

🔧 COSTRUIRE SOFTWARE 🔧

Processo software

–Un processo di sviluppo software (o processo software) è un insieme di attività che costituiscono un prodotto software. Il prodotto può essere o costruito da zero o essere costituito da software preesistente.

Il processo di sviluppo definisce chi fa cosa, quando e come per arrivare a un certo risultato, si parte da dei requisiti nuovi o modificati, si passa attraverso il ciclo di vita e si arriverà al sistema completo con la sua documentazione.



Le attività tipiche di un processo di sviluppo software sono:

- **Specifica** = comprensione di cosa deve fare il sistema e tutti i suoi vincoli (questa può essere ripetuta anche più volte)
- **Progetto** = definizione astratta del problema (cosa si vuole fare)
- **Realizzazione** = produzione effettiva del sistema
- **Validazione** = controllo che il sistema sia quello richiesto
- **Manutenzione ed evoluzione** = modifica del sistema in base a nuove necessità

Metologia di produzione

–È una rappresentazione semplificata e astratta di un processo di sviluppo, viene utilizzata per mostrare il processo software e si basa su un modello (esistono diversi tipi di modelli raggruppabili in categorie).

Una metodologia include:

- **Artefatti** = tutto ciò che bisogna produrre
- **Flusso di produzione degli artefatti** = mostra come costruire il prodotto finale, in particolare mette in luce le dipendenze
- **Notazioni** = convenzioni usate per produrre gli artefatti
 - Regole a cui bisogna sottostare per evitare problemi (es. bug)
 - Suggerimenti, aiuti, buone pratiche (aspetti informali)

Solo vs. Team

-Il team risulta necessario vista la complessità dei software di oggi e infatti in tutti gli sviluppi di successo la differenza la fanno sempre le persone e le regole (processo di sviluppo) che sono seguite. Lo sviluppo in team è naturalmente molto diverso da quello personale, in particolare in un team ci sono persone con esperienze diverse e ruoli diversi, ad esempio:

- **Architetti** = si occupano di come progettare il prodotto
- **Programmatori** = si occupano di come costruire il prodotto
- **Esperti di dominio** = elaborano a cosa serve il prodotto
- **Esperti di interfaccia** = costruiscono l'interfaccia utente
- **Tester** = controlla la qualità del prodotto
- **Project Manager** = decide come usare le risorse di un prodotto
- **Gestori di configurazioni** = decidono come riusare il software esistente

Programming in the small/large/many

- **Programming in the small** -> un programmatore e un modulo.
Si basa sul processo "programma e debugga" per cui il progetto viene tradotto in un programma definito da passi generici e in seguito vengono eliminati tutti gli errori = ciclo infinito:

code ———> compile ———> debug

Questo tipo di programmazione viene chiamato "cowboy programming" e viene utilizzato solo per progetti piccoli, esplorativi e con scopi didattici, visto che in questo modo non è incoraggiata la documentazione e la manutenzione risulta molto complessa.

- **Programming in the large** -> costruzione di un software decomposto in più moduli, su più versioni e più configurazioni. Questo risulta molto complicato e porta a lavorare in team (o tra più team). Utilizzando questa modalità si possono definire termini come:
 - o **Configurazioni** = insieme di elementi (configuration item) formati da non solo file sorgenti, ma tutti i tipi di documenti e in alcuni casi anche hardware.
 - o **Versione** = lo stato di un configuration item in un istante di tempo.
 - o **Baseline** (Milestone) = una specifica o un prodotto che è stato revisionato e approvato dal management (punto fermo).
 - o **Release** = la promozione di una baseline visibile anche a membri esterni al team.
- **Programming in the many** -> costruzione di grandi sistemi che richiede la cooperazione ed il coordinamento di più sviluppatori nel ciclo di vita di un sw.

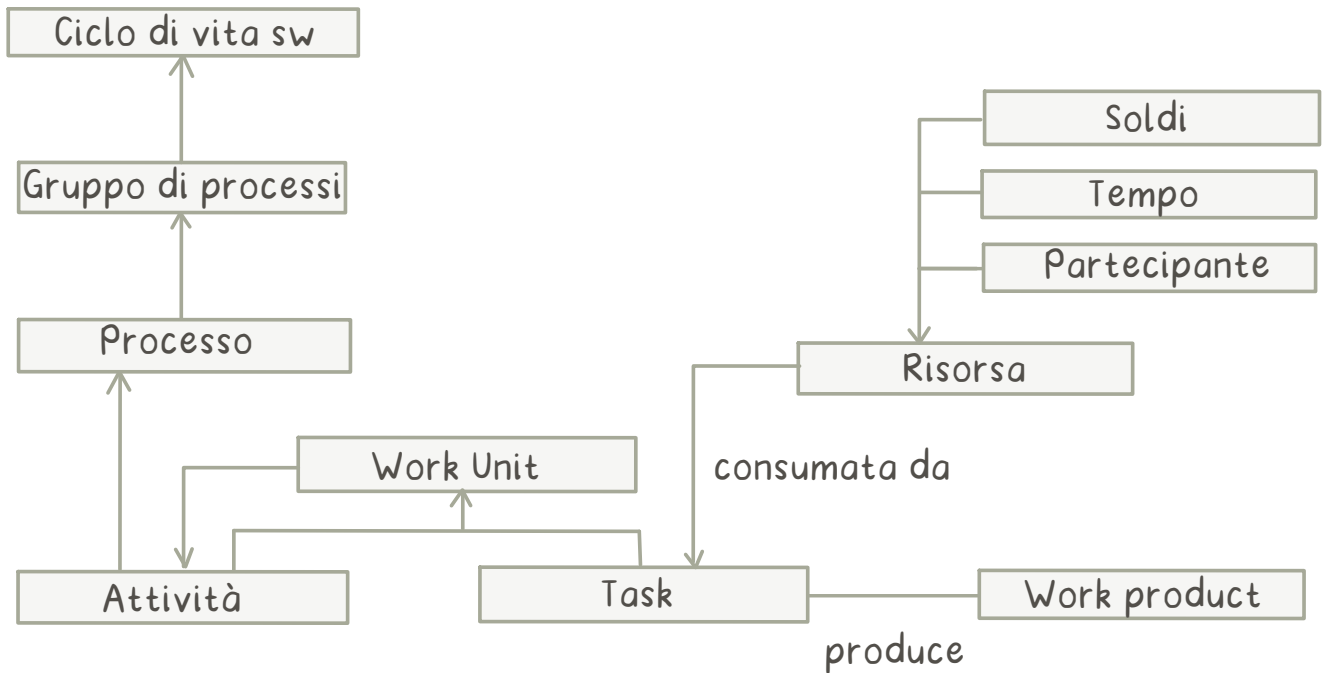
MODELLI DI PROCESSO

Processo software -> attività necessarie per sviluppare un sistema software.

Modelli di processo software -> famiglia di processi.

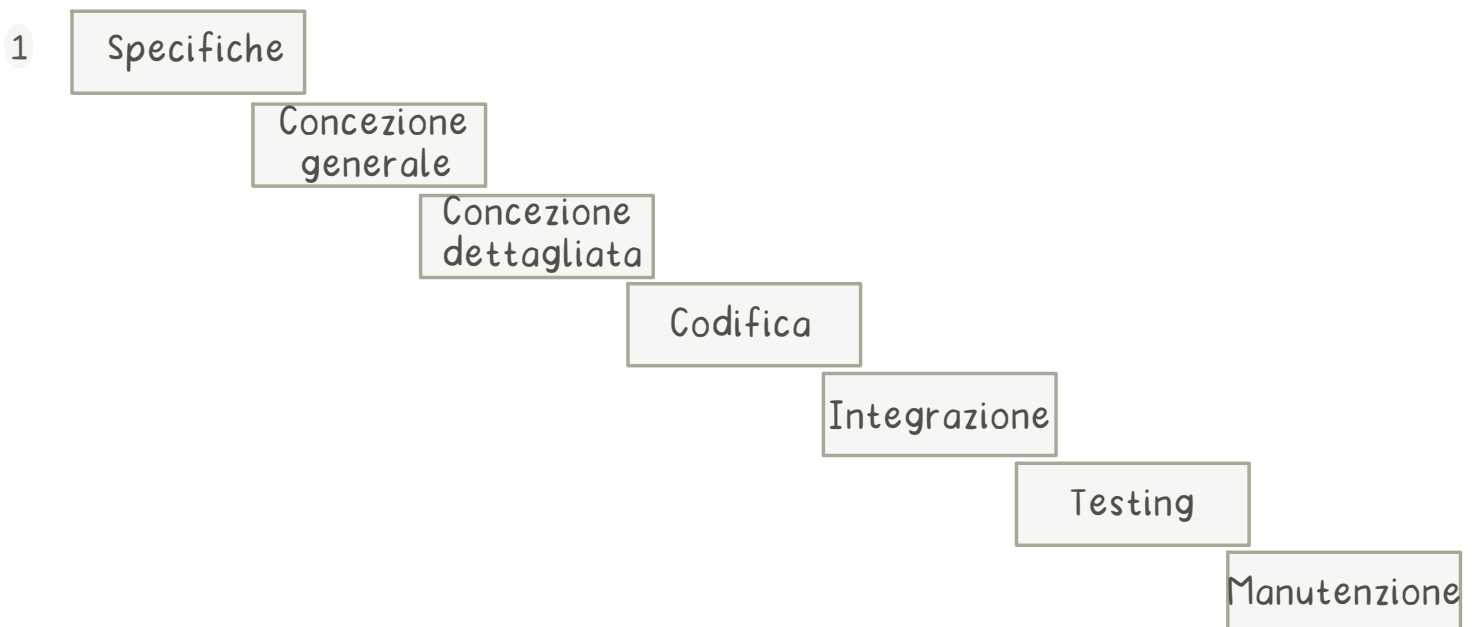
Per descrivere un processo è necessario:

- descrivere/monitorare le attività
- descrivere/assemblare gli strumenti
- descrivere/assegnare i ruoli
- descrivere/controllare gli eventi
- descrivere/validare i documenti
- descrivere/verificare i criteri di qualità



Modelli generali di processo

- 1) **Modelli lineari (Waterfall)** -> specifica e sviluppo sono separati e distinti
- 2) **Modello iterativo (UP)** -> specifica e sviluppo sono ciclici e sovrapposti
- 3) **Modello agile (XP)** -> non pianificato guidato dall'utente e dai test
- 4) **Sviluppo formale (B method)** -> modello matematico trasformato in implementazione



Modelli lineari -> Waterfall, ma esistono diverse varianti

- rigide fasi sequenziali
- fortemente pianificato
- importanza dei documenti (ogni passo forzatamente documentato)
- grande importanza storica

Fasi

Analisi dei requisiti -> comprende utenti e sviluppatori

- il sw deve soddisfare i requisiti
- i requisiti devono soddisfare i bisogni percepiti dell'utente
- i bisogni percepiti riflettono i reali

- Risultato di questa fase: SRS (documento su cosa il sistema deve fare)

Progettazione - si parte dai requisiti per determinare l'architettura del sistema

- diversi approcci e metodologia

- Risultato di questa fase: SDD (documento che identifica tutti i moduli e le loro interfacce)

Codifica -> singoli moduli implementati secondo le loro specifiche (anche testato)

- Risultato di questa fase: codice dei moduli e test dei singoli moduli

Integrazione -> mettiamo tutto assieme e svolgiamo dei test preliminari

- Risultato di questa fase: sistema implementato e STD (doc. test di integrazione)

Test del sistema -> test del sistema globale e validazione

- Risultato di questa fase: V&V (doc. Verifica) e SUD (doc. Utente)

Pro :

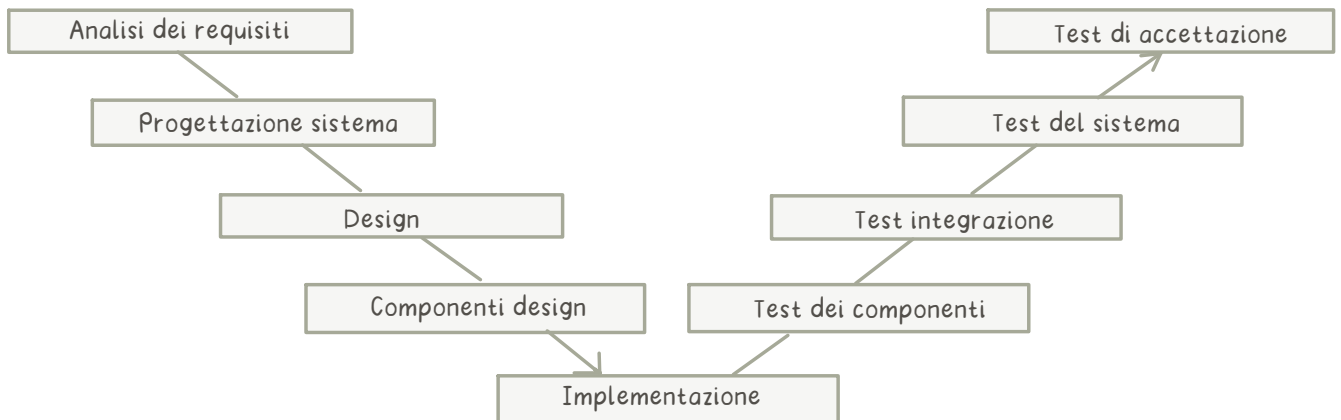
- Molto pianificato (es NASA)
- Produce tanta documentazione
- Orientato agli standard

Contro :

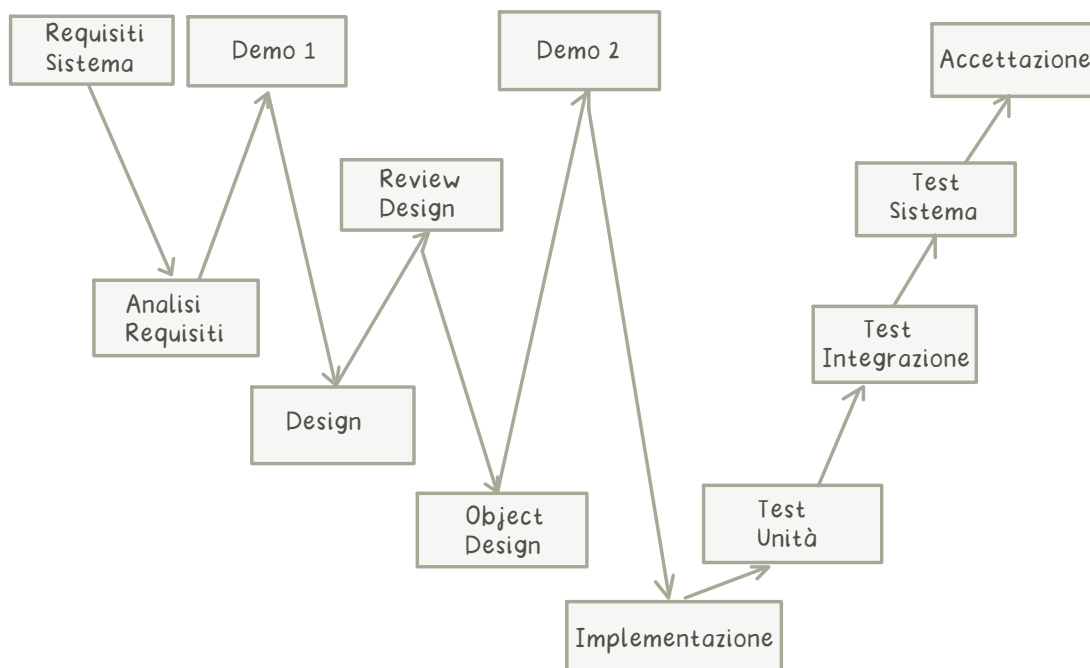
- Molto rigido
- Adatto ad organizzazioni gerarchizzate
- Coinvolge il cliente solo alla fine

V-Model

- Modello lineare (Verification and Validation) → associa fase di test a fase sviluppo
- la fase successiva parte al completamento della precedente



Modello Sharktooth



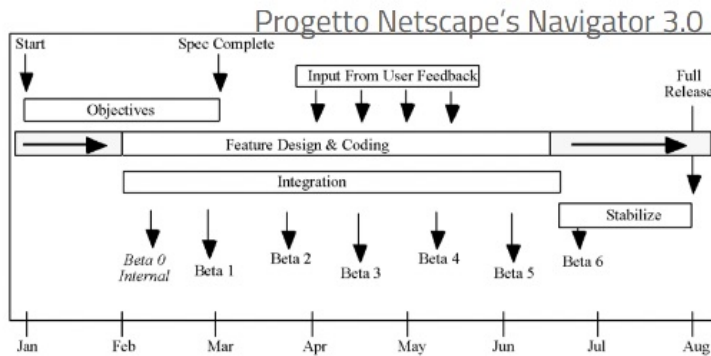
Problema dei processi lineari:

- Modello di sviluppo rigidamente sequenziale
- Il processo non risponde ai cambiamenti di mercato

Modelli iterativi

- I modelli iterativi sono modelli più flessibili, le fasi che abbiamo visto generali (analisi, progettazione ecc...) sono in un qualche modo sovrapposte, quindi riusciamo ad avere alcune versioni più o meno funzionanti che ci danno informazioni e ci permettono in qualche modo di modificare quello che stiamo facendo.

Esempio



Questo è un esempio di come è stato sviluppato netscape navigator.

Fase 1 -> qui le specifiche degli obiettivi sono finite, ma prima di finirle abbiamo già visto due beta (qualche cosa che ci permette di vedere delle idee e di modificare alcune cose)

Fase 2 -> beta 1 beta2 sono distribuite agli utenti che ci daranno un feedback, e questi modificano in qualche modo o la codifica o come sono organizzate le cose.

Fase 3 -> ad un certo punto però arriviamo ad una versione beta che consideriamo stabile. Qui faremo solo degli ultimi aggiustamenti e siamo poi al rilascio.

- Nello specifico il 50% del codice viene sviluppato dopo il rilascio della beta.

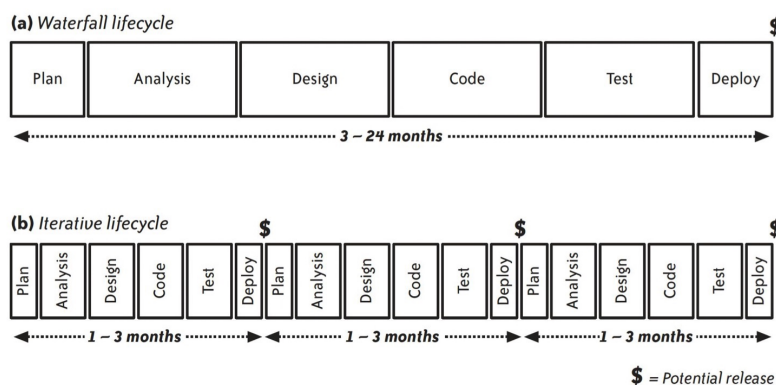
Aspetto temporale

- Il modello Waterfall presenta un tempo ciclico (il software si costruisce in cicli).

Aspetto economico

- La durata del ciclo di vita di un modello Waterfall è di circa 3-24 mesi, e il pagamento avviene solo dopo il rilascio del codice.

Waterfall vs iterativo



Modello a spirale

- Si parte dalla definizione degli obiettivi poi analisi dei rischi e si arriva ad una barriera in cui si dice "Abbiamo dei rischi abbiamo dei piani vale la pena andare avanti?" (Go/No-go) Se andiamo avanti facciamo sviluppo e verifica. Poi facciamo una revisione e ripianifichiamo. Volendo continuiamo la spirale.

Riassumendo:

- 1) **Definizione dell'obiettivo:** ogni round identifica i propri obiettivi
- 2) **Valutazione e riduzione dei rischi:** messa in priorità dei rischi / ogni rischio deve essere affrontato
- 3) **Sviluppo e validazione:** il modello di sviluppo può essere generico / ogni round include sviluppo e validazione
- 4) **Pianificazione:** revisione del progetto e pianificazione del suo futuro

- Un punto importante è che questo sistema cerca di ridurre fortemente i rischi di non riuscita / fallimento.
- Ad ogni Go-No Go si produce un prototipo che permette di valutare i rischi e se è il caso di proseguire oppure no.
- Il modello a spirale è molto più adatto se i requisiti sono INstabili, non è lineare ma pianificato.

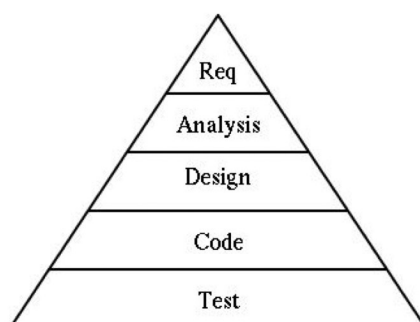
PRO:

- Flessibile, si adatta alle esigenze dell'utente
- Valuta il rischio ad ogni iterazione
- Può sopportare diversi modelli di processo
- Coinvolge il cliente

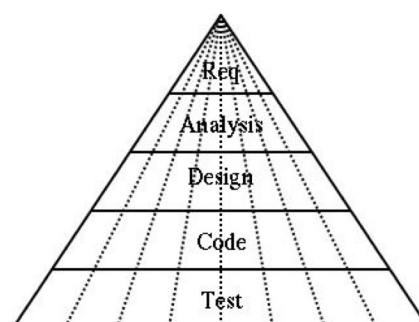
CONTRO:

- Difficile valutare i rischi
- Costoso

Cascata vs iterativi: sforzo



Distribuzione dello sforzo nelle fasi di un processo a cascata: il testing assorbe molto più sforzo delle altre fasi



Segmentazione dello sforzo nelle fasi di un processo iterativo

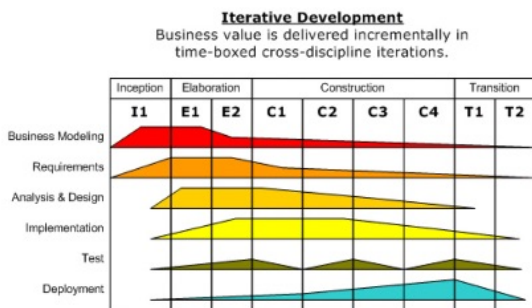
Il modello RUP

-Modello di processo di tipo iterativo e incrementale, diviso in 4 fasi:

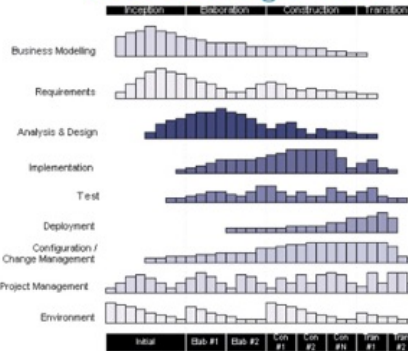
- 1) **Inception** (fattibilità)
- 2) **Elaboration** (progettazione)
- 3) **Construction** (codifica e test)
- 4) **Transition** (deployment)

È articolato su diverse discipline (workflows), inoltre è supportato da strumenti proprietari IBM (esempio: Rational Rose)

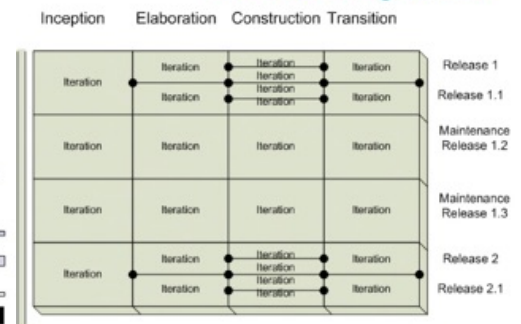
RUP: visione grafica



RUP: visione grafica 2

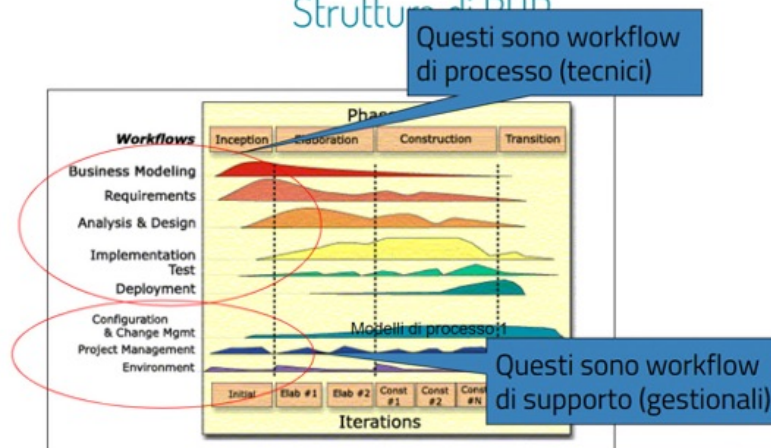


RUP: visione grafica 3



Rispetto agli altri modelli RUP ha un vantaggio, integra alla parte tecnica anche la parte gestionale. È il primo che introduce questi aspetti (soldi, commerciali, umani) Waterfall spirale invece li ignorano.

Struttura di RUP



Le qualità dei processi software

- 1) **Precisione**: il processo descrive ruoli task e artefatti in modo chiaro e comprensibile a chi deve seguirlo
- 2) **Ripetibilità**: il processo può essere duplicato anche da persone diverse, ottenendo lo stesso risultato.
- 3) **Visibilità**: il processo si mostra alle parti interessate
- 4) **Misurabilità**: il processo può essere valutato mediante alcuni indicatori

MODELLI DI PROCESSO AGILI

- Alcuni problemi sono complessi, e richiedono il contributo di molte persone. I requisiti del prodotto probabilmente cambieranno durante lo sviluppo, poiché la soluzione non è chiara all'inizio.

Filosofia agile e volatilità dei requisiti

- Ogni sei mesi o meno, la metà dei requisiti di un prodotto software perde di interesse. La filosofia Agile ha lo scopo di ridurre gli sprechi, e i metodi agili sono molteplici, anche se ci concentreremo su Scrum.

Metodi agili (esempi)

- Extreme Programming (XP)
- Scrum
- Feature-Driven Development (FDD)
- Adaptive Software Process
- Crystal Light Methodologies
- Dynamic Systems Development Method (DSDM)
- Lean Development

Gli aspetti comuni a tutti i metodi agili sono:

- Rilasci frequenti del prodotto (versioni nightly)
- Collaborazione continua del team con il cliente
- Documentazione ridotta
- Valutazione continua di rischi

Manifesto Agile

Per il manifesto Agile prediligiamo:

- 1) Individui e interazioni più che a processi e strumenti
- 2) Software che funziona più che a documentazione completa
- 3) Collaborazione col cliente più che a negoziazione contrattuale
- 4) Reagire al cambiamento più che a seguire un piano

Minimal Viable Product

- Nello sviluppo di un nuovo prodotto, il prodotto minimo funzionante (MVP) è il prodotto con il più alto ritorno sugli investimenti rispetto al rischio.
- È una strategia mirata ad evitare di costruire prodotti che i clienti non vogliono, e quindi a massimizzare le informazioni apprese per ogni euro speso.

Un MVP non è, quindi, un prodotto minimo, ma un processo iterativo di generazione di idee, prototipazione, presentazione, raccolta dati, analisi ed apprendimento.

eXtreme Programming (XP)

XP è una disciplina di sviluppo software basata sui valori di semplicità, comunicazione, feedback e coraggio, porta lo sviluppo ad essere condotto da un team di persone composto da pochi individui, nello stesso locale, e in presenza di un rappresentante di un cliente.

FASI:

- esplorativa
- pianificazione
- iterazione
- produzione

Che continuano a ripetersi fino alla fine dello sviluppo.

Prevede l'utilizzo di user stories, usate al posto dei documenti dettagliati di specifica dei requisiti.

- Vengono scritte dai/con/per i clienti, e contengono le informazioni riguardanti cosa si aspettano i clienti stessi dal sistema.
 - Una storia è descritta da una o due frasi in testo naturale, con la terminologia del cliente (no techno syntax).

Planning game

Le risorse(story points) e i rischi sono valutati dagli sviluppatori, le storie ad alto rischio/priorità vengono affrontate per prime.

Le storie vengono poi catalogate in una pila sul cui fondo sono collocate quelle con priorità più bassa, su questa pila si può:

- aggiungere nuove storie
- cambiare priorità delle storie
- eliminare storie

Per queste operazioni il cliente è necessario perché gli sviluppatori non possono agire per ipotesi ma non devono nemmeno perdere tempo attendendo una risposta del cliente.

Sviluppo test-driven

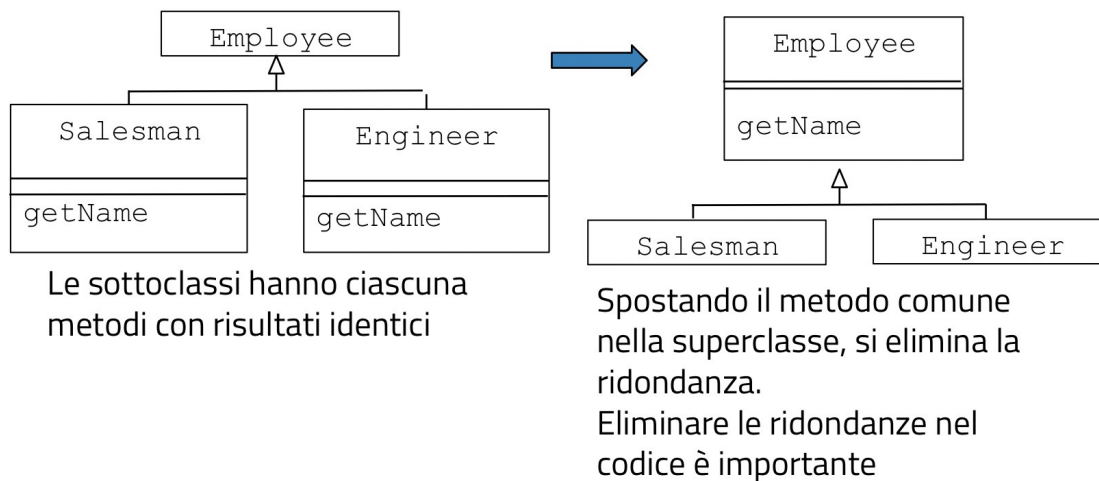
- si sceglie una user story definendo prima i test e poi il codice
- test automatizzati
 - misurano il progresso
 - scritti col cliente
- caratterizzato da piccoli rilasci
 - minimali(microincrementi) e di durata breve e prefissata(timeboxed)
- Il planning game va eseguito ad ogni iterazione di una storia, dopo aver ottenuto feedback dal cliente.

Spesso i progettisti XP creano una metafora per inquadrare il problema e capire come dovrà funzionare il programma, per esempio degli agenti che fanno information retrivial diventano uno sciame d'api che dopo aver trovato il polline lo riportano all'alveare.

Si cerca di progettare in modo semplice:

- facendo la cosa più facile che funzioni
- includendo la documentazione
- rifattorizzando continuamente il codice
 - rimuovendo codice inutile
 - astraendo il possibile (semplificando il codice)
 - utilizzando test automatizzati per non introdurre errori

Refactoring: esempio



Pair programming

- E' una tecnica di programmazione dove due progettisti lavorano sullo stesso computer (paradigma driver-navigatore).
 - uno dei due scrive codice mentre l'altro osserva proponendo soluzioni o cercando difetti nel codice
 - a metà giornata i due si cambiano di ruolo
 - le coppie cambiano giornalmente in modo che tutti prendano dimestichezza col codice

Il pair programming aiuta a prevenire diversi problemi come l'integration hell* e ad tenere alto il morale del team, risolve inoltre conflitti sulla rivendicazione del codice in quanto il codice appartiene a tutti.

|

*Integration hell —> problema derivante dall'integrazione del codice di un singolo, che spesso deve venire adattato per funzionare insieme al codice del team, si risolve integrando ogni piccola modifica seguendo convenzioni di codifica.

PRO:

- le coppie di programmatori sono più produttive (meno errori) e meno stanche (il navigatore non si sforza al 15% invece del 100%)
- i programmatori sono più fiduciosi se lavorano in coppia

CONTRO:

- i test sono lenti e vanno integrati prima di funzionare
- le storie sono complicate ed incline ad errori
- a causa della lentezza dei test il cliente ha poco materiale da osservare

Fattore	Pro-agile	Pro-pianificato
Dimensione	Adatto a team che lavorano su prodotti sw "piccoli". L'uso di conoscenza tacita limita la scalabilità	Adatto a grandi sistemi e team. Costoso da scalare verso i prodotti sw "piccoli"
Criticità	Utile per applicazioni con requisiti instabili (es. Web)	Utile per gestire sistemi critici e con requisiti stabili
Dinamismo	Refactoring	Pianificazione dettagliata e iterata
Personale	Servono esperti dei metodi agili, che sono ancora piuttosto rari	Servono esperti analisti durante la pianificazione
Cultura	Piace a chi preferisce la libertà di fare	Piace a chi preferisce ruoli e procedure ben definiti

ESSENCE

In tutti questi metodi visti in precedenza ogni sviluppatore ha diverse preferenze e idee nella gestione di progetto con una conseguente mancanza di vocabolario comune e differenze sulla gestione con altri sviluppatori.

Pensando solo al metodo Agile ne esisto diverse tipologie tutte diverse come ad esempio lo "Scrum, Kanban, Scrumban"








Per questo nasce il **SEMAT** (Software Engineering Method and Theory) -> è un'iniziativa che ha lo scopo di riformare l'ingegneria del software, dando regole e un linguaggio comune per tutti.

-Così gruppi di lavoro possono valutare il livello qualitativo di un progetto con maggiore facilità e con un migliore dialogo, con il SEMAT si avrebbe un unico metodo simile per tutti e migliorerebbe la comunicazione evitando particolari problemi sia comunicativi che tecnici, di fatto il SEMAT si pone come obiettivo di diventare uno standard.

Nel 2014 i creatori dell' UML creano uno standard di nome ESSENCE che, come per l'UML ovvero il linguaggio di progettazione grafica di un software, l'Essence vuole essere uno schema standard della progettazione dei processi di un progetto software.

Essence prevede -> un Kernel Centrale comune a tutte le pratiche e una componente simbolica affiancata da una descrizione testuale.

-I diversi simboli(immagine sotto) sono associati a funzionalità di questo standard alcune di queste sono:

Name	Symbol
Alpha	
Activity Space	
Activity	
Practice	
Activity Association	
Role / Phase	
Work product	

-Alpha, un simbolo solitamente usato per gli “elementi” che compongono il nostro progetto come “Requisiti”, questi elementi hanno uno stato una lista di attributi che rappresenta la completezza del nostro elemento e il funzionamento dei processi.

-prodotti-> come il codice, i documenti e i report del nostro progetto.

-competenze-> le abilità tecniche richieste dal progetto.

-attività-> processi relativi alle attività del gruppo come organizzazione di riunioni, gestire test, scrivere codice e ruoli.