di un progetto software l'unico dev, insoddisfatto del salario, che conosceva un modulo di importanza critica lascia la società, rallentando in modo serio lo sviluppo del progetto (nonché aumentandone i costi, nel tentativo di cambiare il modulo conosciuto solo dal dev) fino a farlo uscire troppo in ritardo rispetto, ad esempio, alle opportunità di marketing a cui puntava.*

Un esempio del genere non è così raro e il **risk management (*gestione dei rischi*)** si occupa di prevenire questo tipo di complicazioni e fallimenti.

Definizione 1. *Il **risk management** è la disciplina che si occupa di identificare, gestire e potenzialmente eliminate i rischi prima che questi diventino una "minaccia" per il successo del progetto (o anche per eventuali situazioni di revisione del progetto stesso).*

Dobbiamo però dare qualche definizione:

Definizione 2. *Definiamo **rischio** come la possibilità che ci sia un danno.*

Bisogna cercare di prevenire n evento che può portare ad un danno serio (e tanto più è serio il danno tanto è alto il rischio)

Definizione 3. *Definiamo **risk exposure**, che è una grandezza (calcolabile), per calcolare quanto un progetto sia esposto ad un rischio. Viene calcolato come:*

$$RE = P(UO) \cdot L(UO)$$

dove:

- $P(UO)$ *è la probabilità di un* unsatisfactory outcome*, ovvero la probabilità che effettivamente un danno (o comunque un risultato non soddisfacente) sia prodotto*
- $L(UO)$ *è l'entità del danno stesso, ovvero è la perdita per le parti interessate se il risultato non è soddisfacente*

Tanto più un rischio è probabile e tanto più il rischio crea un danno tanto cresce il ***risk exposure****.*

Definizione 4. *Definiamo* ***outcome unsatisfactory (risultato non soddisfacente)*** *come un risultato non positivo che riguarda diverse aree:*

- *l'area riguardante l'esperienza degli utenti, con un progetto che presenta le funzionalità sbagliate, una UI carente, problemi di prestazioni o di affidabilità etc.... In questo caso se i problemi sono gravi si hanno alti rischi, come l'utenza che smette di usare il prodotto, portando al fallimento del prodotto, o anche a conseguenze legali*
- *l'area riguardante i dev, con rischi che per esempio si ritrovano superamento del budget e prolungamenti delle deadlines*
- *l'area riguardante i manutentori, con rischi che per esempio si ritrovano nella qualità bassa di software e hardware*

Se abbiamo un rischio che produce un *outcome unsatisfactory* il primo elemento su cui soffermarsi è lo studio degli eventi che abilitano il rischio, detti **risk triggers**, per evitare che avvengano (comportando di conseguenza che il rischio diventi realtà comportando un *outcome unsatisfactory*, come nell'esempio 1). Sempre in base all'esempio 1 si potrebbe pensare di non sussumere un solo dev con una certa conoscenza o comunque di alzare la paga, per evitare di attivare i *risk triggers*.
Abbiamo quindi due principali **classi di rischio**:

1. **process-related risks**, rappresenti rischi con impatto negativo sul processo e sugli obbiettivi di sviluppo, come ritardi o superamento di costi
2. **product-related risks**, rappresenti rischi con impatto sul prodotto e su obbiettivi del sistema funzionali o meno, come fallimenti riguardanti la qualità del prodotto (sicurezza, prestazioni etc...) o la distribuzione dello stesso

Entrambe le classi possono portare al fallimento del progetto e quindi vanno gestite entrambe.
Bisogna quindi imparare a gestire i rischi. Si hanno principalmente due fasi:

1. una prima fase riguardante il **risk assessment (*valutazione del rischio*)**. In questa fase si hanno:
   - **risk identification**, ovvero l'identificazione dei rischi
   - **risk analysis**, ovvero l'analisi dei rischi identificati (tramite calcolo del *risk exposure* e studio dei triggers)
   - **risk prioritization**, ovvero la prioritizzazione dei rischi analizzati, per focalizzarsi sui più pericolosi per poi scalare ai meno pericolosi

   Alla fine si produce una lista ordinata sulla pericolosità dei rischi
2. una seconda fase riguardante il **risk control** e consiste, in primis, sul **risk management planning**, producendo piani di controllo di due tipi:
   (a) **piani di management**, per la gestione del rischio prima che si verifichino
   (b) **piani di contingency**, per il contenimento di rischi divenuti realtà qualora il *piano di management* fallisca, sapendo cosa fare a priori in caso di emergenza

   Si hanno quindi due sotto-fasi per i due tipi di piani:
   (a) **risk monitoring**
   (b) **risk resolution**

Queste due fasi vengono ciclicamente ripetute durante il ciclo di vita dello sviluppo di un software.

4.1 Risk identification

Si studia come identificare i rischi.
È un'operazione complessa legata alla competenza degli analisti. Un modo comune di farlo è usando delle **check-list**, liste che includono un insieme di rischi plausibili comuni a molti progetti. L'analista scorre tale lista cercando rischi che possono essere applicati al progetto in analisi.

Esempio 2. *Una lista di 10 punti comoda nell'ambito dello sviluppo include:*

1. *problemi con il personale*
2. *budget e deadlines irrealistici*
3. *sviluppo delle funzionalità sbagliate*
4. *sviluppo della UI sbagliata*
5. *gold-plating, ovvero aggiungere più funzionalità del necessario o usare tecnologie troppo avanzate*
6. *continue modifiche ai requisiti*
7. *problemi in componenti software fornite da esterni*
8. *problemi con task svolti da esterni*
9. *problemi con le prestazioni real-time*
10. *voler superare i limiti delle tecnologie attuali oltre le regole e le capacità della computer science*

Alcuni rischi, in astratto, possono verificarsi sempre ma va preso in considerazione solo per **motivi specifici identificabili** nel mio progetto.
Si hanno altri metodi per identificare i rischi:

- riunioni di confronto, *brainstorming* e *workshop*
- confronto con altre organizzazioni e con altri prodotti

4.2 Risk analysis

Per analizzare i rischi si sfrutta esperienza delle reali probabilità che un rischio diventi realtà. Anche in questo si hanno degli schemi su cui basarsi, come *modelli di stima dei costi*, *modelli delle prestazioni*, etc... basati su *simulazioni*, *prototipi*, *analogie con altri progetti* e *check-list* (con stime di probabilità e danno).
Le stime sono molto specifiche sul singolo progetto.
L'analisi dei rischi può anche comportare lo studio delle decisioni da prendere al fine di minimizzare il **risk exposure**, scegliendo o meno tra varie opzioni, scegliendo in modo "guidato" dai rischi. A tal fine si usano i *decision tree* con la radice che rappresenta il problema (un esempio si ha in figura

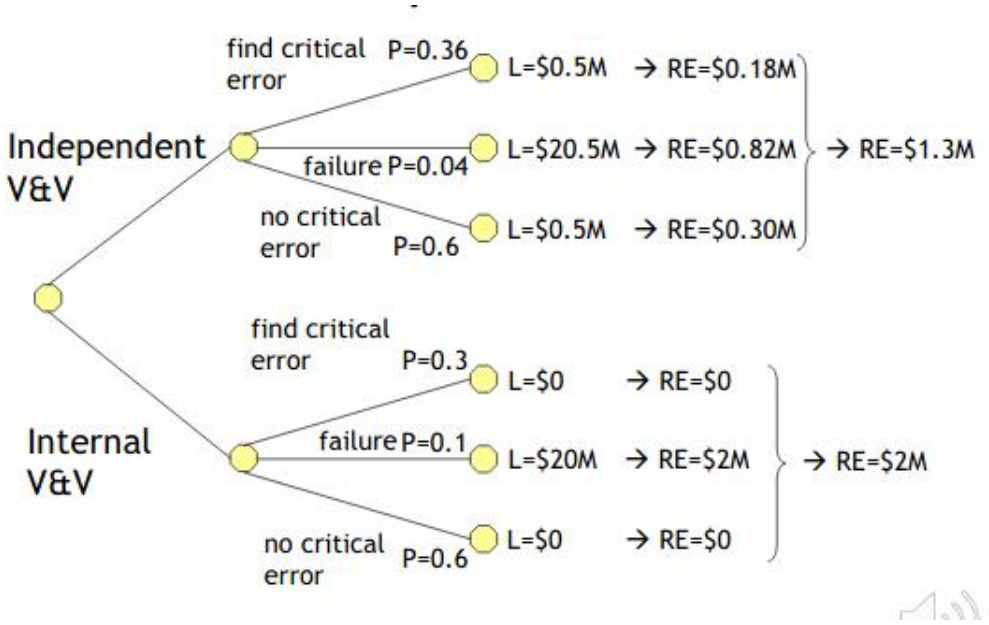

Figura 4.1: Esempio di decision tree

4.1). Si hanno di volta in volta i vari scenari, con le stime di probabilità di trovare un errore critico, di fallimento, di non trovare errori critici etc.... Tali probabilità verranno usate per il calcolo del *risk exposure* insieme ad un quantificatore di $L(UO)$ spesso pari all'effettivo costo che conseguirebbe al risultato ottenuto. Infine i vari *risk exposure* di ogni caso vengono sommati per ottenere il *risk exposure* finale. Si può fare un'**analisi di sensitività** cambiando le percentuali o i costi al fine di ottenere un certo risultato, in modo da capire come si dovrebbe comportare.
Ragionando sulle cause dei rischi usiamo il cosiddetto **risk tree** (esempio in figura 4.2). Questo albero ha come radice il rischio. Ogni nodo, detto **failure node**, è un evento che si può scomporre "via via" in altri eventi, fino alle foglie. La scomposizione è guidata da due tipi di **nodi link**:

1. **and-node**, dove i figli di tali nodi sono eventi legati dal un *and*
2. **or-node**, dove i figli di tali nodi sono eventi legati dal un *or*

Nodi And/Or vengono rappresentati tramite i simboli delle porte logiche.
Dato un *risk tree* cerco le combinazioni di eventi atomici che possono portare al rischio. Per farlo si esegue la **cut-set tree derivation**, ovvero, partendo dalla radice, si riporta in ogni nodo la combinazione di eventi che possono produrre il fallimento e si vanno a calcolare le varie combinazioni degli eventi

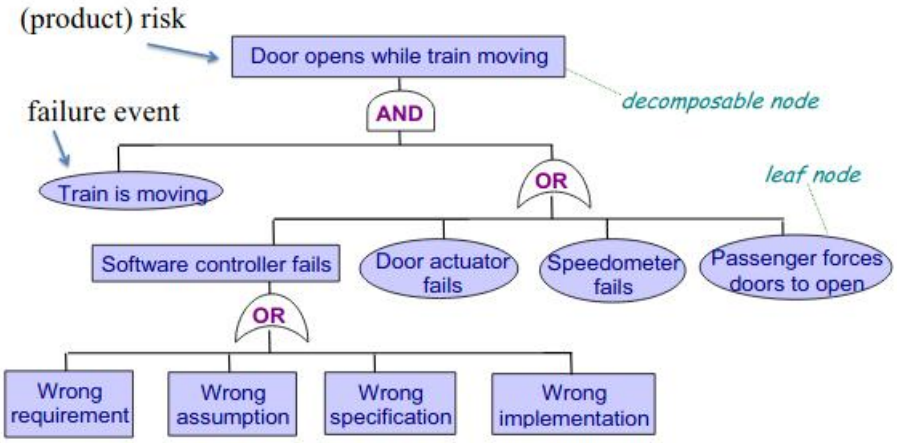

Figura 4.2: Esempio di risk tree

foglia. Praticamente si deriva un insieme di eventi non scomponibili sulle combinazioni dell'*and*.

4.3 Risk prioritization

Bisogna capire quali rischi sono più "rischiosi" degli altri. Per farlo si pongono i valori di $P(UO)$ e $L(UO)$ in un range, per esempio, da 1 a 10, ricalcolando il *risk exposure*. Una volta fatto si lavora in base al *risk-exposure* (che può anche essere un intervallo se le probabilità o i costi sono in un certo intervallo). Si procede plottando i dati su un piano con $P(UO)$ sull'asse delle y e $L(UO)$ su quello delle x e facendo lo *scatterplot* degli eventi (ponendoli quindi come punti in base a $P(UO)$ e $L(UO)$). Qualora i valori siano in un range si rappresentano con un segmento tra i due valori limite di *risk exposure*. Con delle curve posso identificare zone di rischio diverse in base al *risk exposure* per poter catalogare gli eventi. Solitamente alla fase di *risk control* passano una decina di eventi.

4.4 Risk control

Bisogna quindi capire come gestire i rischi.
Per ogni rischio bisogna definire e documentare un piano specifico indicante:

- cosa si sta gestendo
- come mitigare il rischio e quando farlo

- di chi è la responsabilità
- come approcciarsi al rischio
- il costo dell'approccio al rischio

Anche in questo caso ci vengono incontro liste e *check-list* con le tecniche di *risk management* più comuni in base al rischio specifico.
Ci sono comunque strategie generali:

- lavorare sulla probabilità che il rischio avvenga, sulla probabilità dei triggers. Bisogna capire come diminuire la probabilità
- lavorare, nel limite del possibile, sull'eliminazione stessa del rischio
- lavorare sulla riduzione della probabilità di avere conseguenze al danno, non viene quindi ridotto il rischio
- lavorare, nel limite del possibile, sull'eliminazione stessa del danno conseguente al rischio
- lavorare sul mitigare le conseguenze di un rischio, diminuendo l'entità del danno

Bisogna anche studiare le contromisure, da scegliere e attivare in base alla situazione. Si hanno due metodi quantitativi principali per ragionare quantitativamente sulle contromisure:

1. **risk-reduction leverage**, dove si calcola quanto una certa contromisura può ridurre un certo rischio, secondo la seguente formula:
$$RRL(r, cm) = \frac{RE(r) - RE\left(\frac{r}{cm}\right)}{cost(cm)}$$
dove r rappresenta il rischio, cm la contromisura e $\frac{r}{cm}$ la contromisura cm applicata al rischio r. Calcolo quindi la differenza di *risk exposure* avendo e non avendo la contromisura e la divido per il costo della contromisura.
La miglior contromisura è quella con il RRL maggiore, avendo minor costo e maggior efficacia dal punto di vista del *risk exposure*

2. **defect detection prevention**, più elaborato del primo è stato sviluppato dalla NASA. Questo metodo confronta le varie contromisure, confrontando anche gli obiettivi del progetto, in modo

quantitativo facendo un confronto indiretto, producendo matrici in cui si ragiona in modo indipendente sulle singole contromisure e sui singoli rischi ma confrontando anche in modo multiplo.

Si ha un ciclo a tre step:

(a) elaborare la matrice di impatto dei rischi, detta **risk impact matrix**. Questa matrice calcola l'impatto dei rischi sugli obiettivi del progetto. I valori della matrice, ovvero $impact(r, obj)$ (che ha per colonne i rischi r e righe gli obiettivi del progetto obj) variano da 0, nessun impatto, a 1, completa perdita di soddisfazione (e totale non raggiungimento dell'obiettivo indicato). Ogni rischio viene accompagnato dalla probabilità P che accada. Ogni obiettivo è accompagnato dal **peso** W che ha nel progetto (la somma di tutti i pesi è pari a 1). Si possono calcolare altri valori di sintesi. In primis la **criticità** di un rischio rispetto a tutti gli obiettivi indicati:

$$criticality(r) = P(r) \cdot \sum_{obj}(impact(r, obj) \cdot W(obj))$$

La criticità sale se sale l'impatto e se sale la probabilità del rischio.
Un altro dato è la **perdita di raggiungimento** di un obiettivo qualora tutti i rischi si verificassero:

$$loss(obj) = W(obj) \cdot \sum_{r}(impact(r, obj) \cdot P(r))$$

(b) elaborare contromisure efficaci per la matrice. In questa fase si usa il fattore di criticità del rischio. Viene prodotta una nuova matrice con colonne pari ai rischi (con probabilità e criticità) e righe pari alle contromisure. I valori saranno le riduzioni di rischio di una contromisura cm sul rischio r ($reduction(cm, r)$). La riduzione va da 0, nessuna riduzione, a 1, rischio eliminato. Si possono calcolare altri valori di sintesi. Possiamo calcolare la **combineReduction**, che ci dice quanto un rischio viene ridotto se tutte le contromisure sono attivate:

$$(combineReduction(r) = 1 - \prod_{cm}(1 - reduction(cm, r))$$

Un altro valore è l'**overallEffect**, ovvero l'effetto di ogni contromisura sull'insieme dei rischi considerato:

$$overalleffect(cm) = \sum_{r}(reduction(cm, r) \cdot criticality(r))$$

si avrà effetto maggior riducendo rischi molto critici

(c) determinare il bilanciamento migliore tra riduzione dei rischi e costo delle contromisure. Bisogna considerare anche il costo di ogni contromisure e quindi si fa il rapporto tra effetto di ciascuna contromisura e il suo costo e scegliendo il migliore

Analizzando il *contingency plan* viene attuato qualora il rischio si traduca in realtà.
I passa quindi al **risk monitoring/resolution**. Queste due parti sono tra loro integrate. I rischi vanno monitorati e occorrenza vanno risolti il prima possibile. Tutte queste attività sono costose e si lavora su un insieme limitato di rischi, una decina.

Capitolo 5

Capability Maturity Model Integration

Il **Capability Maturity Model Integration (*CMMI*)** che è un "programma" di formazione e valutazione per il miglioramento a livello di processo gestito dal *CMMI Institute*.
Bisogna prima introdurre il concetto di **maturità dei processi**. Questa nozione è abbastanza intuitiva. La probabilità di portare a termine un progetto dipende dalla *maturità del progetto* e la maturità dipende dal grado di controllo che si ha sulle azioni che si vanno a svolgere per realizzare il progetto. Si ha quindi che:

- il progetto è **immaturo** quando le azioni legate allo sviluppo non sono ben definite o ben controllate e quindi i dev hanno troppa libertà che rischia di degenerare in una sorta di anarchia nel controllo del processo di sviluppo, alzando la probabilità di fallimento
- il progetto è **maturo** quando le attività svolte sono ben definite, chiare a tutti i partecipanti e ben controllate. Si ha quindi un modo per osservare quanto si sta svolgendo e verificare che sia come pianificato, alzando le probabilità di successo e riducendo quelle di fallimento

Risulta quindi essenziale ragionare sulla *maturità del processo*.
La **maturità del processo** è definita tramite un insiemi di livelli di maturità con associate metriche per gestire i processi, questo è detto **Capability Maturity Model (*CMM*)**. In altri termini il modello CMM è una collezione dettagliata di *best practices* che aiutano le organizzazioni a migliorare e governare tutti gli aspetti relativi al processo di sviluppo, dalla gestione dei rischi al testing, dal design al project management etc. . ..

Un processo migliore porta ad un prodotto migliore.

La storia dei CMMs parte dagli anni novanta e porta alle versione di CMMI del 2018 che andremo ad analizzare.
Analizziamo quindi nel dettaglio il modello:

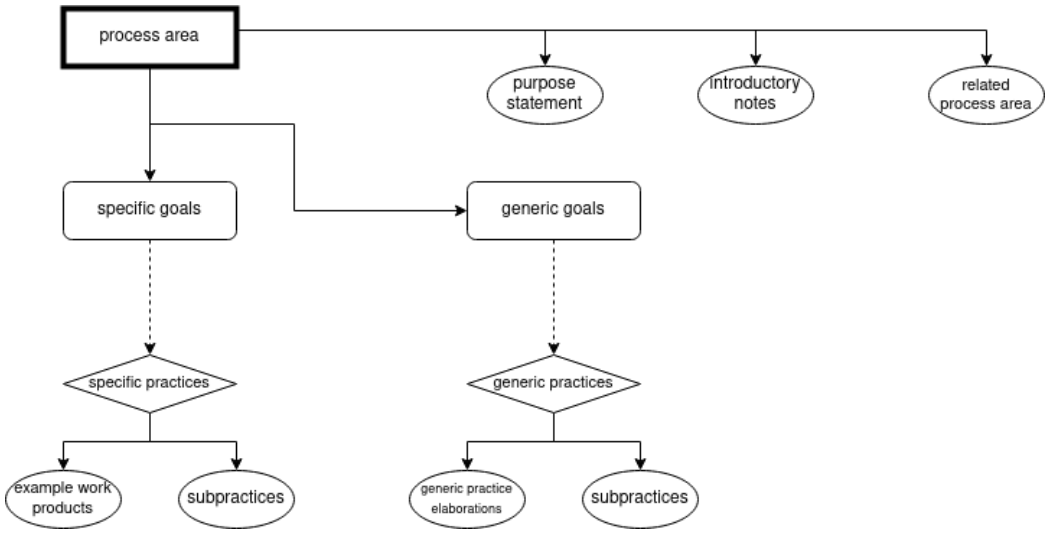

Figura 5.1: Diagramma dei componenti del CMMI, secondo l'organizzazione standard. Nei rettangoli (tranne *process area*) abbiamo i *required*, nei rombi gli *expected* e negli ovali gli *informative*

All'interno del diagramma (che rappresenta letteralmente un documento) notiamo:

- **process area**, che racchiude al suo interno una collezione di pratiche organizzate secondo obiettivi e riguarda una certa area del processo. Nel CMMI abbiamo 22 diverse *process area*, tra cui *configuration management*, *project planning*, *risk management* etc... Nel diagramma si ha lo studio di una generica *process area*
- ciascuna **process area** ha:
  - un **purpose statement**, che descrive lo scopo finale della *process area* stessa
  - un **introductory notes**, con nel note introduttive che descrivano i principali concetti della *process area*
  - un **related process area**, se utile, con la lista delle altre *process area* correlate a quella corrente

- le **process area** si dividono in due *tipologie di obiettivi*:

    1. **specific goals**, ovvero gli obiettivi specifici della singola *process area* in questione. Questi obiettivi caratterizzano la *process area*
    2. **generic goals**, ovvero gli obiettivi comuni a **tutte** le **process area**. Questi obiettivi rappresentano quanto la *process area* sia ben integrata e definita nel contesto del processo ma questi criteri sono generali

- all'interno di ogni **specific goals** (per questo la freccia tratteggiata) abbiamo una serie di **specific practices**, ovvero quelle azioni che se svolte permettono di raggiungere quell'obiettivo specifico e, a loro volta, tali pratiche sono organizzate in:

    – **example work product**, ovvero elenchi di esempi di prodotti che possono essere generati attraverso l'adempimento delle pratiche

    – **subpractices**, ovvero pratiche di "grana più fine"

- all'interno di ogni **generic goals** (per questo la freccia tratteggiata) abbiamo una serie di **generic practices**, comuni a tutti, con le pratiche che devono essere svolte per gestire positivamente una qualsiasi *process area* e, a loro volta, tali pratiche sono organizzate in:

    – **generic practices elaborations**, ovvero ulteriori informazioni di dettaglio per la singola pratica

    – **subpractices**, ovvero pratiche di "grana più fine"

    Tra i principali *generic goals (GG)* abbiamo:

    – *GG1*: raggiungere i *specific goals*, tramite l'esecuzione delle *specific practices*

    – *GG2*: "ufficializzare" un *managed process*, tramite training del personale, pianificazione del processo, controllo dei *work product* etc...

    – *GG3*: "ufficializzare" un *defined process*, tramite la definzione rigorosa del progetto e la raccolta di esperienze legate al processo

Per capire quanto un processo software è organizzato secondo questo standard bisogna "mappare" quali goals e quali pratiche si stanno perseguendo e seguendo e usare CMMI non solo come "ispirazione" ma come vero e proprio **standard** per definire le azioni da svolgere nonché per confrontare il nostro operato e studiarlo qualitativamente. Lo studio qualitativo mi permette di stabilire la **maturità** del progetti, secondo un certo livello di *compliance*, detto **CMMI level**. Tale qualità che può essere certificata da enti certificatori appositi (si ha anche una repository pubblica con i livelli di *compliance* di diversi progetti di organizzazioni famose).
Studiamo a fondo questi livelli di maturità e la loro codifica.
Si hanno due linee di sviluppo/miglioramento:

1. **capability levels (*CL*)**, che cattura e rappresenta quanto bene si sta gestendo una particolare *process area* (quindi ciascuna *process area* può raggiungere un diverso CL). Quindi per una singola *process area* mi dice quanto bene sto raggiungendo i *generic goals* (e di conseguenza anche i vari *specific goals*, in quanto quelli generici impongono il controllo di quelli specifici). A livello di grafico punta all'ovale con *specific goals*. Il CL ha valore da 0 a 3:

    - **level 0: *incomplete***, dove probabilmente non si stanno nemmeno svolgendo tutte le pratiche richieste per quella *process area* o sono state svolte solo parzialmente
    - **level 1: *performed***, dove si eseguono le pratiche e i vari *specific goals* sono soddisfatti
    - **level 2: *managed***, dove oltre alle pratiche si ha anche una gestione (controllo, allocazione di risorse, monitoring, review etc...) delle attività stesse, come indicato nello standard. Si ha una policy per l'esecuzione delle pratiche
    - **level 3: *defined***, dove l'intero processo è ben definito secondo lo standard, descritto rigorosamente e si ha un processo completamente su misura dell'organizzazione

    I CL di ciascuna *process area* possono essere rappresentate su un diagramma a barre, dove viene indicato il CL attuale e il **profile target**, ovvero il livello a cui quella *process area* deve arrivare (che non è per forza il terzo).

2. **maturity levels (*ML*)**, che cattura il livello raggiunto dall'intero processo di sviluppo ragionando su tutte le *process area* attivate.

Rappresenta quanto bene si sta lavorando sull'insieme intero della *process area*. A livello di grafico punta direttamente all'elemento specificante la *process area*. Il ML ha valore da 1 a 5:

- **level 1:** ***initial***, dove si ha un processo gestito in modo caotico
- **level 2:** ***managed***, dove si ha già un processo ben gestito secondo varie policy
- **level 3:** ***defined***, dove si ha un processo ben definito secondo lo standard aziendale
- **level 4:** ***quantitatively managed***, dove si stanno anche raccogliendo dati che misurano quanto bene sta funzionando il processo, facendo quindi una misura quantitativa dello stesso
- **level 5:** ***optimizing***, dove grazie alle informazioni raccolte nel livello precedente ottimizzo il processo, in un'idea di *continuous improvement* del progetto stesso

Gli ultimi due livelli sono davvero difficili da raggiungere.

Rapportiamo quindi CL e ML in base avari blocchi di *process area*, che scalano per "importanza":

- il processo è a ML=2 se tutte le *process area* del blocco (che si riferisce alle *process area* riferite al blocco *planned & executed*) sono svolte tutte a CL=2
- il processo è a ML=3 se aggiungo le *process area* del blocco *org. standard* e queste sono tutte a CL=3
- il processo è a ML=4 se aggiungo le due *process area* del blocco *quantitative PM* che devono essere svolte a CL=3
- il processo è a ML=5 se aggiungo le ultime due *process area*, ovvero quelle del blocco *continuous improvement*, che devono essere svolte a CL=3

Si possono confrontare CMMI e le pratiche agili.
Ciò che viene svolto ai livelli 2 e 3 (con qualche piccolo adattamento) di *maturity level* si fa ciò che viene fatto anche coi metodi agili. In merito ai livelli 4 e 5 di *maturity level* si hanno pratiche che non rientrano nell'ottica dei metodi agili. Quindi un'organizzazione può usare i metodi agili ed essere standardizzata rispetto CMMI raggiungendo un *maturity level* 2 o 3.
CMMI è quindi uno standard industriale con certificazioni ufficiali.

Capitolo 6

Requirements engineering

Quando si parla di **requirements engineering (*RE*, ingegnerizzazione dei requisiti)** di fatto ci si concentra sulla comprensione di come una soluzione software si deve comportare per risolvere un certo problema. In questo senso bisogna prima comprendere quale sia il **problema** da risolvere e in quale **contesto** tale problema si verifica, per poter arrivare ad una soluzione corretta ed efficacie a problemi "reali". Bisogna ben comprendere il problema, non si parla quindi della progettazione in se, ma del "cosa" deve fare il software.

Esempio 3. *Vediamo un esempio banale.*
Bisogna sviluppare il software per l'adaptive cruise control. In questo caso si ha:

- *il problema che consiste nel seguire correttamente la macchina che si ha davanti in autostrada*
- *il contesto che consiste in:*
  - *guida della macchina*
  - *intenzioni del guidatore*
  - *norme di sicurezza*
  - *etc...*

Esempio 4. *Vediamo ora l'analogia classica per spiegare il RE, il* problema mondo *e la* soluzione macchina*.*
Si hanno:

- *il **mondo**, con un problema derivante dal mondo reale, mondo stesso che produce tale problema che bisognerà risolvere con un calcolatore. Si hanno all'interno:*

  - componenti umane*, ovvero staff, operatori, organizzazioni etc...*
  - componenti fisiche*, ovvero device, software legacy, madre natura etc...*

- *la* ***macchina****, che bisogna sviluppare per risolvere il problema. Si ha necessità quindi di:*
  - software *da sviluppare o acquistare*
  - piattaforma hardware/software*, con eventuali device annessi*

- requirements engineering *che si occupa di:*
  - *definire gli effetti della* macchina *sul problema del* mondo
  - *definire assunzioni e proprietà principali del* mondo *stesso*

Macchina e mondo possono condividere alcuni componenti, con la macchina che modifica il mondo (nell'esempio precedente la macchina che disattiva il cruise control agisce sia a livello software che a livello del mondo esterno). Nei RE studiamo quindi il mondo*, senza definire come funziona internamente la macchina (nelle parti che non interagiscono direttamente col mondo, per esempio scelte di design etc…).*

Quando si parla di RE è bene distinguere due elementi:

1. ogni volta che prendiamo in consideriamo un problema esiste sempre un **system-as-is**, ovvero un sistema preesistente che già risolve il problema (anche se magari in modo non efficiente o qualitativamente sufficiente). Si ha quindi sempre un sistema da cui partire (esempio banale per il cruise control è dire che in realtà la soluzione al problema già esiste con la guida senza cruise control)

2. esiste sempre un **system-to-be**, ovvero il sistema che si andrà a realizzare. In altre parole è il sistema quando la *macchina/software* ci opera sopra

Studiare il *system-as-is* è essenziale per poter lavorare al *system-to-be*.

Definizione 5. *Il requirements engineering è, formalmente, un insieme di attività:*

- *per esplorare, valutare, documentare, consolidare, rivisitare e adattare gli obiettivi, le capacità, le qualità, i vincoli e le ipotesi su un* system-to-be
- *basate sul problema sorto dal system-as-is e sulle nuove opportunità tecnologiche*

L'output di queste attività è un ***documento di specifica dei requisiti*** *con tutto ciò che soddisfa il sistema. Se l'output non è un singolo documento si ha una collezione di singoli requisiti, che nel metodo agile sono storie/cards e in altri metodi un repository centrale con un db condiviso contenete i vari requisiti.*
In ogni caso si ha un insieme di requisiti su come si deve comportare il sistema che si realizza, descrivendo quanto descritto nei punti precedenti.

In un modello simil-cascata di sviluppo software il RE è una delle primissime attività, subito dopo quelle di definizione del sistema e di business plan. Per gli aspetti tecnici è probabilmente la prima attività svolta, occupandosi di *ottenere il giusto sistema da sviluppare (per fare la cosa giusto)*, prima di design, implementazione ed evoluzione software etc... che si occupano di *ottenere il software giusto, sviluppandolo nel modo corretto.*
Errare nel RE può portare ad un ottimo software che risolve i problemi sbagliati (o non tutti i problemi che dovrebbe risolvere). Sbagliare i RE è una causa di fallimento del progetto (anche in un contesto non agile).
Lavorare sui RE non è semplice per diversi motivi:

- si deve ragionare su tante versioni del sistema:
  - as-is
  - to-be
  - to-be-next, volendo essere lungimiranti per il comportamento del sistema, sapendo e prevedendo evoluzioni future
- si lavora in *ambienti ibridi*, tra umani, leggi, device, policy, leggi della fisica, etc..., quindi in un contesto eterogeneo che va compreso a fondo (ignorare degli aspetti può portare al fallimento)
- si hanno diversi aspetti funzionali, qualitativi e di sviluppo

- si hanno diversi livelli di astrazione, con obiettivi strategici a lungo termine di inserimento sul mercato e dettagli operazionali
- si hanno tanti stakeholders, quindi con diverse parti interessati di cui risolvere problemi e interessi (con potenziali conflitti tra i vari stakeholders)
- si hanno tante attività tecniche legate l'una con l'altra:
  - conflict management
  - risk management
  - evaluation of alternatives
  - prioritization
  - quality assurance
  - change anticipation

  per capire cosa fare, in che ordine, con che rischi, che livelli di qualità, etc...

6.1 Tipi di requisiti

Bisogna anche ragioni sui tipi di requisiti su cui si deve lavorare.
Una prima differenza si ha nel modo in cui sono scritti i requisiti. Si hanno quindi:

- **descriptive statements (*dichiarazioni descrittive*)**, che indicano dei requisiti non negoziabili, rappresentano dei comportamenti derivanti dalle leggi del mondo su cui lavora la macchina. Si ha quindi zero margine di modifica
- **prescriptive statements (*dichiarazioni prescrittive*)**, che indicano requisiti negoziabili

Esempio 5. *Vediamo un esempio.*
Per il primo modo si ha:

La stessa copia non può essere presa in prestito da due persone diverse contemporaneamente

per il secondo:

Un cliente abituale non può prendere in prestito più di tre libri contemporaneamente

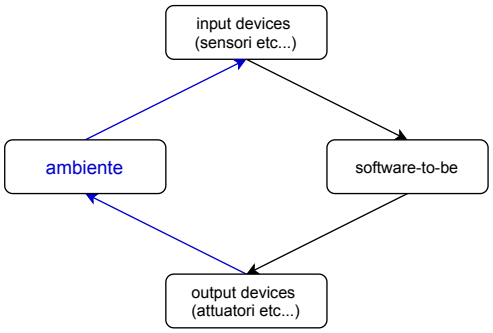

Figura 6.1: Raffigurazione del modello Parnas95

Entrambi sono importanti e vanno considerati. Si possono avere requisiti non ovvi, o lo sono in un contesto specialistico e quindi spesso non ovvi a chi lavora sul software (che generalmente non è del settore specialistico), che magari nemmeno sa che esistono.
I requisiti possono inoltre differire per gli elementi che prendono in considerazione. Posso avere requisti:

- che riguardano come si comporta l'ambiente, per capirne il funzionamento (che andrà ad influenzare il software)
- che studiano come si comporta il software nell'ambiente, studiando i cosiddetti *shared fenomena*

I requisti comunque non riguardano mai il comportamento interno del software (variabili, error code etc...).
I requisti che riguardano l'ambiente possono essere di due forme:

1. **domain properties**, ovvero proprietà riguardanti l'ambiente, immutabili
2. **assumptions**, ovvero le assunzioni fatte su come è fatto l'ambiente. Il software funzionerà solo negli ambienti che soddisfano quella certa assunzione (per esempio un numero massimo di chiamate API a cui il sistema è in grado di rispondere). Le assunzioni descrivono gli ambienti compatibili con il software. Se troppo restrittive possono portare al fallimento del progetto, che riguarderebbe pochissimi casi particolari

In ottica di un ragionamento di interazione tra sistema software e ambiente citiamo il **modello Parnas95**, della metà degli anni novanta.

In modo elementare il modello schematizza questa interazione software-ambiente in modo da esplicitare i device usati dal software-to-be per interagire con ambiente. Si ha uno schema composto da 4 elementi:

1. dei device di input, simil sensori, che percepiscono l'ambiente (ad esempio richieste che arrivano dal web)

2. dei device di output che permettono di interagire con l'ambiente (che possono ad esempio essere le risposte renderizzate nel browser)

Nel caso, ad esempio, di sistemi embedded, di domotica o simili tali device diventano parecchio complessi.
È utile quindi fare una distinzione per quanto concerne i requisiti del software. Abbiamo due famiglie principali:

1. **requisiti funzionali**, che indicano le funzionalità che un sistema, *system-to-be*, deve implementare, cosa deve essere in grado di fare. Non sono prevedibili prima di studiare il sistema

2. **requisiti non funzionali**, che indicano delle qualità o dei vincoli sulle funzionalità e quindi sui requisiti funzionali. Possono essere in termini di prestazioni, sicurezza, qualità etc... delle funzionalità. Questa famiglia di aspetti non funzionali è più o meno standard

Esempio 6. *Posso avere il requisito funzionale:*

Il cliente è in grado di prendere in prestito al massimo tre libri

e quello non funzionale:

Un cliente può prendere in prestito i libri solo dopo il riconoscimento

Si ha quindi una tassonomia dove si trovano organizzati aspetti non funzionali tali per cui, dopo aver individuato le funzionalità, possono essere usate per chiedersi se, per ciascun aspetto, esso è rilevante, individuando nuovi requisiti non funzionali da includere nell'insieme dei requisiti.
Per alcuni tipi di progetti requisiti funzionali e non sono difficilmente distinguibili, spesso mischiandosi (si pensi, ad esempio, alla sicurezza nello sviluppo dio un firewall).

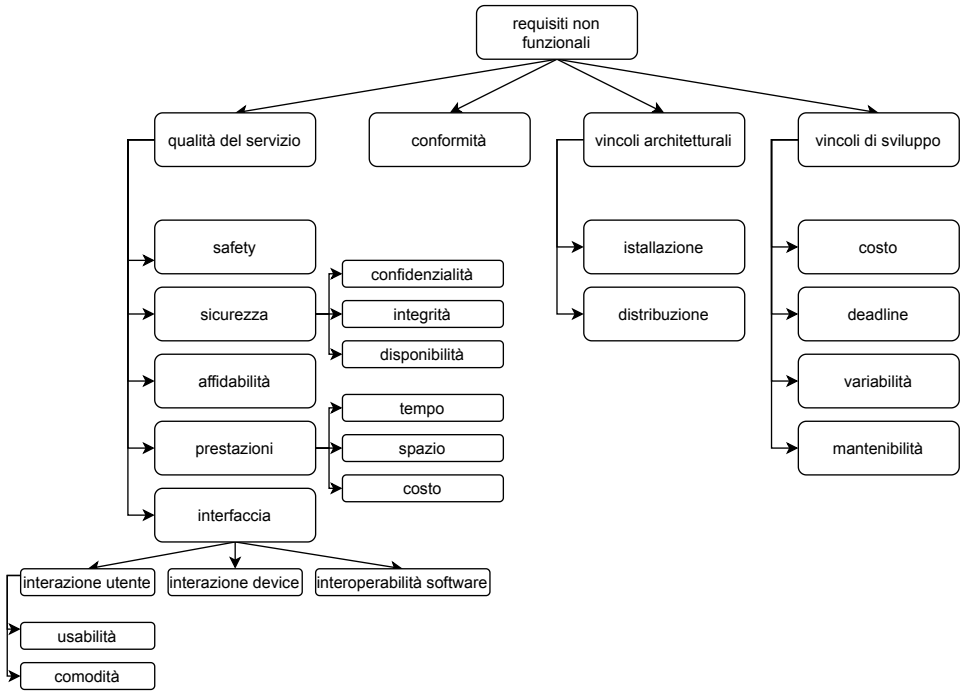

Figura 6.2: Tassonomia dei requisiti non funzionali

6.2 Qualità dei requisiti

Si hanno diversi aspetti di cui preoccuparsi nell'analisi dei requisiti. Si hanno tante qualità indispensabili:

- **completezza** di quanto descriviamo, descrivendo tutti i requisiti rilevanti del progetto, identificando tutti i comportamenti del sistema e documentarli in modo adeguato. È una qualità virtualmente irraggiungibile in modo assoluto, non è infatti verificabile. Inoltre i requisiti variano al proseguire del progetto, interagendo anche con gli stakeholders, rendendo questo uno degli aspetti più difficili da studiare
- **consistenza** dei requisti, senza conflitti. Nella mole di requisiti si possono purtroppo facilmente introdurre inconsistenze
- **non ambiguità** di ciò che si scrive. Tutto deve essere chiaro e non soggetto ad interpretazione
- **misurabilità**, quando rilevante, e quindi non basato su interpretazioni vaghe ma si misure concrete e specifiche

- **fattibilità**, ovvero non si devono avere requisiti basati su funzionalità irrealizzabili
- **comprensibilità**
- **buona struttura**
- **modificabilità**, con possibilità di avere il risultato mantenibile del tempo
- **tracciabilità**, individuando tutti gli artefatti ottenibili come conseguenza e tracciandoli. Si tracciano anche le dipendenze tra requisiti

Vediamo quindi gli errori che si fanno quando si va ad identificare i requisiti, per poter poi studiare tecniche per evitare, nel limite del possibile, tali errori:

- **omissioni**, non riuscendo ad identificare qualche requisito. Anche un riconoscimento tardivo è un problema in quanto comporta la modifica del documento, non sempre facile
- **contraddizioni**, avendo conflitti tra i requisiti (anche solo, banalmente, per i requisiti di un treno si potrebbe avere che non si possono aprire le porte tra due stazioni ma anche se si possono aprire le porte in caso d'emergenza)
- **inadeguatezza**, avendo requisiti che non sono adeguati per un determinato problema
- **ambiguità**, avendo requisiti interpretabili
- **non misurabilità** di certi requisiti (specialmente non funzionali), che comportano difficoltà di gestione, precludendo confronti etc. . .

Si hanno anche altri tipi di errore:

- essere **troppo specifici** nella definizione dei requisiti includendo anche comportamenti interni al software che non dovrebbero essere descritti in questa fase. Bisogna dire quali funzionalità bisogna realizzare, non come
- descrivere requisiti **non implementabili** considerando vincoli temporali o di costo
- descrivere requisti **complessi da leggere**, ad esempio con un uso eccessivo di acronimi

- avere **poca struttura** nella stesura dei requisiti, a livello visivo. Un documento tecnico deve essere facile da gestire e da usare
- se si produce un documento (e non nel caso dell'uso del db) bisogna evitare di fare riferimento a requisiti che non sono stati ancora descritti, ovvero evitando il **forward reference**
- evitare "**rimorsi**" di non aver definito nel momento giusto certi concetti che magari erano stati usati, senza definizione, precedentemente. Questi vanno definiti all'inizio del documento (nelle slide si dice tra parentesi)
- evitare la **poco modificabilità** del documento. È buona norma avere delle "costanti simboliche", definite all'inizio, a cui fare riferimento nel documento, in modo che un'eventuale modifica si rifletta su tutto il documento
- evitare l'**opacità/logica di fondo/rationale/motivazioni** dei requisti, in modo che sia chiaro il perché esso è stato incluso, portando a mettere in discussione requisiti in realtà sensati a cui si arriverebbe comunque dopo ulteriore analisi, dopo aver perso ulteriore tempo

6.3 Il processo RE

Vediamo le principali fasi del RE.
Si hanno 4 fasi a spirale:

1. **domain understanding & elicitation**
2. **evaluation & agreement**
3. **specification & documentation**
4. **validation & verification**

domain understanding & elicitation In questa fase si hanno due parti principali:

1. **domain understanding**
2. **requirements elicitation**

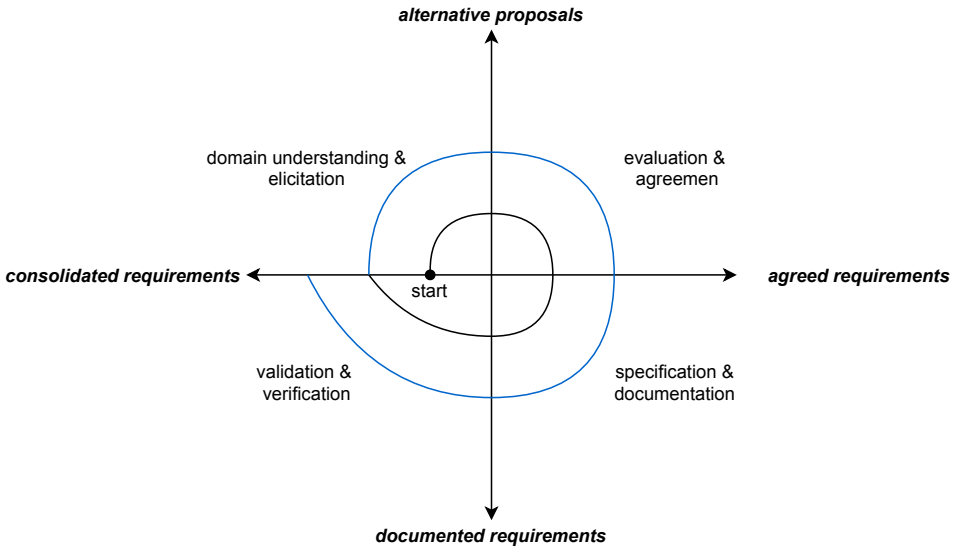

Figura 6.3: rappresentazione a spirale delle quattro fasi del RE

Nella *domain understanding* si ha:

- lo studio del *system-as-is* in ottica di studio del dominio applicativo, studio del business organizzazione che vuole il prodotto. Si studiano anche forze e debolezze del *system-as-is*. Si studia il dominio applicativo, spesso sconosciuto e complesso, per ottenere il miglior *system-to-be* possibile
- si identificano gli stakeholders (e quindi tutte le parti interessate) del progetto per poter capire a fondo interessi e fini

Come **output** della *domain understanding* si hanno:

- le sezioni iniziali per la bozza di proposta preliminare
- il glossario dei termini

Passando alla *requirements elicitation* si uno studio più approfondito nel mondo:

- si ha un'ulteriore analisi dei problemi legati al *system-as-is*, alla ricerca di sintomi, cause e conseguenze
- vengono identificati, grazie all'aiuto degli stakeholders:
  - opportunità tecnologiche

  - condizioni del mercato
  - obiettivi di miglioramento
  - vincoli, organizzativi e tecnici, del *system-as-is*
  - alternative per raggiungere l'obiettivo e assegnare le responsabilità
  - scenari di ipotetica interazione software-ambiente
  - requisiti del software
  - assunzioni sull'ambiente

In **in output** alla *requirements elicitation* si hanno ulteriori sezioni per la bozza di proposta preliminare

evaluation & agreement Come risultato nella fase precedente si effettuando decisioni basate sull'interazione con i vari stakeholders (che spesso richiedono funzionalità conflittuali etc...), per poter valutare e decidere, avendo cambiamenti dei rischi in base a:

- identificazione e risoluzione di conflitti di interesse
- identificazione e risoluzione di rischi legati al sistema proposto
- comparazione e scelta tra le alternative proposte in merito a obiettivi e rischi
- prioritizzazione dei requisti, al fine di:
  - risolvere conflitti
  - definire vincoli di costi e tempi
  - supportare lo sviluppo incrementale

In **in output** a questa fase si hanno le sezioni finali per la bozza di proposta preliminare, dove si documentano gli obiettivi selezionati, i requisiti, le assunzioni e il *rationale*, la logica di fondo delle opzioni selezionate

specification & documentation In questa fase si raccoglie quanto detto nelle prime due fasi per produrre il documento, si hanno quindi:

- definizione precisa di tutte le funzionalità del sistema scelto, tra cui:

  - obiettivi, concetti, proprietà rilevanti del dominio d'interesse, requisti del sistema, requisti del software, assunzioni sull'ambiente e responsabilità
  - motivazioni e logica di fondo delle opzioni scelte
  - probabili evoluzioni del sistema ed eventuali variazioni
- organizzazione di quanto appena citato in una struttura coerente
- documentazione del tutto in un formato comprensibile a tutte le parti, mettendo in allegato:
  - costi
  - piano di lavoro
  - tempi di consegna del risultato

In **output** a questa fase si ha il vero e proprio **Requirements Document (*RD*)**

validation & verification In questa fase si studia l'RD, ovvero si studia la garanzia di qualità dell'RD (in modo equivalente a quanto può essere fatto durante la produzione di un software1, analizzando varie attività:

- **validazione**, ovvero vedendo se quanto contenuto nell'RD è adeguato con quanto si necessita
- **verifica**, controllando se ci sono omissioni o inconsistenze
- **correzione** di eventuali errori e difetti

In **output** a questa fase si ha un RD consolidato.

Ciclicità del processo Come è stato detto queste quattro fasi si ripetono in modo iterativo in modo "a spirale". Questo viene fatto in quanto si possono avere evoluzioni nel processo nonché correzioni di quanto già fatto che si propagano su tutto il documento. Le evoluzioni e le correzioni possono sopraggiungere durante:

- il RE stesso
- lo sviluppo del software
- dopo il deploy del software stesso

Dopo ogni ciclo si è molto più consci del sistema e si può passare al miglioramento del RE con più efficacia.
Questo sistema è molto compatibile con i vari **metodi agili**.