

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:

- **Specificazione** = compresione 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

Metodologia 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:
 - **Configurazioni** = insieme di elementi (configuration item) formati da non solo file sorgenti, ma tutti i tipi di documenti e in alcuni casi anche hardware.
 - **Versione** = lo stato di un configuration item in un istante di tempo.
 - **Baseline (Milestone)** = una specifica o un prodotto che è stato revisionato e approvato dal management (punto fermo).
 - **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.

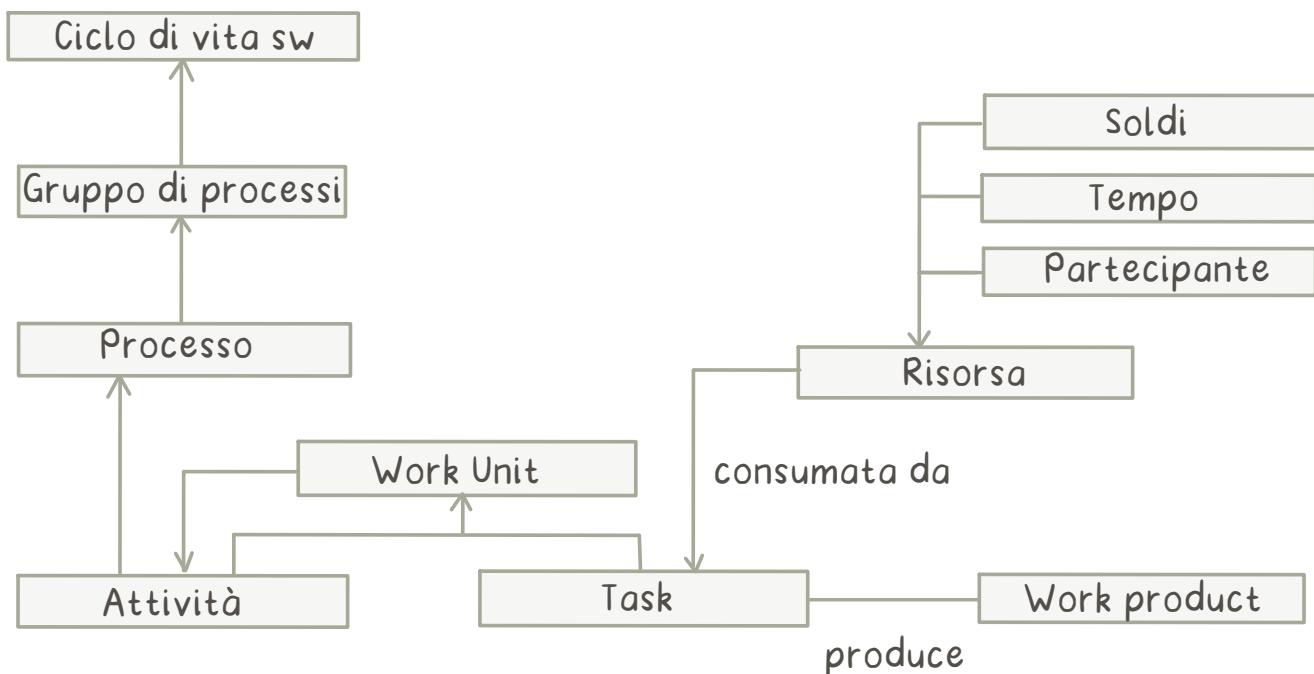
MODelli DI PROCESSO

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

Modelli di processo software -> famiglia di processi.

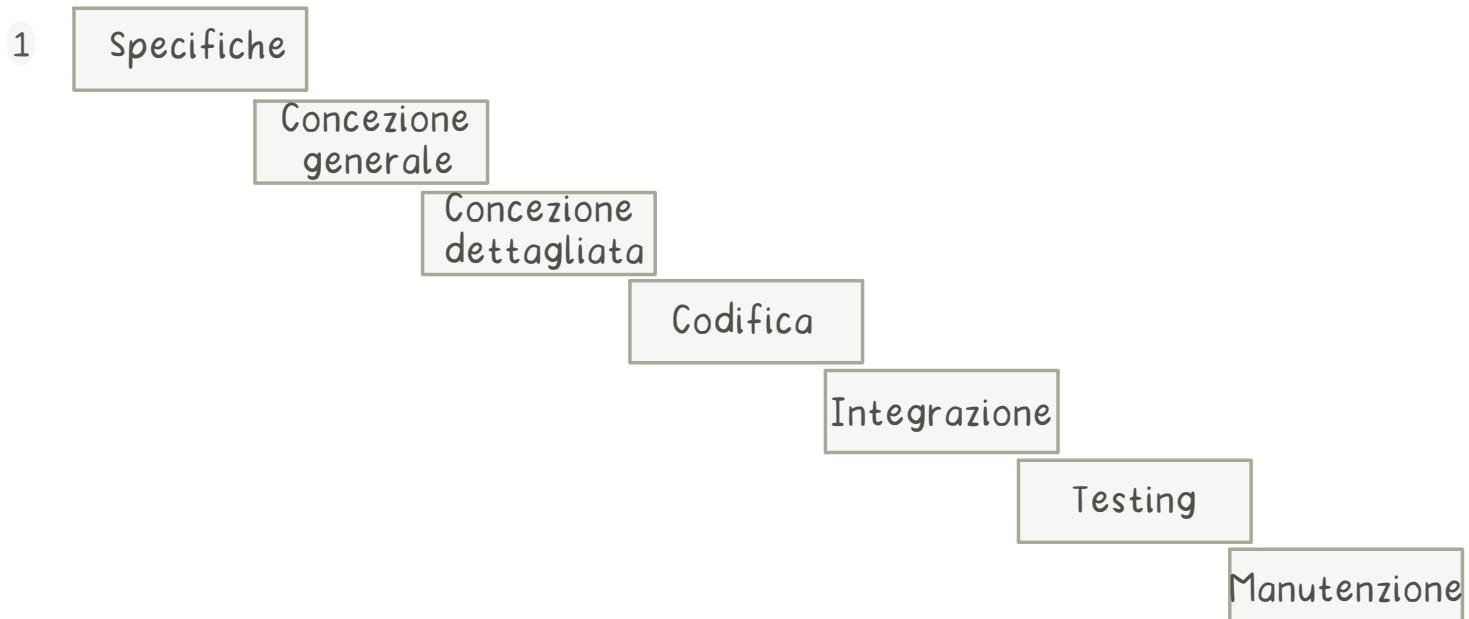
Per descrivere un processo è necessario:

- 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 :

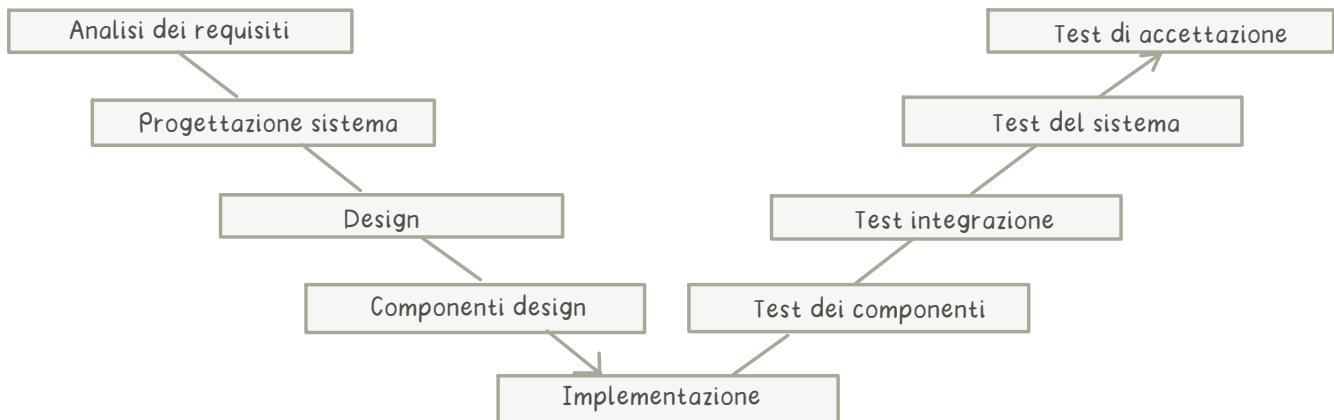
- Molto pianificato (es NASA)
- Produce tanta documentazione
- Orientato agli standard

Contro :

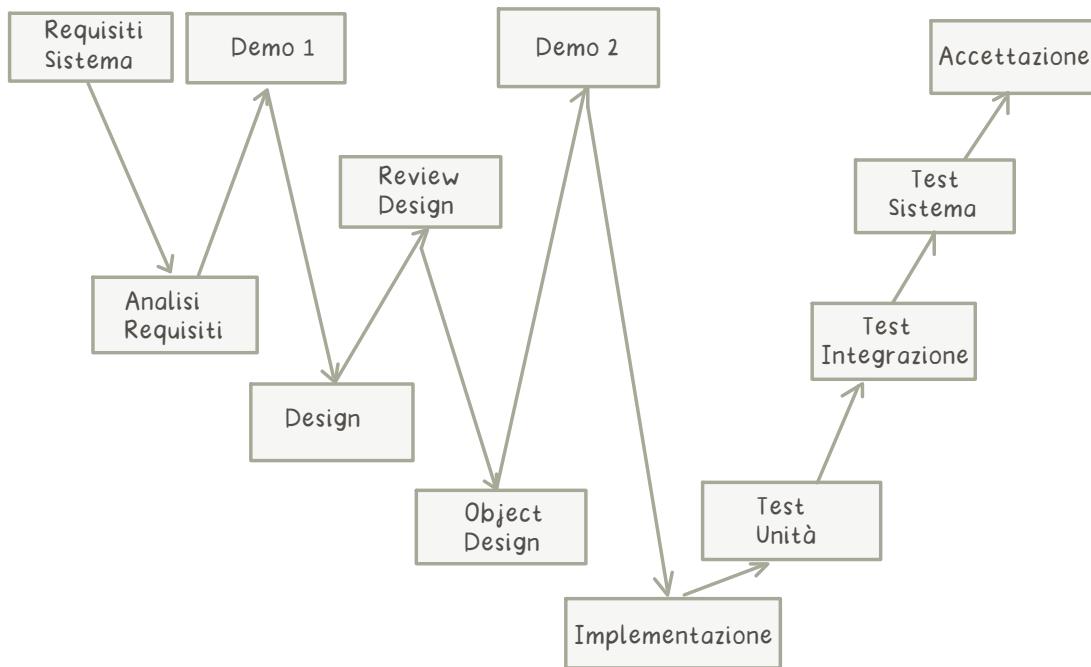
- Molto rigido
- Adatto ad organizzazioni gerarchizzate
- Coinvolge il cliente solo alla fine

V-Model

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



Modello Sharktooth



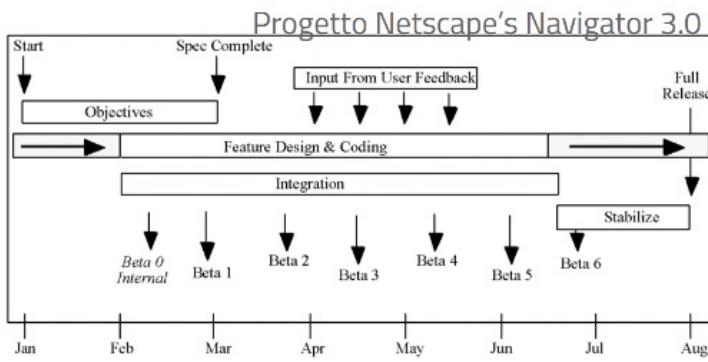
Problema dei processi lineari:

- 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.

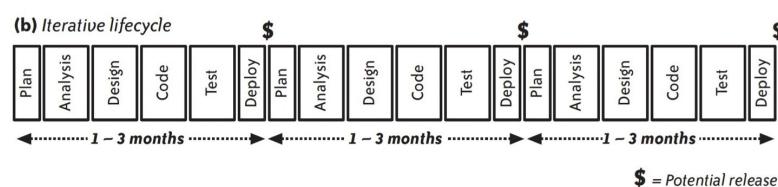
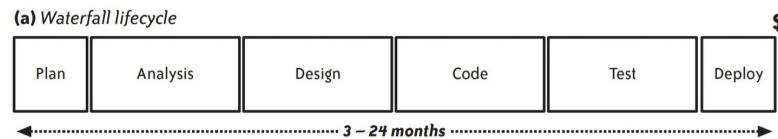
Aspetto temporale

- 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



\$ = Potential release

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.

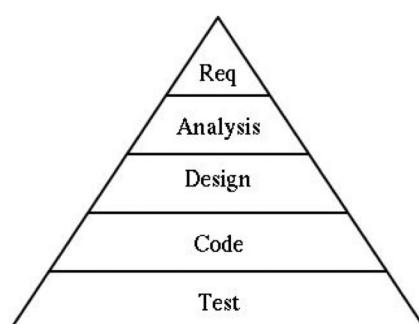
PRO:

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

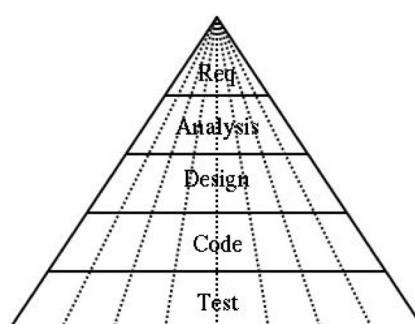
CONTRO:

- 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

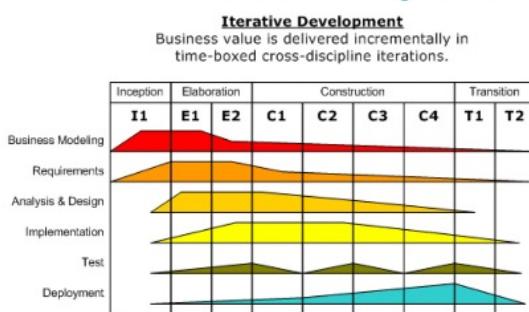
Il modello RUP

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

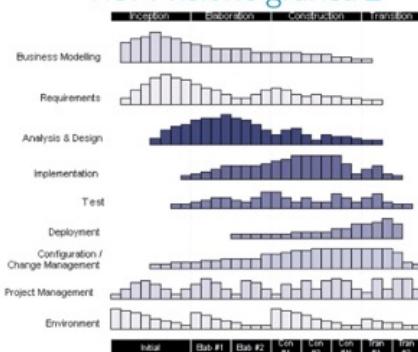
- 1) Inception (fattibilità)
- 2) Elaboration (progettazione)
- 3) Construction (codifica e test)
- 4) Transition (deployment)

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

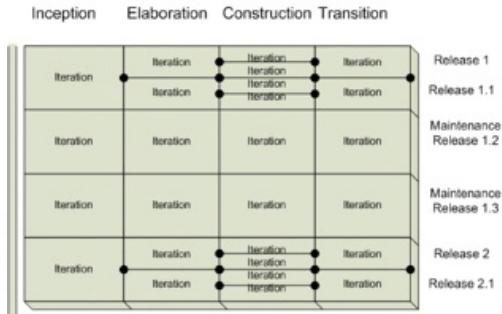
RUP: visione grafica



RUP: visione grafica 2

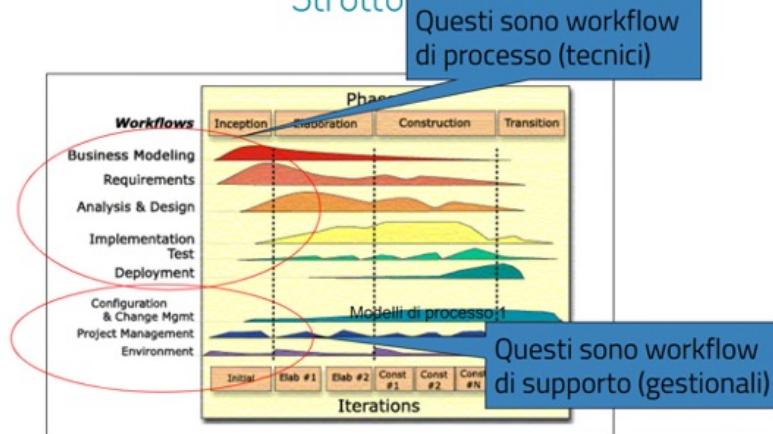


RUP: visione grafica 3



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

MODELLO DI PROCESSO AGILE

-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)

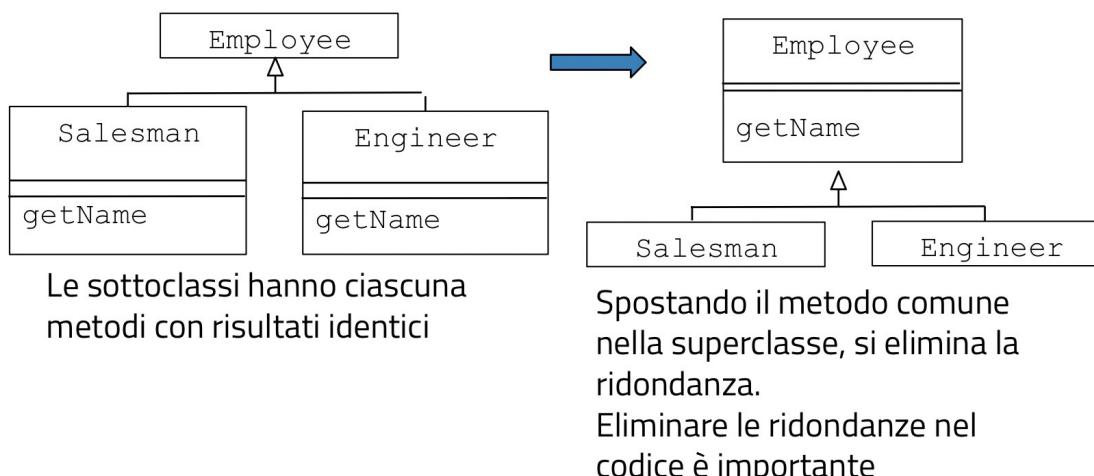
- Il planning game va eseguito ad ogni iterazione di una storia, dopo aver ottenuto feedback dal cliente.

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
 - astraeendo 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 programmatore sono più produttive (meno errori) e meno stanche (il navigatore non si sforza al 15% invece del 100%)
- i programmatore 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 esistono 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.

⌚FASI E TECNICHE DEL PROCESSO DI SVILUPPO⌚

Un processo di sviluppo di un software è diviso in diverse attività:

1) Specifica

- a) Stabilisce requisiti dei committenti e vincoli sul sistema da sviluppare
- b) Si realizza con uno studio di fattibilità, estrazione ed analisi dei requisiti
- c) Specifica ed in fine valida i requisiti

2) **Progettazione**, fase che converte la specifica in un sistema funzionante realizzando prima l'architettura (struttura divisa in vari livelli, che realizza la specifica).

- a) La progettazione ha un approccio sistematico, ottenuto mediante diagrammi di flusso, modelli E/R e modelli UML

3) **Implementazione**, fase strettamente legata alla progettazione che converte l'architettura in codice eseguibile, dopo la conversione segue la fase di debugging, per scoprire eventuali errori di codifica.

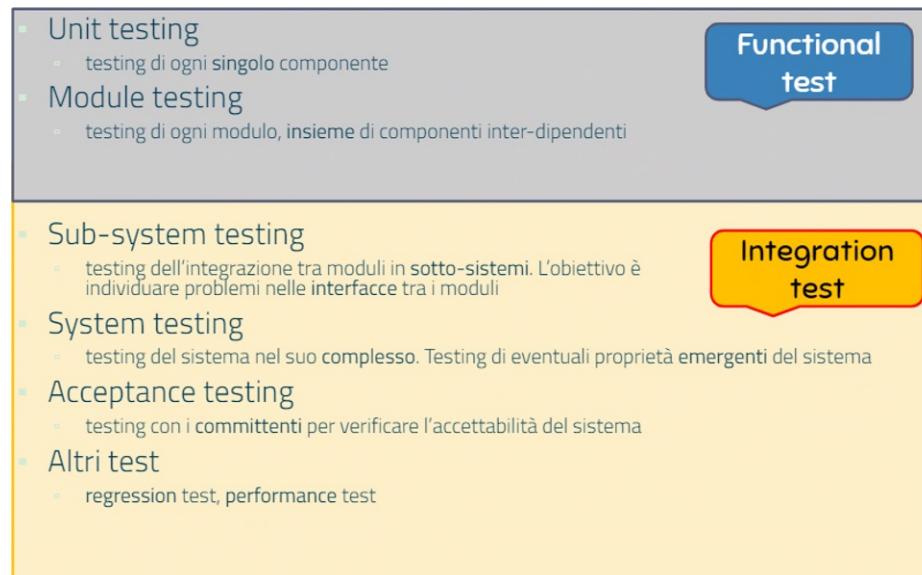
4) **Verifica e validazione**, fase che controlla che il sistema sia conforme alle specifiche (verifica) e che soddisfi i requisiti del cliente/utente (validazione). Vengono inoltre eseguiti altri due step aggiuntivi.

a) Revisione, in cui vengono evidenziati eventuali difetti del prodotto (logici, quelli di codifica li eliminiamo col debugging)

b) Testing, viene eseguito eseguendo il nostro sistema su diversi test cases, verificando il suo funzionamento in questi casi possiamo pensare che il suo comportamento sia "buono" in ogni istanza di utilizzo.

Non bisogna confondere il testing con il debugging, il primo ha come scopo quello di trovare il bug, il secondo deve trovare la causa del bug.

Il testing è importante e complesso, e per questo viene diviso in:



Le tipologie di testing più utilizzate sono:

1. **Black box testing**, test che ignora il programma, concentrandosi solo sui risultati
2. **White box testing**, usa la struttura interna del programma per costruire il test
3. **Grey box testing**, combina black e white box testing
4. **Monkey testing**, si basa sul principio della scimmia instancabile, invia input casuali al programma, riuscendo così a testare una grande quantità di casi
5. **Exploratory testing**, dipende dall'abilità del tester che utilizza la sua conoscenza del codice e grazie alla sua esperienza costruisce test ad hoc.

Il testing si divide ancora in:

- **Testing in the small** -> testa una piccola parte di codice verificandone il suo funzionamento in relazione ad input significativi, ottenendo così una verifica della copertura delle istruzioni. L'obiettivo è quello di testare qualsiasi istruzione e condizione presente nel codice (Es. ogni if e relative diramazioni), per fare ciò dobbiamo conoscere il codice di un programma, utilizzeremo quindi il white box testing.

- **Testing in the large** -> è eseguito su tutto il sistema facendo diventare impossibile applicare il white box testing (troppo dispendioso e difficile), utilizziamo il black o gray testing accettando in input test selezionati in base alle specifiche.

In entrambi i casi, il testing manuale è in disuso, ogni programmatore deve essere in grado di utilizzare le tecniche di testing automatizzato.

L'ultima fase del testing è l'**ispezione del software**, che controlla la qualità del codice e ricerca specifici errori, può essere eseguita in due modi:

- **Code walk-through**, viene effettuato simulando il comportamento del programma ed utilizzando la documentazione del codice, si cercano di trovare i problemi, non di risolverli.
- **Code inspection**, ha come scopo di ricercare classi specifiche di errori (loop infiniti, letture illegali di aree di memoria, rilascio improprio di memoria).

5) Manutenzione ed evoluzione

I requisiti ed i desideri

Il progettista ha il compito di costruire i requisiti del progetto, tenendo però sempre in conto i desideri e i bisogni del cliente. I requisiti (specifiche) sono il frutto di un accordo tra lo sviluppatore ed il committente e specificano quello che il programma deve riuscire ad ottenere, ogni fase di sviluppo ha bisogno di specifiche, più o meno dettagliate.

La creazione di una specifica si compone in:

- Progettista e committente si accordano su **ciò che il software deve fare** (inserito nel SRS o backlog)
- Progettista e programmatore si accordano sull'**architettura dei moduli e dei servizi da implementare**
- I programmatore si accordano sulle **interfacce dei moduli da costruire**

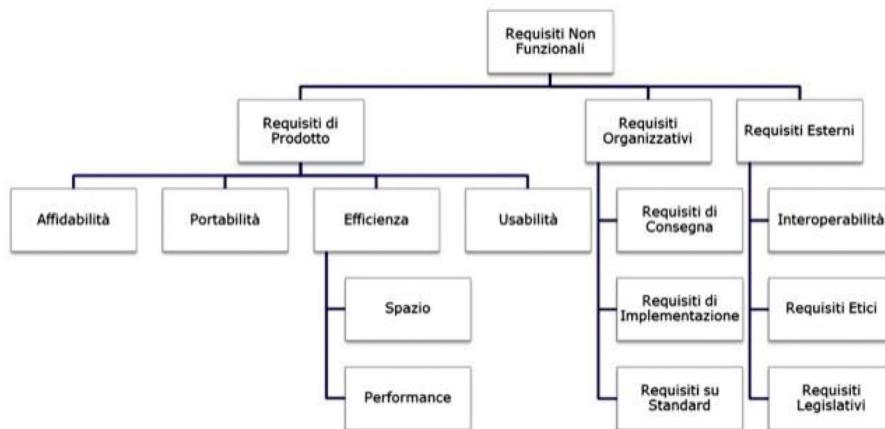
La specifica dice soltanto cosa il sistema deve fare, il come viene fatto qualcosa viene lasciato alle fasi successive, precisamente alla fase di implementazione.

Requisiti

- **Requisito** -> descrizione delle caratteristiche richieste al sistema (essenziale). Possiamo dividere i requisiti in tre gruppi:
 - **Requisiti FUNZIONALI** -> come il sistema deve reagire agli input e come deve comportarsi.
 - **Requisiti NON-FUNZIONALI** -> caratteristiche dei servizi offerti dal sistema (reattività, affidabilità..).
 - **Requisiti di DOMINIO** -> derivano dal dominio applicativo.

NB. Alcuni requisiti possono essere non-funzionali in alcuni domini e funzionali in altri.

Tassonomia di Sommerville



Modello FURPS

F : funzionalità
U : usabilità
R : affidabilità
P : performance
S : supportabilità

Le specifiche sono prima di tutto un elenco di requisiti discussi tra sviluppatore e committente, si passa poi alla descrizione dell'interfaccia tra macchina e ambiente controllato.

Le specifiche possono anche variare durante la manutenzione di un prodotto.

Manutenzione di un prodotto

Manutenzione CORRETTIVA : cambia l'implementazione ma non le specifiche (problema se l'errore si trova nelle specifiche)

Manutenzione ADATTIVA : modifiche ai requisiti (problema incoerenze sul codice)

Manutenzione PERFETTIVA : in requisiti funzionali non cambiano

Come devono essere le specifiche ?

- Chiare -> non ambigue
- Consistenti -> non devono contenere contraddizioni
- Complete -> completezza interna : definire tutti i termini e concetti di cui fa uso
completezza esterna : tutti i requisiti devono essere specificati
- Incrementalità -> le specifiche vanno raffinate in modo incrementale (vari prototipi)

Come si verificano le specifiche ?

- Si osserva il comportamento dinamico del sistema
- Si analizzano le proprietà del sistema

- Specifiche operazionali
 - *Diagrammi di flusso di dati*
 - Diagrammi UML
 - *Macchine a stati finiti*
 - *Reti di Petri*
- Specifiche descrittive
 - *Diagrammi E/R*
 - Class-Responsibility Card
 - User Story

Come si scoprono i requisiti?

- Raccogliendo informazioni :

- Stakeholder analysis -> analisi che identifica tutte le persone che potrebbero essere influenzate dal sistema
Pro: sono considerati tutti i punti di vista, si avrà un elenco con priorità
Contro: richiede molto tempo e genera troppi dati
- Brainstorming -> permette di raccogliere molte idee da un gruppo di persone
Pro: genera molte idee in poco tempo
Contro: molte idee vanno scartate, non sempre produttivo
- Intervista -> permette di raccogliere molte idee da un gruppo di persone

SRS - SOFTWARE REQUIREMENT SPECIFICATION

E' un documento che riporta le specifiche dei requisiti del software definita dallo standard IEEE 830

- utilizzato come parte di contrattazione con il cliente

L'SRS è influenzato da diverse figure:

- **il Committente**: colui che paga il prodotto e solitamente ne decide i requisiti
- **il Fornitore**: ovvero lo sviluppatore
- **l'Utente**: colui che usa il prodotto

In quanto documento l'SRS deve:

- descrivere le specifiche di requisiti del software
- fungere da contratto ovvero documento legale, che viene stilato e che mostra le specifiche

L'SRS deve essere:

- **Corretto**: deve essere verificabile e verificata la correttezza dell' SRS e la veridicità
- **Non ambiguo**: esente da ambiguità di comprensione su termini
- **Completo**: avere tutte le informazioni necessarie dell' SRS
- **Consistente**: senza conflitti tra i vari requisiti
- **Ordinato per priorità** dei requisiti: ogni requisito deve possedere un identificatore che stabilisce la priorità, i criteri di ordinamento possono essere diversi

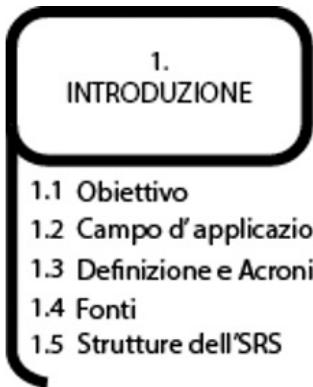
Metodo Moscow

MUST - essenziale (fondamentali)	Mo	Must-haves
SHOULD - condizionale (migliorano la qualità del software, ma non compromettono il suo utilizzo)	S	Should-haves
COULD - opzionale (introducono il valore aggiunto al software)	Co	Could-haves
WOULD - gradito (ma può essere tralasciato)	W	Won't- and Would-haves

- **Verificabile**: stabiliamo in modo oggettivo se il requisito viene soddisfatto o meno,
- **Modificabile**: l'SRS è di per sé una cosa stabile ma ha delle parti che prevedono la modificabilità dovuta a errori di progettazioni e aggiornamento
- **Tracciabile**: ogni requisito deve essere tracciabile all'interno del codice del software

Per un'ideale riuscita sarebbe ottimo se il committente e il fornitore scrivessero insieme i requisiti

Abbiamo una suddivisione in 3 grandi blocchi



1. Introduzione

Partiamo da un esempio:

Abbiamo un Software chiamato "Ambulatorio" che fornisce una serie di servizi relativi alla gestione della cartella clinica di ciascun paziente di un effettivo ambulatorio, vengono forniti 3 servizi:

1. Prima visita(PV)
2. Day Hospital(DH)
3. Visita di controllo(C)

Per ogni paziente abbiamo cartella clinica con tutte le informazioni mediche relative quindi tutte le varie visite, le prenotazioni, diagnosi ecc.

L'introduzione è la parte più discorsiva infatti raccoglie la natura del nostro software come la descrizione e gli obiettivi da soddisfare, quindi deve avere una sua definizione di obiettivo, in questo caso occuparsi dei 3 servizi, deve inoltre avere un dizionario utile alla raccolta di tutte le documentazioni e nomenclature e termini abbreviazioni che verranno nominati all'interno.

2. Descrizione Generale

Il nostro esempio prosegue, infatti il software Ambulatorio nasce dalla necessità di sostituire il sistema "AMB31" che era sviluppato per un sistema DOS, "Ambulatorio" è pensato per personale medico che ha conoscenza del settore medico ma meno in campo informatico inoltre deve essere utilizzato su Windows 95 o 98 con possibile portabilità per sistemi Mac.

Questa seconda parte della descrizione serve alla descrizione di particolari caratteristiche:

- il fruitore di questo sistema non è informatico
- ci sono pregresse conoscenze basate su un sistema precedente(AMB31)

Riassumendo abbiamo dei vincoli software sia di interfaccia nell'usabilità che hardware nella possibilità di installazione su alcune tipologie di macchina per esempio se sappiamo che deve essere installato su un PC del 1995

3. Specifica dei Requisiti

Il nostro software "Ambulatorio" deve avere un tempo di avvio inferiore a 10 s con un tempo di stampa di 10/15 secondi, utilizzare il database SQLite e stare ai vincoli di progetto, l'accesso al sistema è tramite password.

L'interfaccia Esterna deve prevedere un'interfaccia amichevole con una alta usabilità e intuitività, i requisiti funzionali descritti tramite schede, dove viene descritto il processo la sequenza di azioni input effettuate e la risposta sull'output con le eventuali risposte di errore,

INPUT [chi esegue, quale esegue] → descrizione[sequenza di operazioni] →
OUTPUT[operazione eseguita]

Requisiti Non Funzionali, comprendono:

- tutti i Requisiti Prestazionali come i terminali e dispositivi supportati
- i Database che può cambiare in base alla tipologia che si vuole utilizzare(SQLite nel nostro esempio)
- i Vincoli Generali del progetto quindi limitazioni hardware o a standard già esistenti
- Dobbiamo specificare il Grado di affidabilità del software al momento del rilascio
- L'accessibilità al sistema comprende accessibilità al sistema da tutti gli utenti
- Sicurezza controllo e gestione tramite sistemi di controllo password
- Manutenibilità rendere il software mantenibile tramite modularità
- Portabilità software facilmente portatile dove il DBMS è indipendente dal sistema operativo

PROGETTAZIONE AGILE VS PROGETTAZIONE TRADIZIONALE

L'SRS ha un orientamento estremamente tecnico dettagliato ed è legato molto agli aspetti legali contrattuali (questo poiché fornisce la base del contratto che verrà firmato). Quindi per sua natura tenderà ad essere molto grande, molto approfondito e richiede una grande preparazione (ci si possono mettere mesi per scrivere un SRS). Teoricamente l'SRS dovrebbe evolvere con il tempo ma solitamente non viene quasi mai modificato.

L'SRS fornisce delle basi di lavoro per il progetto.

Non è ASSOLUTAMENTE AGILE -> poiché facciamo una grande pianificazione e poi non la tocchiamo più.

I progetti agili hanno un approccio completamente diverso al problema dei requisiti. Non si escludendo gli approcci classici, la progettazione dei requisiti è incentrata su due artefatti: user story & il product backlog

Che cos'è una User Story?

È una descrizione di una necessità di un utente, descritta in modo estremamente sintetico e di solito segue questo modello:

As a **<user type>**
I want **<to do something>**
so that **<I get some value>**

Punti fondamentali:

1. **La tipologia di utente**: che tipo di utente ha interesse a sviluppare questa storia
2. **Le sue necessità**: vuole fare qualche cosa
3. **Perché?**: in modo da ottenere qualche vantaggio

*Come **utente della biblioteca** (ruolo)*

*Voglio **consultare il catalogo** (scopo o funzione)*

*Per **prendere a prestito un volume** (valore ottenuto)*

Le User Story dovrebbero essere arricchite da alcune condizioni/casi specifici

Es: cosa succede al nostro utente se vuole consultare il catalogo per prendere un volume e questo volume non c'è?

Consideriamo:

- l'iscritto ha già preso diversi libri che ha a casa, gli facciamo prendere il libro?
- Il volume è stato ordinato ma non è arrivato, gli facciamo un messaggio/sollecito? Insomma ci sono queste considerazioni, che possono essere utilizzate in vario modo e possono essere usate come varianti delle User Story e possono essere ciascuna spezzata in una versione indipendente.

Task → siccome le user story sono poco tecniche le dividiamo in task più piccole e tecniche.

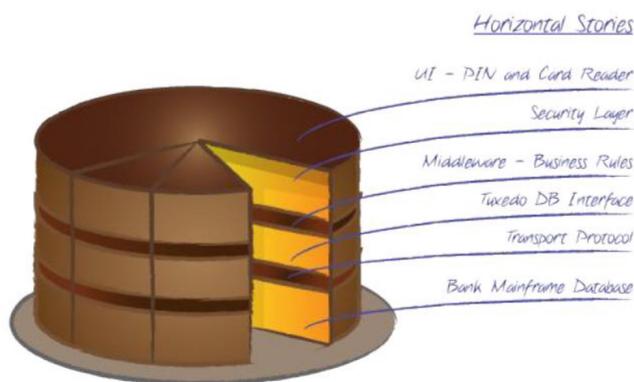
User Story con task esempio:

Come utente del portale voglio iscrivermi inserendo il mio CV e lo stipendio desiderato per ricevere informazioni su ogni offerta di lavoro che soddisfi la mia richiesta

Quindi devo:

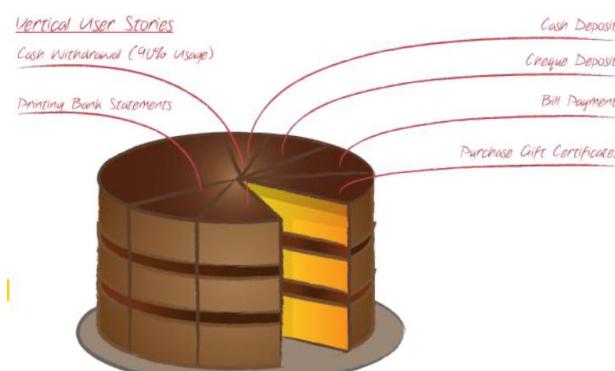
- Realizzare autenticazione con OAuth, facebook o altre cose per autenticarmi
- Realizzare upload del CV e delle richieste
- Realizzare un Web Crawler che cerca le cose in giro
- Sintetizzare report
- Inviare segnalazione all'utente

Molto spesso però gli sviluppatori preferiscono ragionare in backlog per ragioni di efficienza:



Non c'è nulla di sbagliato in questo esempio SE NON
LA PROSPETTIVA, non permette di vedere le US

Dobbiamo fare le partizioni verticali che permettono di confezionare le singole funzionalità ed avere un feedback continuo dal cliente



La prima fetta è il caso principale

Come usiamo il bancomat?

- La maggior parte delle volte lo usiamo per prendere dei soldi. Quindi diventa essenziale che sia la parte che sviluppiamo per prima.

Qual'è il vantaggio?

Nel momento in cui io realizzo la mia prima storia (e devo usare tutto, devo avere il database, la banca, la sicurezza ecc...) Questa parte la posso far partire. Volendo aprire un bancomat che fa solo questo. Non sarà il massimo avere un bancomat che mi fa solo ritirare, ma almeno il 90% dell'uso del bancomat è assicurato, poi andro ad aggiungere in futuro le funzionalità extra, man mano che vengono realizzate.

Quindi qual è il ruolo delle US nel processo di sviluppo agile?

Il product owner costruisce le US, che vengono usate sia dallo sviluppatore, sia dal tester che produce il programma di test.

- Utente: come descrivo i miei desideri?
- Stakeholder(chi paga la produzione del prodotto): come posso far sì che il prodotto abbia successo?
- PM: come traccio e pianifico questo compito?
- BusinessAnalyst: quali sono i dettagli di questa feature?
- UX(interfaccia utente): quali sono i bisogni dell'utente?
- Developer: quali task devo eseguire oggi?
- QA: come posso validare questo task completato?

Come per i libri, ci sono storie buone e storie cattive. Per ottenere obiettivi sensati (S.M.A.R.T.) bisogna investire (I.N.V.E.S.T.) nelle user story

S.M.A.R.T

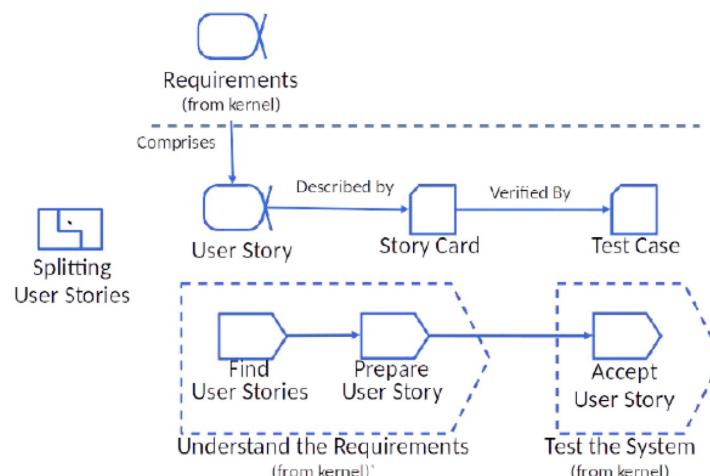
- Specifici
- Misurabili
- Accettati
- Realistici
- Temporizzati

- Independent (così si può lavorare in parallelo, serve un lavoro in contemporaneo!)
- Negotiable (contrario di srs, possono le us cambiare durante il progetto (sicuramente cambiano), spesso si negozia un compromesso se cambia.)
- Valuable (la us deve dare qualcosa al cliente, obiettivo principale evitare storie tecniche, es passiamo al framework 3d rispetto al 2d. perchè? l'utente ne trae vantaggio? se non si riesce ad identificare il valore della storia probabilmente è sbagliata)
- Estimable (importante per quanto lavoro si può fare, se non si riesce a stimare ci sono problemi. le us sono tarate su un periodo di sviluppo)
- Sized right (se troppo grande deve essere lavorata. e spezzata.)
- Testable (deve essere testabile)

User stories difettose: esempi

ID	Description	Issues
US ₁	As a User, I'm able to click a particular location from the map and thereby perform a search of landmarks associated with that latitude longitude combination	Not atomic: two stories in one
US ₂	As a care professional I want to see the registered hours of this week (split into products and activities). See: Mockup from Alice NOTE: - First create the overview screen - Then add validations	Not minimal, due to additional note about the mockup
US ₃	Add static pages controller to application and define static pages	Missing role
US ₄	As a User, I want to open the interactive map, so that I can see the location of landmarks	Conceptual issue: the end is in fact a reference to another story
US ₅	As a User, I'm able to edit any landmark	Conflict: US ₅ refers to any landmark, while US ₆ only to those that user has added
US ₆	As a User, I'm able to delete a landmark which I added	
US ₇	As a care professional I want to save a reimbursement. - Add save button on top right (never greyed out)	Hints at the solution
US ₈	As a User, I am able to edit the content that I added to a person's profile page	Unclear: what is content here?
US ₉	As an Administrator, I am able to view content that needs to be reviewed	The type of content is not specified
US ₁₀	Server configuration 	In addition to being syntactically incorrect, this is not even a full sentence
US ₁₁	As an Administrator, I am able to add a new person to the database followed by	Viewing relies on first adding a person to the database
US ₁₂	As a Visitor, I am able to view a person's profile	
US ₁₃	As a care professional I want to see my route list for next/future days, so that I can prepare myself (for example I can see at what time I should start traveling)	Difficult to estimate because it is unclear what seeing my route list implies
US ₁₄	As an Administrator, I receive an email notification when a new user is registered	Deviates from the template, no "wish" in the means
EP _A US ₁₅	As a Visitor, I'm able to see a list of news items, so that I can stay up to date on news As a Visitor, I'm able to see a list of news items, so that I can stay up to date on news	The same requirement is repeated both in epic EP _A , and in a user story US ₁₄

Esempio di essence che descrive come fa le storie



Problema della stima

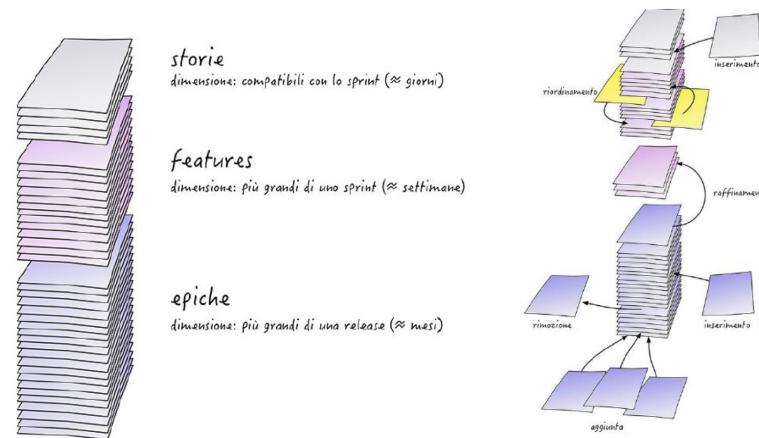
Il dominio del tempo è la cosa che più sfugge ed è inoltre una delle cose più importanti da stimare.

Tutte le US dovrebbero essere stimate, la stima è indicata in "Story Points". Sono un'unità di misura astratta e relativa, utile per confrontare le storie tra loro, inoltre la stima delle user story incoraggia il coinvolgimento di tutti in una discussione costruttiva.

Secondo la metodologia agile ciò è sbagliato poiché le persone che dovrebbero stimare il tempo di realizzazione di una storia, quindi del progetto, sono quelli del team di sviluppo.

L'attività per fare questa discussione costruttiva si chiama Planning Poker / Planning Game, questa mette insieme diverse esperienze e punti di vista.

Backlog (ovvero l'insieme delle User Story). Queste storie sono priorizzate, a seconda delle dimensioni, dei rischi, delle necessità. Una selezione di storie da realizzare costituisce l'obiettivo di un'interazione (detto anche sprint)



Backlog in essence



PRO

- Cicli di feedback continui
- Involvemente cliente, visione comune, conversazione
- Prioritizzazione
- Massimizzazione usabilità
- Facilita la stima dei tempi

CONTRO

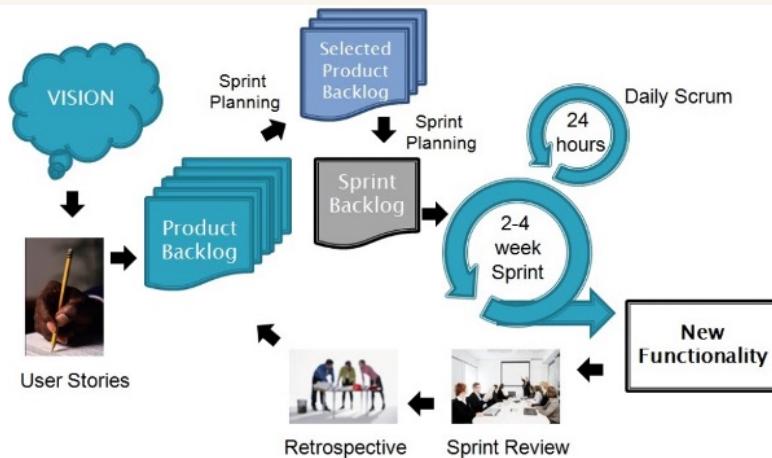
- Non adatto per aspetti puramente tecnici (esempio bug)
- Problemi legali
- Perdita di vista dell'aspetto globale (design povero)
- Difficili da scrivere

Tools per progetti agili

- Post-it, whiteboard, penna
- Trello
- WeKan
- Pivotal Tracker
- Taiga
- Atlassian Jira, Microsoft Teams e VersionOne (costosi ma completi per lo sviluppo)

SCRUM

Scrum è un framework agile per la gestione del ciclo di sviluppo del software, iterativo e incrementale e al suo interno è possibile utilizzare vari processi e tecniche (viene infatti spesso utilizzato con XP).



Perché usare Scrum?

Scrum permette al cliente di ispezionare ogni 1-4 settimane le versioni funzionanti del software. Il cliente definisce le funzioni da realizzare e le loro priorità -> team decide ogni giorno il modo migliore per produrre tali funzionalità. Ogni 1-4 settimane nasce una nuova versione.

Caratteristiche

- Centralità team -> auto organizzante che si basa sulla divisione in ruoli
- Sviluppo guidato da storie e test -> prodotto cresce in "sprint"
- Sprint -> unità di base di sviluppo in Scrum ed è di durata fissa = 1-4 settimane
- Artefatti -> es. Product Backlog (in cui sono contenuti i requisiti), Sprint Backlog (quale requisito verrà relizzato)
- Eventi -> veri e propri meeting es. Sprint Planning, Daily Scrum, Sprint Review

Team e ruoli

Un team è formato da minimo 5 persone e massimo 7, tutti responsabili della riuscita del progetto. Si possono dividere in tre ruoli specifici:

1. **Product Owner (PO)** = definisce le funzionalità, stabilisce le priorità, fissa date di rilascio, feedback, gestisce gli stakeholders, accetta o rifiuta i risultati.
2. **Scrum Master (SM)** = elimina ostacoli, difende il team, controlla il processo e gestisce management.
3. **Membri del team (TM)** = definiscono i task, stimano lo sforzo, permettono evoluzione processo e assicurano qualità.

Scrum identifica tre pilastri

- **Trasparenza**: gli aspetti significativi del processo sono visibili ai responsabili del risultato finale
- **Ispezione**: ispezione frequente con lo scopo di rilevare deviazioni indesiderate
- **Adattamento**: se l'ispezione rileva che un aspetto è al di fuori dei limiti, allora bisogna adattare il processo o il materiale processato

Valori

- **Focus**:
 - Scrum Master: rimozione degli impedimenti e applicazione del modello Scrum all'interno del team
 - Project Owner: massimo valore per il prodotto
 - Team: sviluppo della soluzione migliore
- **Coraggio**:
 - Scrum Master: coraggio nel proteggere e guidare il team
 - Project Owner: deve fidarsi delle attività svolte dallo Scrum Master
 - Team: Realizzazione dei Work Item, superando i propri limiti di adattamento
- **Apertura e impegno**

Apertura: l'intero team deve essere aperto ai cambiamenti di ogni tipo

Impegno:

- Scrum Master: si impegna ad instanziare la cultura Scrum
- Project Owner: si impegna con gli stakeholder per ottenere un costante incremento delle soluzioni disponibili
- Team: dichiara il proprio impegno nelle sessioni di Sprint Planning

Rispetto

Ogni singolo membro dello Scrum Team deve rispettare i propri colleghi.

Il processo

Step 1

1. Stesura del Product Backlog, una struttura gerarchica che contiene i concetti principali del progetto;
2. Creazione di una lista contenente le User Stories prioritizzate;

Step 2

Il processo scrum si scomponete di Sprint:

- Durata costante
- Inizia con una seduta di gruppo in cui si pianificano le attività e gli obiettivi dello Sprint
- design > codifica > test
- incrementa ad ogni Sprint il Minimal Viable Product (MVP)
- Finisce con il rilascio di un prodotto aggiornato

Step 3

Flusso di lavoro: ogni membro del team può modificare lo sprint del backlog, che di solito risiede in un tabellone detto TaskBoard.

I membri prenotano il lavoro da fare su scelta personale.

Step 4

Daily Scrum:

- Durata: 15 minuti
- Ogni membro racconta la sua situazione, niente problem solving
- Scopo: evitare incontri inutili

Pratiche di sviluppo:

Ogni team è libero di dotarsi delle metodologie più adatte, anche se molti tendono a utilizzare molte delle pratiche Agili indicate dall'Extreme Programming (XP)

Burndown Chart:

andamento in tempo reale di uno sprint, dando un valore temporale ad ogni step dello sprint, spesso utilizzato con lo sviluppo agile, e di conseguenza viene preso in considerazione dallo Scrum Master.

La stima del tempo rimanente viene aggiornato dal programmatore

Cancellazione dello sprint:

Il Product Owner può cancellare uno sprint durante il suo svolgimento se l'obiettivo dello sprint è obsoleto, ad esempio se il progetto subisce una modifica o se l'obiettivo si rivela irraggiungibile

Step 5

Sprint review:

- Durata: 2 – 4 ore
- Il team presenta i risultati ottenuti, cosa è stato o non è stato completato in questo sprint

Viene sviluppato il Potentially shippable Product Increment:

Il prodotto potenzialmente rilasciabile è la versione compilata e integrata delle User Story della presente iterazione (e tutte le precedenti)

Step 6

Sprint retrospective:

- Durata: 3 ore
- Riflessione relativa al processo: cosa è andato bene e quali impedimenti sono stati trovati?
- Richiede la partecipazione di tutto il Team Scrum, compreso SM e PO
- Obiettivo: esaminare l'ultimo sprint e identificare migliorie potenziali
- Creare un piano per attuare i miglioramenti al modo di lavorare in team

Step 7

L'organizzazione decide la cadenza di rilascio al cliente:

- al termine di “uno sprint”, di “un gruppo di sprint” o “al completamento di ogni feature”

Step 8

Backlog Refinement

Attività in cui il PO e il DT modificano, ri-stimano e riordinano il Backlog sulla base dell'esperienza acquisita.

Recap del processo Scrum

Project Owner (PO)

Scrum Master (SM)

Developer Team (DT)

Stakeholders (S)

Team relativamente piccoli, 5 – 9

Potrebbero esserci più PO, o più DT, o addirittura Scrum di Scrum

Tipici problemi con Scrum :

- Ignoranza dei valori agili e di Scrum
- Prodotto software non testato a fine Sprint (cattiva definizione di Fatto)
- Backlog non pronto all'inizio di uno Sprint (cattiva definizione di Ready)
- Mancanza di supporto dai manager o stakeholders

Patti e funzioni nel team :

- il PO nel Team difenderà gli interessi degli Stakeholder
- agli Stakeholder verranno poste solo domande utili allo sviluppo
- alla fine di ogni sprint gli Stakeholder valideranno le nuove funzionalità
- Lo Scrum Master verrà chiamato a rimuovere gli ostacoli
- il Developer Team non può cambiare funzionalità senza il consenso dell'intero team

XP o Scrum?

XP	Scrum
Orientato alla qualità (test driven)	Orientato al project management
Iterazione: 1-2 settimane	Sprint: 2-4 settimane
Requisiti sempre modificabili	Req modificabili alla fine dello sprint
Il cliente ordina le storie	Il team ordina le storie
Coaching informale	Scrum master certificato
Buone pratiche tipiche di XP: TDD, Pair programming, planning game, refactoring	Buone pratiche tipiche di Scrum: Retrospettiva post-mortem, uso di strumenti di Project management, planning poker

☰ Progettazione ☰

A differenza della fase di Analisi in cui si cerca di capire il problema, la Progettazione è il processo in cui le specifiche vengono trasformate nell'architettura di sistema:

- insieme di moduli
- descrizione delle funzioni dei moduli (progettazione dettagliata)
- come i moduli si relazionano tra loro (progettazione architettonica)

Progettazione architetturale

In questa fase viene eseguita la divisione in sottosistemi e moduli che a loro volta vengono scomposti in componenti. L'unione di queste divisioni creano un sottosistema i cui i moduli forniscono servizi ad altri moduli e i componenti vengono implementati direttamente.

In particolare, dobbiamo sapere come un modulo deve relazionarsi agli altri moduli (architetture) e definire le funzioni di ogni modulo, specificando come realizzarle (dettaglio).

Le principali strategie per la modularizzazione sono:

- top down**, in cui si parte da un sistema globale andandolo a decomporre in pezzi più piccoli (moduli)
- bottom up**, si parte dai singoli e semplici moduli, che vengono poi aggregati fino a costruire il sistema completo.

Architettura d un sistema software

L'architettura di un sistema (unione di moduli e descrizione di questi) è fondamentale per ottenere una dimensione d'insieme del progetto, rendendo più facile capirne la complessità.

Un importante tipo di architettura è quella client-server, che ha la caratteristica di essere asimmetrica (ogni componente server, client, rete ha un ruolo specifico e non interscambiabile si basa sul paradigma richiesta (client)-risposta (server)).

PRO

- L'aggiunta di nuovi client è semplice
- La distribuzione dei dati è semplice (client chiede, server risponde)

CONTRO

- Lo scambio di messaggi impegnava spesso molte risorse
- I server eseguono azioni ridondanti

L'architettura di rete può essere divisa in:

-**2 tier** -> in cui partecipano due componenti, può a sua volta dividersi in:

- zero client, tutto il lavoro è delegato ai server
- thin client, il server si occupa di gestire i dati e della parte algoritmica, il client rappresenta solo l'interfaccia
- thick client, il server gestisce solo i dati, tutto il resto è delegato al client

-**3 tier** -> l'architettura si spezza in 3 componenti invece che in 2, il client si occupa dell'interfaccia e il server della gestione dati, viene aggiunto un middleware che si occupa di far interagire client e server.

Questa è l'architettura standard di un sistema informativo in quanto è modulare (generale ed adattabile) ed è ottimo per isolare e rendere i componenti riutilizzabili

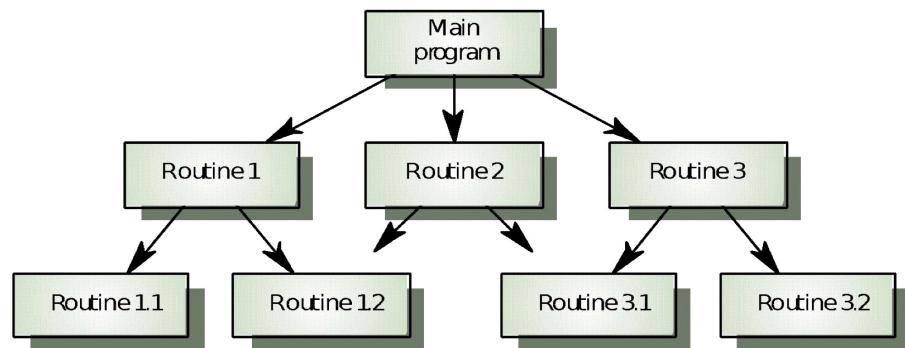
Controllo

Il controllo può essere:

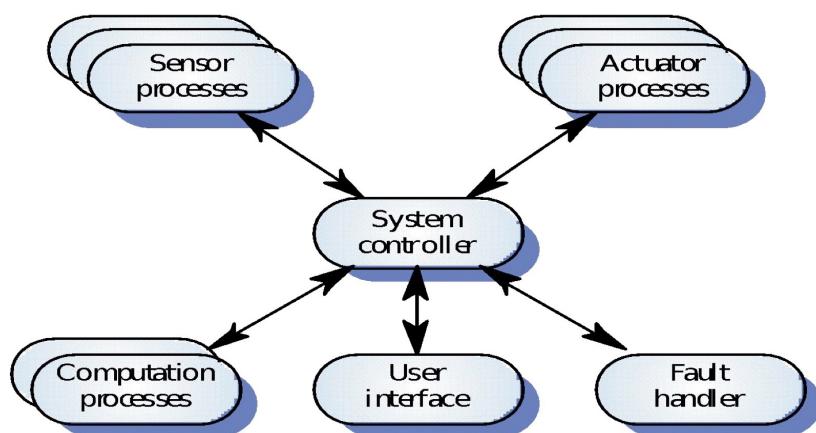
-Centralizzato, in cui un singolo sotto-sistema esegue il controllo sull'intero sistema

In questo sistema si distinguono due modelli:

-quest/response, un programma principale fa partire il controllo che poi si dirama nelle gerarchie di livello inferiore (applicabile a sistemi sequenziali)

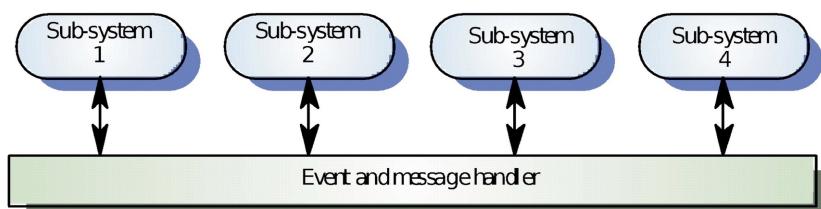


-master/slave, in cui un sotto-sistema controlla attivazione, spegnimento e coordinazione delle attività di tutti gli altri sotto-sistemi (applicabile a sistemi concorrenti)

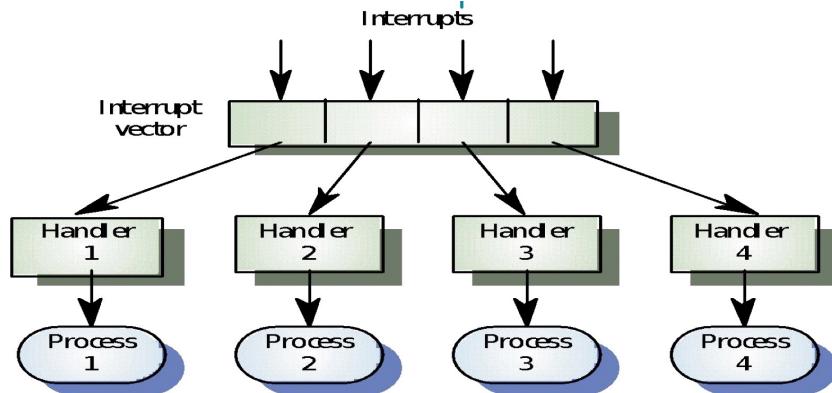


-Ad eventi, in cui ogni sotto-sistema risponde agli "stimoli" provenienti da altri sotto-sistemi, anche qui possiamo distinguere due modelli:

-broadcasting, ogni evento che viene ricevuto viene inviato a tutti i sotto-sistemi



-interrupt, in cui ogni interruzione blocca il programma in corso e forza l'esecuzione di un'altro programma legato al relativo processo



Progettazione al dettaglio

Questa fase specifica le caratteristiche di un modulo e spiega ai programmati cosa bisogna implementare, gli obiettivi di questa fase sono il creare moduli indipendenti ma connessi tra loro senza dover conoscere la loro struttura interna (black box).

Questo è ottenuto tramite l'interfaccia di un modulo, che descrive:

- funzionalità realizzate nel modulo
- come si utilizza (quali funzioni devono essere invocate)
- valori di input
- valori di output

Nella OOP ogni modulo viene pensato come un insieme di oggetti (classi, metodi) e con interfacce molto definite, qui possiamo distinguere due modelli:

- statici (class diagram)
- dinamici (interaction diagram)

Altra caratteristica importante di un modulo è la riusabilità, che è la probabilità che un determinato modulo possa essere riutilizzato in un progetto differente, vengono preferiti perché sono facilmente modificabili ed adattabili, hanno ricevuto più test ma costano molto più di moduli non riusabili.

MODELLO MVC

“È un modello architettonale di sviluppo del software per applicazioni o siti web dinamici. Ha l'obiettivo di ottimizzare le operazioni di manutenzione e aggiornamento del Software al costo di aumentare le spese di costruzione del software si riducono notevolmente le spese di manutenzione e aggiornamento.”

- è molto utilizzato in applicazione
- utilizzato in framework desktop e web come Django, Angular, Vue

MVC sta per MODEL, VIEW, CONTROLLER

Il Model rappresenta (o contiene) tutti i file informativi, attributi o database implementando anche tutta la struttura del database in evitando così di scrivere le query.

Nel model si crea la struttura del programma.

Esempio se prendessimo il FileSplitter sarebbero tutti gli attributi che vengono modellati come oggetti ovvero gli “elementi base”.

La View dal nome è l'interfaccia utente CLI, GUI, TUI, implementata a schermo ciò che è stato già costruito tramite il Controller e il Model, può essere implementata con dei template.

Il Controller è la logica del nostro programma, colui che controlla e gestisce il nostro model,

Esempio: in un programma Java tutti i metodi che manipolano gli attributi sono rappresentati dal controller

ad esempio nel nostro FileSplitter sono quegli oggetti che hanno i metodi per gestire e manipolare oggetti passati dal model, quindi i vari oggetti con i metodi che gestiscono la divisione dei file.

- I moduli sono strettamente collegati ma fondamentalmente indipendenti.
- Model View e Controller possono essere modificati senza compromettersi l'uno con l'altro.

Esempio1:

La mia View può essere una CLI, per passare ad una GUI basterà cambiare il file dentro la cartella View,

I dati a schermo saranno sempre gli stessi perché passati dal Model e Controller.

Esempio2:

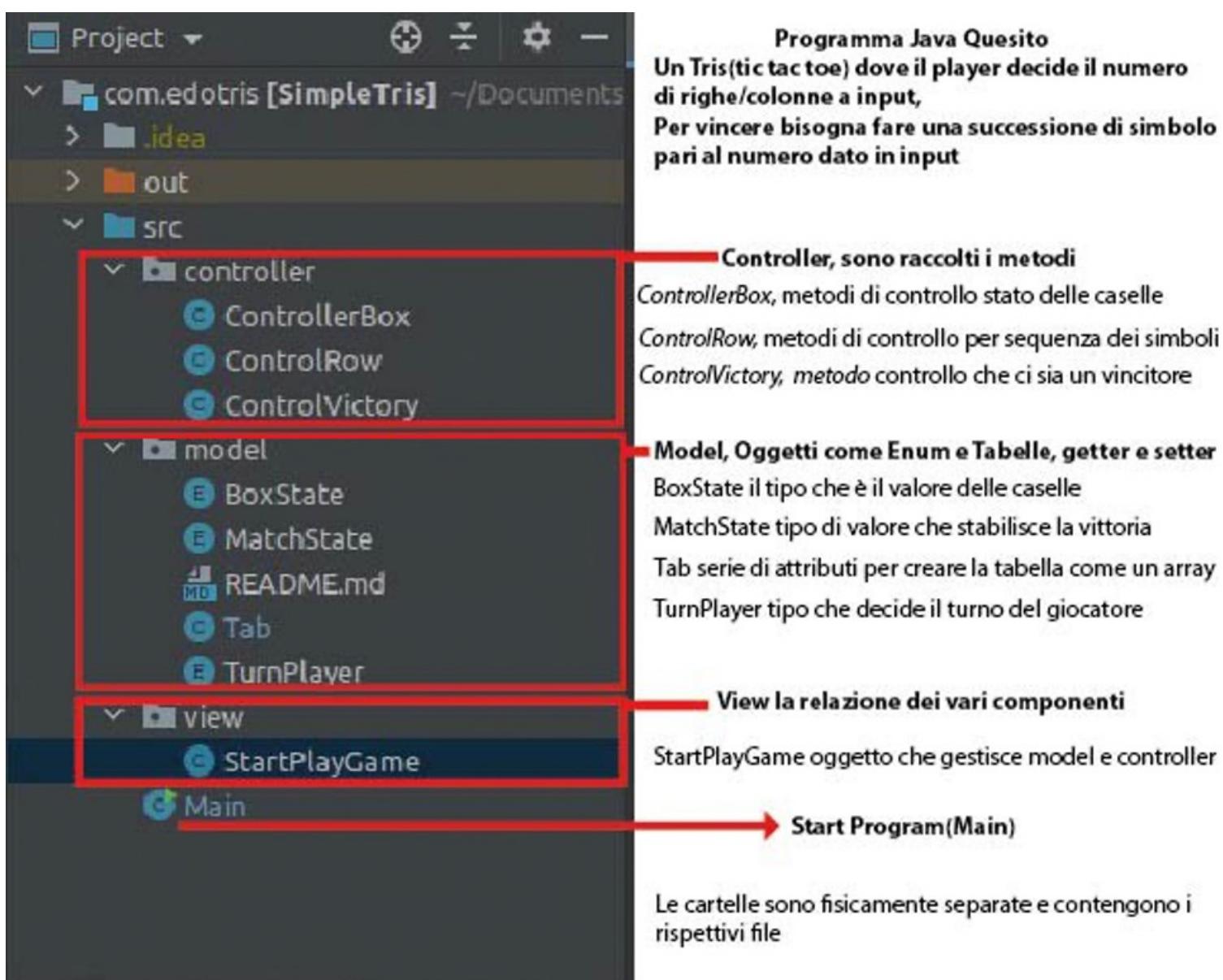
Un Database MySQL viene gestito tramite Model se decidiamo di cambiare Database possiamo modificare il singolo file all'interno del File Model.Database senza modificare le restanti parti del programma

Ricapitolando

- MVC è un sistema Modulare
- si ha un encapsulamento delle risorse
- 3 componenti della nostra architettura sono individuali e modificabili senza corrompersi a vicenda
- Al costo di una alta complessità si riduce la difficoltà di gestione e manutenzione
- l'implementazione di Framework riduce la difficoltà di gestione ma con una alta curva di apprendimento

In un sistema Java, MVC nella pratica viene rappresentato con fisicamente 3 Cartelle:

- il Model rappresenta gli oggetti che contengono gli attributi, setter e getter
- il Controller rappresenta gli oggetti che contengono solo i metodi
- la view rappresenta a schermo la relazione tra Model e Controller



Che cosa si intende per modello?

- Un modello è una astrazione che cattura le proprietà che ci interessano (ovvero proprietà utili).

Perché usare dei modelli?

- Per conoscere e capire il soggetto in analisi e comunicare ad altri la propria conoscenza (= capire se si è compreso davvero il soggetto).

I progetti software sono complessi, coinvolgono molte persone e le loro caratteristiche cambiano nel tempo = c'è bisogno di capire su che cosa concentrarsi all'interno dello sviluppo, per farlo usiamo un metodo di sviluppo che si basa su:

- **Linguaggio di modellazione** = notazione per rappresentare entità del sistema ed esprimere caratteristiche importanti del progetto
- **Processo** = serie di passi per la produzione del progetto

Linguaggio di modellazione – UML

UML è un linguaggio semiformale e grafico (attraverso l'uso di diagrammi) che serve per capire, modificare, mantenere o documentare delle informazioni del sistema (artefatti). Il linguaggio è indipendente dall'ambito del progetto, dal processo di sviluppo, dal linguaggio di programmazione, ma ha delle regole sintattiche e semantiche che vanno rispettate.

Struttura modello UML

Il modello è costituito da:

1. **Viste** = mostrano diversi aspetti del sistema tramite un insieme di diagrammi.

Esistono due possibili approcci per la divisione in viste, in particolare il "4+1 viste di Krutchen" divide il modello in cinque viste:

- **Logica** che divide tutto in classi, oggetti e relazioni
- **Sviluppo** che organizza il sistema in blocchi strutturali
- **Processuale** che cattura gli aspetti del design (es. sincronizzazione thread/processi)
- **Fisica** che descrive la mappatura del software sull'hardware
- **"Use case view"** che descrive i requisiti del sistema

2. **Diagrammi** = rappresentano graficamente una parte del modello composta da viste.

Esistono 14 diagrammi raggruppati in due categorie distinte:

- **Diagrammi di struttura** che mostrano la struttura statica del sistema e delle sue parti e le relazioni presenti tra queste.
- **Diagrammi di comportamento** che mostrano il comportamento dinamico degli oggetti del sistema ovvero la loro evoluzione nel tempo e le dinamiche delle loro interazioni

3. **Elementi del modello** = concetti che ci aiutano a realizzare i vari diagrammi (es. attori, caso d'uso...)

Un UML composto da:

- Entità strutturali

- o Strutturali vere e proprie (classi, interfaccia)
- o Comportamenti (quindi come un oggetto parla con un altro)
- o Di raggruppamento (quindi i package, come abbiamo visto in java) o Informativa (annotazioni)

- Relazioni

Collegano fra di loro le entità:

- o Associazione
- o Dipendenza
- o Generalizzazione o Realizzazione

- Diagrammi

Permettono di vedere il modello come abbiamo visto da diverse prospettive. I principali sono:

- o Diagramma dei casi d'uso
- o Delle classi
- o Interazioni
- o Sequenza, comunicazione, interazione generale, temporizzazione
- o Delle attività
- o Di stato

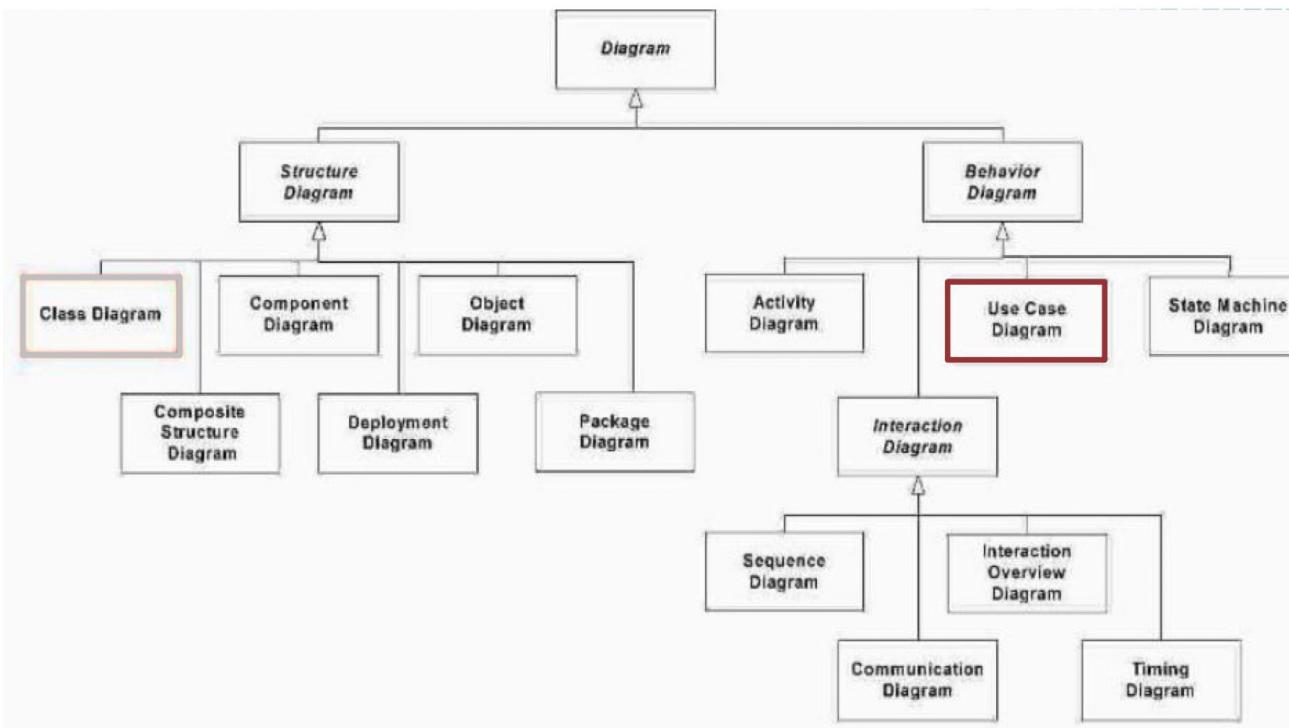
Esistono diversi software che permettono di gestire gli strumenti UML. Lo strumento proprietario tipico è il RATIONAL ROSE (quello sviluppato dalla compagnia che ha creato il linguaggio). Esiste uno strumento open source, ne esistono diversi, ad esempio ArgoUML. Oppure esistono anche strumenti ultralight come UMLETINO.

Quindi in sintesi :

- Modellare è indispensabile per qualunque progetto non banale.
- UML è un linguaggio general-purpose per la modellazione.
- non è perfetto, ma è potente e costituisce uno standard diffuso e accettato.
- costruire un buon modello è difficile.

Diagramma dei casi d'uso

(Ci servirà di più nella fase di analisi e di primissima progettazione. Questi tipi di diagrammi spesso si trovano inseriti anche negli SRS)



Tipo di diagramma comportamentale, di solito il primo che si affronta.

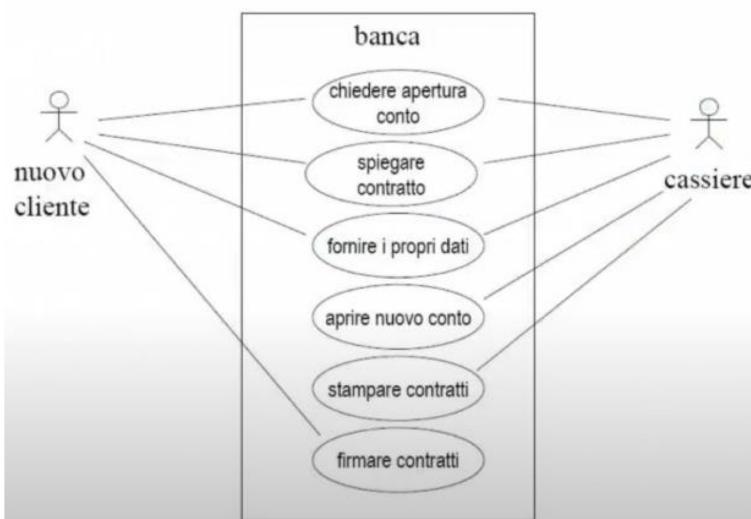
È un diagramma che esprime un comportamento, l'oggetto è un sistema o una sua parte e individua 2 cose importanti:

1. Chi o cosa ha a che fare con il sistema (ATTORE)
2. Che cosa l'attore può fare (CASO D'USO)

È il primo tipo di programma che si fa perché di solito è quello che vede le cose più da un punto di vista generale, un approccio più TOP-DOWN. Quindi analizza il comportamento del sistema MA NON considera minimamente in che modo il sistema lo realizza, può essere utilizzato anche per la verifica funzionale, cioè si controlla se a partire dal diagramma il sistema fa quello che promette

Quindi serve per descrivere i requisiti (anche se non coincide con l'analisi), convalida l'architettura, verifica il sistema.

Un esempio:



Interazione nuovo cliente che apre un conto in banca, nella colonna centrale vediamo i casi d'uso. La linea collega chi vuole fare qualche cosa, il cliente è l'attore, il rettangolo è il confine del sistema, e all'interno come già detto sono i casi d'uso, il cassiere è l'altro attore.

- "i desiderata" sono ciò che il cliente desidera.
- Formalizzare "i desiderata" in requisiti è una delle parti più difficile delle parti dell'ingegneria del software perché una specifica errata porta ad un software sbagliato.
- Il diagramma e la modalità dei casi d'uso aiutano l'interazione con il cliente e migliorano l'estrazione dei requisiti (funzionali) Un esempio di desiderata:

Vorrei vendere i manufatti che realizzo...

Non vorrei un mercato locale...

Vorrei gestire gli ordini da qualunque posto perché viaggio...

Sono sempre descritti in modo vago e non è chiaro che cosa viene fatto e da chi. Riorganizzandolo potremmo vederli in questo modo:

Vorrei avere la possibilità di creare un catalogo dei miei manufatti... Vorrei un catalogo liberamente consultabile da chiunque...

Vorrei che gli interessati all'acquisto potessero inviarmi un ordine, che io provvederò ad evadere previa una qualunque forma di registrazione...

Viene schematizzata meglio e verrà formalizzata in qualche modo.

CASO PRATICO

L'algoritmo generale per la modellazione dei casi d'uso è:

- Estrazione dei requisiti
- Individuare il confine del sistema
- Individuare gli attori
- Individuare i casi d'uso
 - o Quindi specificare il caso d'uso
 - o Creare gli scenari

ATTORE

L'attore specifica un ruolo assunto da un utente o un'altra entità che interagisce col sistema nell'ambito di un'unità di funzionamento (caso d'uso).

È inoltre l'istanziatore del caso d'uso e scambia informazioni con il sistema.

La cosa importante è che l'attore è esterno al sistema.

L'importante è individuare gli algoritmi.

Qui in seguito una serie di domande utili per individuarli:

- Chi/cosa usa il sistema?
- Che ruolo ha chi/cosa interagisce col sistema?
- In quale parte dell'organizzazione è utilizzato il sistema?
- Chi/cosa avvia il sistema?
- Chi supporterà e manterrà il sistema?
- Chi/cosa avvia il sistema?
- Chi supporterà e manterrà il sistema?
- Altri sistemi interagiscono col sistema?
- Ci sono funzioni attivate periodicamente? es. backup
- Chi/cosa ottiene o fornisce informazioni dal sistema?
- Un attore ha diversi ruoli?

CASI D'USO

I casi d'uso sono semplicemente un ovale con dentro una descrizione sintetica. È definito come una specifica sequenza di azioni, incluse eventuali sequenze alternative o di errore che un sistema può eseguire interagendo.

Quindi in generale è un qualcosa che l'attore vuole ottenere dal sistema in qualche modo, ed è descritto dal punto di vista dell'attore e spesso causato da lui.

Domande utili:

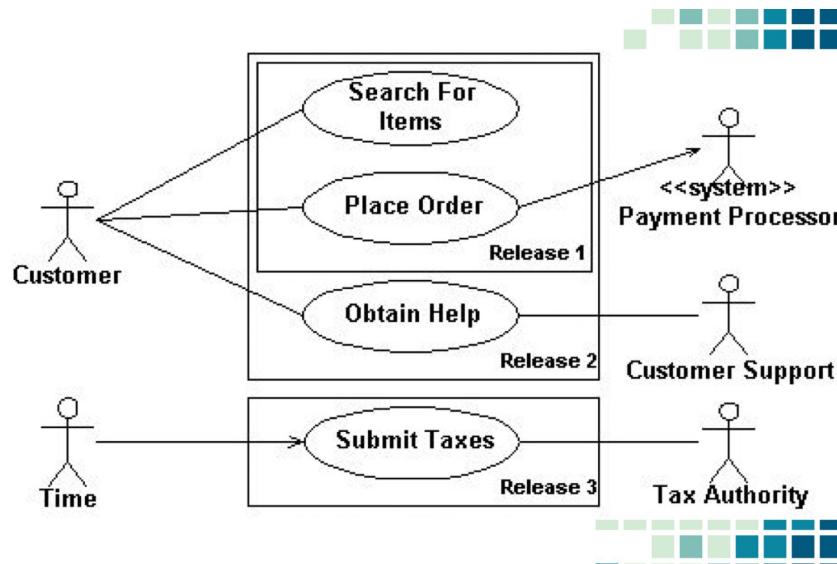
- Ciascun attore che funzioni si aspetta?
- Il sistema gestisce (archivia/fornisce) informazioni? Se sì, quali sono gli attori che provocano questo comportamento?
- Alcuni attori vengono informati quando il sistema cambia stato?
- Gli attori devono informare il sistema di cambiamenti improvvisi?
- Alcuni eventi esterni producono effetti sul sistema?
- Quali casi d'uso manutengono il sistema?
- I requisiti funzionali sono tutti coperti dai casi d'uso?

Come si fa ora a recuperare informazioni sui casi d'uso?

- Interviste con gli esperti del dominio
- Bibliografia del dominio del sistema
- Sistemi già esistenti
- Conoscenza personale del dominio

IL SISTEMA

Di solito è delineato in questo modo:



ASSOCIAZIONE

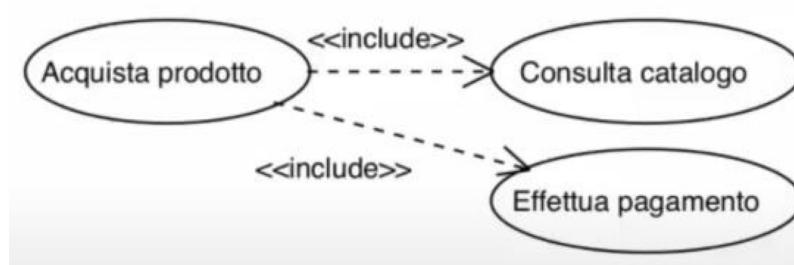
L'associazione collega gli attori con i casi d'uso, un attore si può collegare solo ai casi d'uso, un caso d'uso non si può collegare ad un altro caso d'uso riguardante lo stesso argomento.

Si possono inoltre aggiungere info opzionali quali cardinalità, ruolo ecc... come visti in database. Esistono 2 tipi di relazioni interne tra i casi d'uso e sono inclusione ed estensione.

Inclusione:

- Completamento o obbligatorio
- Indica un comportamento obbligatorio ed è spesso un prerequisito per effettuare il caso d'uso.

Se voglio acquistare un prodotto ad esempio devo obbligatoriamente "consultare il catalogo" e "effettuare il pagamento"

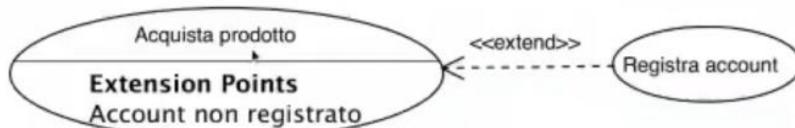


Estensione:

- Comportamento opzionale



Si può indicare il caso in cui si estende



Cosa sappiamo delle User Stories?

- Favoriscono la conversazione tra utente e team
- Devono essere semplici e non tecniche
- Permettono una stima dei tempi da parte di chi li deve realizzare
- Ciclicamente aggiornate, formano la base degli sprint

Non abbiamo "vere" conversazioni, solo esposizione di idee indipendenti.
Le user story sono un ottimo strumento di pianificazione dello sviluppo.
Non sono però uno strumento per la scrittura dei requisiti, quindi si rischia di disperdersi senza raggiungere alcun obiettivo.

User Story Mapping (UML)

La UML visualizza il viaggio dell'utente come se fosse un viaggio, e ha diversi vantaggi:

- Prospettiva centrata sull'utente
- Aiuta la prioritizzazione e l'identificazione degli obiettivi
- Rivelà più facilmente le dipendenze e i rischi

La UML è un modello visivo del backlog che dà una rappresentazione immediata del progresso.

- Raccontare e visualizzare storie, non solo scriverle
- Minimizzare output, massimizzare risultati e impatto.
- Costruire MVP per verificare l'impatto sul mercato/cliente

Story mapping in 4 passi

1) Centrare il problema (Framing the picture)

- Che cosa vuole ottenere l'utente?
- Parlare con gli Stakeholder, analisti, project manager...

2) Raccontare la storia (Mapping the picture)

- Scrivere la storia completa sotto forma di passi su schede

3) Organizzare

- Mettere su piani diversi attività significative e semplici user story
- Esplorare flussi alternativi, correggere errori ed eccezioni

4) Piano di rilascio

- Identificare le funzionalità che compongono il primo rilascio, il MVP
- Ripartire le features/storie rimanenti in release successive

Analisi vs Progettazione

L'analisi è la prima fase della progettazione di software e si occupa di descrivere il problema da risolvere, la Progettazione invece ha come obiettivo quello di completare l'Analisi affinché si possa passare alla fase successiva (Implementazione).

Come procedere nella fase di Analisi:

- Raggruppare le entità del problema in delle classi
- Capire quali attributi e quali operazioni devono essere forniti dalle classi
- Disegnare una mappa delle relazioni tra le classi
- Individuare il comportamento delle classi con i diagrammi di comportamento

Le classi di analisi sono molto spesso formate dai nomi presenti nella specifica e nella documentazione, per individuare questi nomi ci avvaliamo di due metodi:

• Analisi nome-verbo

- o Si analizza la documentazione evidenziando nomi e verbi, i nomi e la descrizione di questi (Es. Conto corrente e numero di conto corrente) sono i candidati a diventare classi o attributi, i verbi invece potrebbero diventare responsabilità di classe.

• Analisi CRC, che sta per:

- o Classe, raggruppa gli oggetti più importanti
- o Responsabilità, compiti da eseguire
- o Collaborazioni, oggetti che lavorano insieme per soddisfare una responsabilità

Queste informazioni vengono poi organizzate in CRC cards

Class name:	Superclass:	Subclasses:
Padrone	Persona	
Responsibilities	Collaborations	
Invita	Invito, Persona, Lista	
Compra	Denaro, Negozio, Cibo, ...	
Sa il giorno della festa	Agenda	

La sequenza con cui vengono raccolte queste informazioni è:

1. Determinare le classi candidate

- Consultare la documentazione
- Segnalare nomi ed espressioni
- Stilare la lista dei candidati
- Brainstorming per trovare ulteriori candidati

Una birreria è frequentata dai clienti e dallo staff. In particolare, lo staff raccoglie gli ordini e consegna le birre. Si paga alla cassa (e lo staff può dare il resto se necessario). Il gestore del pub si occupa, oltre che del servizio, anche di controllare la disponibilità di ogni birra in frigo e, se necessario, aggiungerne altre.

Una **birreria** è frequentata dai **clienti** e dallo **staff**. In particolare, lo staff raccoglie gli **ordini** e consegna le **birre**. L'**avventore** paga alla **cassa** (e i camerieri possono dare il **resto** se necessario). Il **gestore del pub** si occupa, oltre che del **servizio**, anche di controllare la disponibilità di ogni **birra** in **frigo** e, se necessario, aggiungere **bottiglie** dal **magazzino**.

2. Stabilire le classi effettive

- Le classi vengono divise in categorie (Critica, Indeciso, Irrilevante)
- Si eliminano sinonimi e ripetizioni
- Si distinguono le classi dagli attributi

- | | |
|---|--|
| <ul style="list-style-type: none">▪ Birreria (non parte del sistema, <u>è</u> il sistema)▪ Cliente▪ Staff▪ Avventore (sinonimo)▪ Ordine▪ Birra▪ Cameriere (sinonimo)▪ Cassa (da rivedere)▪ Resto (attributo) | <ul style="list-style-type: none">▪ Gestore▪ Servizio (da rivedere)▪ Frigo▪ Bottiglia (sinonimo)▪ Magazzino (non parte del sistema) |
|---|--|

3. Assegnare responsabilità

- Si stabiliscono i comportamenti delle classi focalizzandosi su cosa fanno piuttosto del come, si segue la "filosofia" Keep It Simple, Stupid (KISS)

<ul style="list-style-type: none">▪ Cliente<ul style="list-style-type: none">▫ Ordina birra▫ Riceve birre▫ Paga▫ Riceve resto▪ Staff<ul style="list-style-type: none">▫ Prende ordini▫ Prende birre▫ Serve birre▫ Prende soldi▫ Dà resto	<ul style="list-style-type: none">▪ Birra<ul style="list-style-type: none">▫ Conosce il proprio prezzo▪ Gestore<ul style="list-style-type: none">▫ Controlla birre▫ Aggiunge birre in frigo▪ Frigo<ul style="list-style-type: none">▫ Sa quante birre possiede e il loro tipo▫ Permette di aggiungere una birra
--	--

4. Assegnare collaboratori

- I collaboratori sono classi che dipendono o che sono di supporto ad altre classi, sono utili per capire la relazioni tra le classi principali

5. Identificare gerarchie

- Vengono identificate in base alle relazioni tra le classi (Is-A, Has-A, Part-Of)

Class name: STAFF	Superclass:	Subclasses: GESTORE
Responsibilities	Collaborations	
Prende le birre	Birra, Frigo	
Prende gli ordini	Ordine, Cliente	
Prende i soldi	Cliente, Cassa	
Dà il resto	Cliente	

Class name: GESTORE	Superclass:	Subclasses:
Responsibilities	Collaborations	
Controlla birre in frigo	Frigo	
Aggiunge birra in frigo	Frigo, Birra	

Come procedere nella fase di Progettazione:

- Raffinare le classi del modello di Analisi
- Trasformare le entità più astratte in modelli più concreti ed implementabili con un linguaggio object oriented tenendo presente dei vincoli del linguaggio e di piattaforma
- Le classi di analisi vengono trasformate in classi di progettazione

CRC roleplay

L'utilizzazione del roleplay è utile per raffinare le classi per avere una modellazione più precisa con specifici scenari la preparazione delle CRC cards:

1.definire scenari → 2.preparare sessione → 3.recitare scenari → 4.aggiornare CRC e scenari

Esempio:

Un partecipante assume il ruolo di un oggetto che può agire indipendentemente: esempio un oggetto di una biblioteca potrebbe agire:

"sono un libro: il signore degli anelli"cosa posso fare per te?"

l'oggetto dovrebbe presentare tutte le informazioni in suo possesso.

Partendo da questo si cerca di costruire l'intero scenario fino alla conclusione ponendosi domande utile al fine di completare le schede sull'oggetto.

UML - Diagramma delle classi

A cosa serve un diagramma delle classi?

Un diagramma delle classi consente di descrivere tipi di entità con le loro caratteristiche e le eventuali loro relazioni.

Questo diagramma rappresenta la struttura statica di una applicazione e può essere visto come un grafo (nodi → classi/interfacce, relazioni → archi).

Un diagramma offre un livello di astrazione che ci permette di isolare le caratteristiche essenziali di una entità, definire il confine dell'entità, sottolineare l'idea invece del lato concreto e gestire la complessità concentrandosi sulle caratteristiche importanti.

Definizione di classe e oggetto

Una classe è un descrittore di un gruppo di entità che condividono alcune caratteristiche comuni:

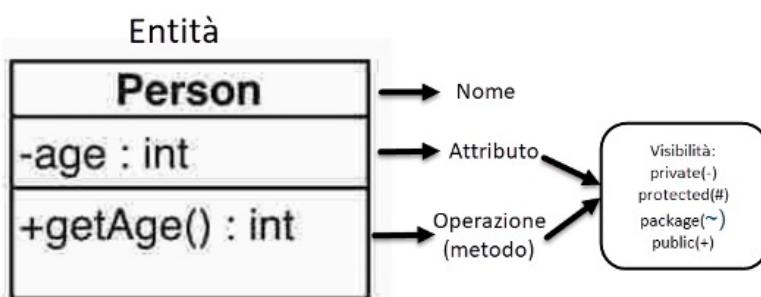
- Attributi
- Operazioni (metodi)
- Relazioni

Gli oggetti (istanze) di una stessa classe definiscono un caso specifico di un tipo di entità e differiscono tra loro per i valori degli attributi e per le relazioni che li legano ad altri oggetti.

Rappresentazione grafica classe

Una classe si rappresenta con un rettangolo diviso in al massimo 3 parti:

- **Nome** (obbligatorio) : inizia con una maiuscola (no caratteri speciali), deve essere non ambiguo e derivare al dominio del problema
- **Stereotipo** (<>>)
- **Attributi** : visibilità nome[molteplicità: tipo = valoreIniziale]
La molteplicità viene indicata con [n]:tipoDelVettore
Per gli attributi statici si utilizza la sottolineatura
- **Operazioni** : visibilità nome [(nomeParam:tipoParam, ...):tipoRestituito]
Solo il nome è obbligatorio.
Si sottolineano i metodi statici e i costruttori.
I parametri possono essere preceduti da un modificatore che indica la direzione d'uso: in (ingresso), out (uscita per riferimento) e inout (per riferimento e inizializzato).



La ragione fondamentale per usare i diagrammi di classe è di comunicare, con un formalismo indipendente dai linguaggi di programmazione, i concetti che guidano la realizzazione di un sistema orientato agli oggetti. Infatti di solito un programma scritto con un linguaggio orientato agli oggetti è complicato da leggere e capire (visto che non ha una struttura sequenziale).

Le classi di analisi

Solitamente contengono solo gli attributi e i metodi più importanti, (quelli che abbiamo evidenziato da un'analisi superficiale), in molti casi è indicato solo il nome. Assegnare un valore iniziale in una classe di analisi può essere utile ad evidenziare qualche modello.

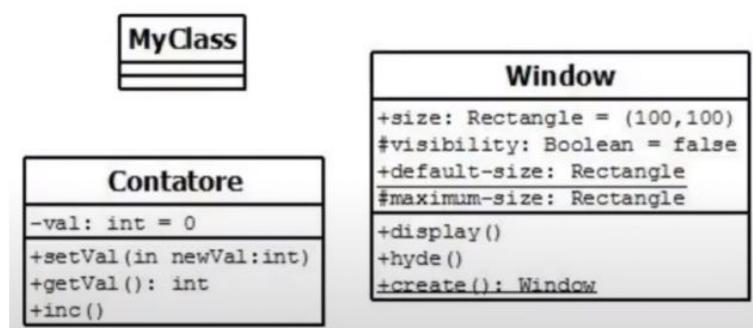
Le classi di implementazione

Possono fornire una specifica completa; diciamo che quando arriviamo al livello di avere un diagramma UML di classe di progettazione teoricamente potremmo prendere lo schema per poi passarlo ad un programmatore e questo dovrebbe essere in grado di creare un codice funzionante.

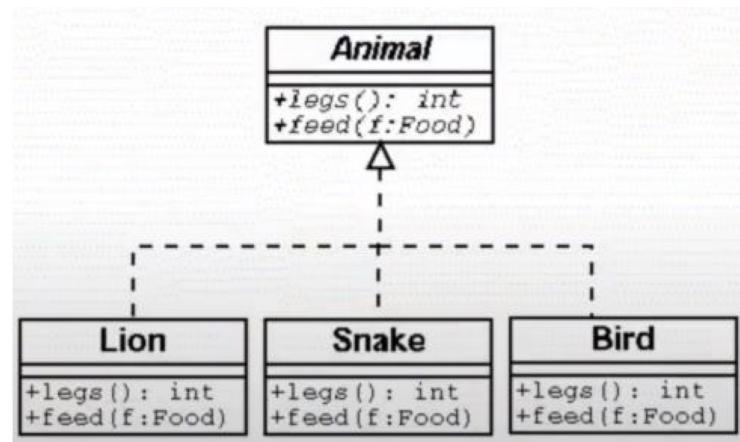
Viste dall'alto

Le classi di analisi pochi dettagli MA visione di insieme

Le classi di progettazione dettagli superiori e può darsi che si rischi di perdere la visione di insieme



Inoltre è possibile aggiungere nel diagramma delle classi note o commenti. Nel caso in cui ci siano dei metodi astratti, questi ultimi sono indicati in corsivo.



Le relazioni

Rappresentano una connessione tra elementi del grafico, possono avere associati un nome, dei ruoli, la molteplicità e dei vincoli.

Tipi di relazioni

Generalizzazione	→
Realizzazione	→
Associazione	—
Dipendenza	→
Aggregazione	◊—
Composizione	◆—

L'associazione:

È il modello di relazione più generico, significa che gli oggetti sono connessi in un qualche modo agli oggetti di un'altra classe.

Fanno riferimento l'uno all'altro. È la più flessibile e la meno vincolante.

Rappresenta l'abilità di un'istanza di mandare messaggi all'altra istanza. Può coinvolgere più classi e anche la stessa classe più volte.

La freccetta per esempio può indicare la direzione della lettura del nome (utile per aumentare la leggibilità), inoltre indica la direzione della relazione. Specificando la navigabilità si riduce la dipendenza fra le classi: I legami bidirezionali sono critici!

La molteplicità limita il numero di oggetti di una classe che possono partecipare ad un'associazione di un dato istante.

- Può essere espressa:
 - Con un solo simbolo (0..1..*)
 - Con un intervallo (0..2 1..*)
 - Con una lista di simboli o intervalli separati da virgole (0..3,5,6..9)

Esempio di associazione: gioco dei dad

- Dai casi d'uso

