Appunti di ingegneria del software

Alice Porcu e Wilma Valentino

Capitolo 1 – Introduzione

Software: *è un* ***insieme di programmi informatici*** *che svolge un qualunque compito con tutta la* ***documentazione*** *associata ad esso (questa distingue un software artigianale da uno professionale). Deve essere: manutenibile, stabile (senza errori) e usabile (user friendly).*

Ingegneria del software: *si occupa della progettazione di software (o produzione). È una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione del software, dalla nascita al funzionamento e alla manutenzione. Le attività fondamentali dell'ingegneria del software sono: la specifica, la convalida e l'evoluzione del software (manutenzione). NB: ormai la spesa in software rappresenta una frazione significante dell'economia in tutti i paesi sviluppati (molto più alta che in hardware).*

- Quali sono le **attività principali** dell'ingegneria del software?

1) **Specifica del software:** considero i requisiti che vengono forniti da chi mi richiede un sistema software. L'**ingegneria dei requisiti** si occupa di considerare i requisiti espressi dall'utente e portarli ad un linguaggio più ad alto livello (es. diagrammi, alberi,...) con una rappresentazione più rigorosa.

2) **Realizzazione** del software:

• Progettazione

• Implementazione

si appoggia sui risultati della specifica del software e dell'ingegneria dei requisiti. La fase di progettazione consiste nell'architettura del software, vari moduli, interazione con l'utente, ecc. Posso usare diversi approcci, ma quello che ci interessa è la progettazione orientata agli oggetti (UML = Unified Modeling Language ! fusione di tre modi di progettazione orientata agli oggetti. Fornisce metodi e strumenti per fare ciò).

3) **Validazione del software**: validazione dei singoli componenti sviluppati (in dettaglio) fino ad arrivare alla validazione del software nel suo complesso (prima validazioni interne e poi validazioni con l'utente finale). Se l'utente viene coinvolto in tutte le fasi non è più un utente finale ma un utente "istruito" (opposizione al cambiamento).

4) **Evoluzione del software**: il software va incontro a numerosi cambiamenti: aggiunte di requisiti, modifiche, complessità crescente ed evoluzione autoregolata, ecc.

- Qual è la differenza tra ingegneria del software e informatica?

L'informatica si concentra sulla teoria e sui fondamenti; l'ingegneria del software si concentra sullo sviluppo pratico.

Possiamo dividere i software in:

- **Prodotti generici**: sono sistemi autonomi (autosufficienti) commercializzati e venduti ad utenti generici. È lo sviluppatore a decidere cosa deve fare il software e a scegliere di modificarlo (es. app per dispositivi mobili, sistemi operativi, elaboratori di testo, Office, videogiochi).
- **Prodotti personalizzati**: sono sistemi commissionati da uno specifico utente (es. ospedale, hotel, azienda) per andare incontro alle sue necessità. Le modifiche del software vengono decise tra utente e sviluppatore (es. applicativi per gestire un reparto di terapia intensiva, forte dialogo tra medico e sviluppatore).
- **SUP**: software per la gestione dei sistemi di un'azienda, è un sistema grezzo e generico, alle richieste dei vari utenti viene configurato in un prodotto specifico e unico (via di mezzo tra software generici e specifici). Ad ogni nuova commissione non si ripartire da zero col codice, viene riutilizzato il codice generico.

La **qualità del software** non si basa solo su quello che fa ma anche sul suo comportamento durante l'esecuzione, la struttura e l'organizzazione dei sorgenti e della documentazione associata.

Caratteristiche di un buon software

- **Accettabilità (usabilità)**: deve essere comprensibile e usabile dagli utenti e compatibile con gli altri sistemi che essi usano.

- **Manutenibilità**: il software dovrebbe essere fatto in modo da adattarsi ed evolversi rispetto alle necessità del cliente che possono cambiare nel tempo (progettato quindi per una vita medio-lunga).
- **Efficienza**: utilizzo del minor numero di **risorse** nel minor **tempo** (tempo di elaborazione e uso della memoria).
- **Efficacia**: quando il software è in grado di soddisfare i requisiti dell'utente cioè fa davvero quello per cui è stato pensato (obbiettivo).
- **Affidabilità e sicurezza**: un software sicuro (safety) non deve causare danni fisici o economici a chi lo usa, un software sicuro (security) non deve essere violato, e un software affidabile (reliability) deve garantire il funzionamento corretto sempre.

Esistono 4 aspetti correlati dei sistemi software:

1- **Diversità (eterogeneità) - Scala**: cioè aumentare le tecniche di sviluppo software affidabili, esse devono essere flessibili cioè devono coprire più dispositivi (voglio gestire l'eterogeneità dei dispositivi su cui il cliente vuole farlo girare). Il software viene sviluppato su scala, da dispositivi più piccoli a più grandi.

2- **Variabilità:** le aziende moderne si evolvono molto rapidamente quindi bisogna essere in grado di modificare velocemente il codice in base alle nuove esigenze. Molte tecniche di sviluppo software sono lunghe rischiamo quindi di ritardare la consegna oppure di aumentare i costi di produzione.

3- **Fiducia e protezione:** bisogna garantire che il software non possa essere attaccato da utenti malintenzionati che sia possibile assicurare sempre la protezione delle informazioni.

Come si distribuiscono i costi rispetto alle attività principali dell'ingegneria del software?

Il costo principale sta nell'**evoluzione** del software (garantire una vita lunga e manutenibilità al software). Altro costo significativo sta nella **validazione** (fase fondamentale, specialmente in alcuni ambiti. Ad esempio: medico e bancario). I restanti costi si trovano nella progettazione e realizzazione.

Tipi di applicazioni:

- Applicazioni **stand alone:** autonome cioè che possono essere eseguiti su pc o dispositivi mobili senza bisogno di accedere alla rete (es. il sistema operativo)

- Applicazioni **Interattive basate sulle transizioni:** (transizioni = operazioni "tutto o niente") applicazioni che girano su un computer remoto, i dispositivi degli utenti ci accedono tramite portale (es. Amazon, e-commerce). Questo tipo di app spesso include un grosso db che viene consultato e aggiornato a ogni transazione.

- Sistemi di **controllo integrati (embedded o dedicati):** sistemi che controllano e gestiscono dispositivi hw, sono numerosi (es. sistema antibloccaggio freni, bancomat, navigatore).

- Sistemi di elaborazione **Batch:** sistemi per grandi blocchi di dati in sequenza, grossa mole di elaborazione dati (es. banche online), no interazione con utente.

- Sistemi di **intrattenimento:** per uso personale per il divertimento (es. giochi per specifiche console) , estrema interazione con l'utente molto rapida (real time rendering).

- Sistemi per **modellazione, progettazione e simulazione:** sviluppati da scienziati e ingeneri per creare modelli o processi software per fare altro software (es. AutoCAD). Sistemi eseguiti in parallelo che richiedono alte prestazioni e capacità di calcolo.

- Sistemi **per la raccolta e l'analisi di dati:** sistemi che raccolgono i dati dal loro ambiente e li uniscono ad altri sistemi di elaborazione (es. raccolta dati da dei sensori, sito Arpav).

- Sistemi di sistemi: sistemi software non indipendenti composti da un numero di altri sistemi software che si parlano fra loro (es. ebay → rimanda a → paypal → rimanda a specifica banca).

- Sistema **Data intensive:** simile a modellazione: dati analizzati e utilizzati per ulteriori servizi.

INGEGNERIA DEL SOFTWARE ED ETICA

Gli sviluppatori di software hanno delle responsabilità. Ogni strumento software dà delle responsabilità agli ingegneri che devono fornire una spiegazione di utilizzo all'utente e soprattutto fornire dei software che funzionino (comportamento onesto ed eticamente responsabile), facendo tutte le validazioni del caso. Se il software è usato male, non è più responsabilità dello sviluppatore ma dell'utente.

Esempi di associazioni che si preoccupano del connubio ingegneria-etica sono: ACM, IEEE,..

La responsabilità professionale nell'ambito dell'informatica sarà sicuramente legata a questi argomenti:

- Confidenzialità: aspetti, dettagli e informazioni che vengono da altri colleghi e dai committenti che hanno un valore e un'importanza, per cui terze parti non devono venirne a conoscenza; richiedono quindi aspetti di confidenzialità. In alcuni ambiti è più richiesta che in altri (es. informazioni cliniche). La confidenzialità non ha nulla di negativo: è un principio di qualità nell'esercizio della professione.

- Competenza: due controparti: da una parte devo richiedere all'ingegnere di essere sempre aggiornato nell'ambito in cui lavora; dall'altra non si devono fare false promesse a chi mi richiede una qualsiasi attività legata al software. Si deve essere onesti sulle capacità e le competenze che si hanno (se non si fa ciò, è eticamente non corretto).

- Onestà etica: si deve essere onesti rispetto a quello che si sta facendo (corollario del punto precedente). Proprietà intellettuale del software: come si regola la proprietà intellettuale (brevetti, diritti d'autore, licenze, ecc.). Un ingegnere del software non può essere impreparato in questo ambito.

- Uso scorretto del Software: non posso usare le mie competenze per usare in modo malevolo software fatti da me o da altri. Non posso installare software, senza che l'utente lo sappia, che servono a me per scopi personali. Le mie capacità devono servire per gli altri, al più per uno scopo positivo. Le associazioni ACM e IEEE hanno scritto una sorta di codice etico per gli ingegneri del software. Esso si sofferma principalmente sul fatto che chi sviluppa software non deve farlo per creare danno o disagio ad altri.

- Giudizio: si deve essere integri e indipendenti nei giudizi professionali (soluzioni tecniche).

- Rapporto con i colleghi: non si deve intralciare scorrettamente il lavoro di altri colleghi. Si deve essere giusti e supportare il loro lavoro.

Capitolo 2 - Fondamenti dell'ingegneria del software

L'ingegneria del software:

1) è una disciplina **INGEGNERISTICA**: vuol dire che bisogna far funzionare le cose con l'uso selettivo di tecniche, teorie e metodi. Devo risolvere problemi anche quando teorie o metodi non esistono ancora e con soluzioni sotto certi vincoli organizzativi o finanziari.

2) si occupa di tutti gli aspetti della produzione del software: quindi non solo degli aspetti tecnici dello sviluppo bensì anche della gestione dei progetti e degli strumenti di sviluppi, dei metodi e delle teorie che supportano la produzione software.

3) è importante perché: vi è un amento di richieste di software professionali. Bisogna essere capaci di produrre sistemi affidabili e sicuri, in maniera e economica e rapida.

4) è più conveniente nel lungo termine. Il mancato utilizzo dei metodi di ingegneria del software comporta costi più elevati per la fase di test, la garanzia di qualità e per la manutenzione.

Processi (di sviluppo) software

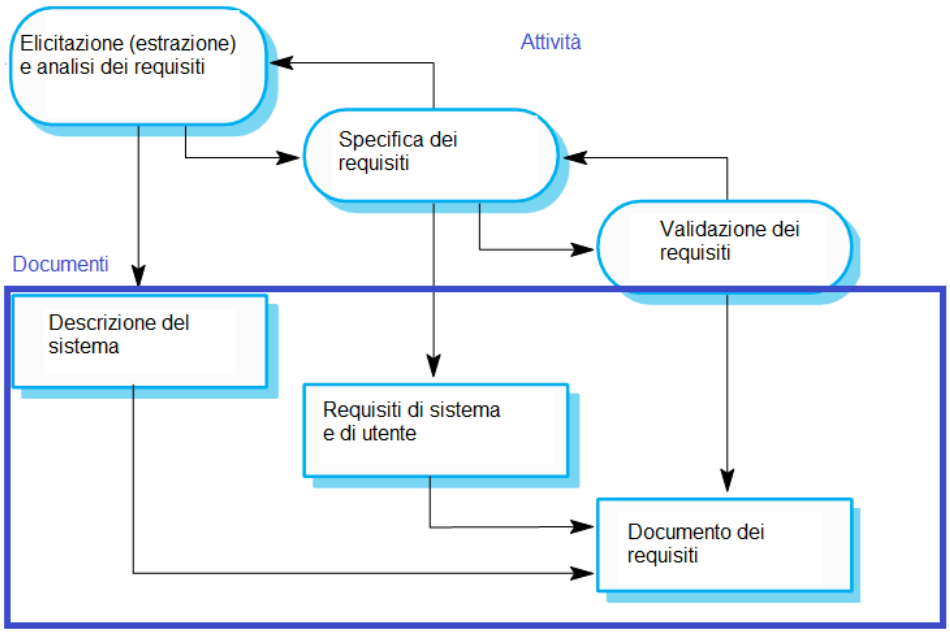
*Un **processo di sviluppo software** è l'approccio sistematico dell'ingegneria del software, si tratta di un insieme di attività che porta alla creazione di un **prodotto software.***

Attività dei processi (di sviluppo) software

Ci sono 4 attività fondamentali comuni a tutti i processi software:

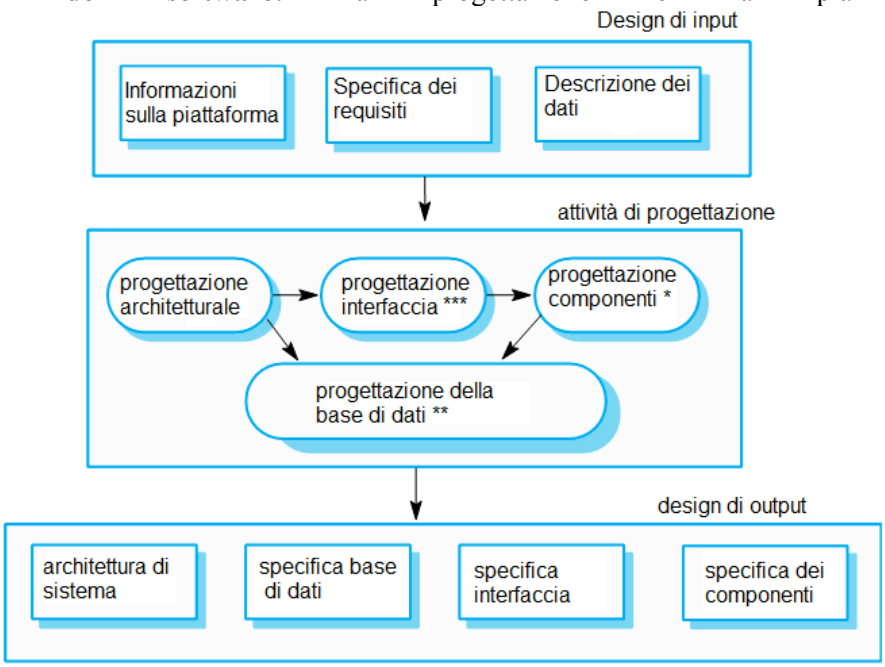
1- **Specifica del software:** clienti e ingegneri stabiliscono le funzionalità e i vincoli operativi del software da produrre (scopo e vincoli), si divide in:

- Studio di fattibilità: quello che mi viene richiesto è affrontabile in termini di costi e di risorse? Quali risorse, tempi e costi devo prevedere per arrivare alla specifica del software? (Manca nel grafico, questa parte dà come documento il rapporto di fattibilità).
- "Elicitazione" (estrazione) e analisi dei requisiti: i requisiti devono essere capiti e interpretati. Questo lavoro lo faccio sulla base di sollecitazioni da parte dell'utente (lui mi dice cosa non gli va bene usando una prima versione), utilizzo documenti e dati per aiutare la comprensione dell'utente, essi fondano le varie parti del software. Devo intervistare gli utenti e saper gestire quello che mi dicono, è difficile gestire più utenti che mi danno più specifiche.
- Specifica dei requisiti : mi sto muovendo da una parte informale ad una parte almeno un po' più formalizzata (grafici, diagrammi,...).
- Validazione dei requisiti : devo validare il fatto che i requisiti forniti dall'utente siano stati letti e interpretati in maniera corretta.



NB. La specifica dei requisiti definisce i requisiti in dettaglio, mentre la validazione valuta la validità dei requisiti stessi.

2- **Sviluppo del software:** la progettazione e la pianificazione del



software.

*= serve per sviluppare sia l'architettura sia i singoli componenti (es. design pattern). Se c'è riuso, bisogna selezionare e manipolare i componenti .

** = collezione di dati, gestita da un sistema che vi permette di accedere ai dati interrogando il sistema (su tutti i sistemi c'è sempre una qualche forma di base di dati).

*** = con interfaccia intendiamo che le singole parti del software parlano tra loro , non solo l'interfaccia utente.

NB: il riuso del software rappresenta una parte importante dell'ingegneria del software, poiché mi aiuta a non iniziare da 0 ogni volta, necessario capire quando e come riutilizzare il codice.

3- **Convalida del software:** devo prevenire gli errori e assicurare che il software corrisponda a cosa il cliente aveva chiesto. Voglio impedire di inserire codici senza senso ma verificare che il software non dia problemi.

- **Component testing**: eseguito da parte dello stesso sviluppatore, faccio la verifica di ogni singolo componente. Inseriscono: dati reali, dati interni che l'utente non vede e anche dati assurdi (es. anno bisestile = multipli di 4, non divisibili per 100 ma per 400).
- **System testing**: devo verificare che i vari componenti comunichino e facciano funzionare complessivamente in modo corretto il software. Questo test viene fatto dai programmatori, i quali inseriscono dati, non sono ancora arrivato all'utente finale.
- L'**acceptance testing:** si divide in alpha test e beta test. L'**alpha test** sono fatti da utenti vicini al software e che quindi hanno conoscenza del software stesso (polarizzati) invece il **beta testing** è una ampia gamma di utenti (eterogenei, non polarizzati).

4- **Evoluzione del software:** modifiche del software per soddisfare eventuali cambiamenti dei requisiti del cliente.

Descrizione di un processo software

Quando descriviamo processi software parliamo di attività da svolgere all'interno dello sviluppo del software (es. lo sviluppo dell'interfaccia o la definizione del modello dei dati o il tipo di dato per un informazione: float per misurare temperatura piuttosto che int).

La descrizione dei processi può includere anche :

- **Prodotti** : sono il risultato dell'attività del processo (output).
- **Ruoli e le risorse** necessarie.
- **Pre e post condizioni**: requisiti da rispettare prima e dopo un'attività (es. cosa mi aspetto che succeda al termine di una determinata attività).

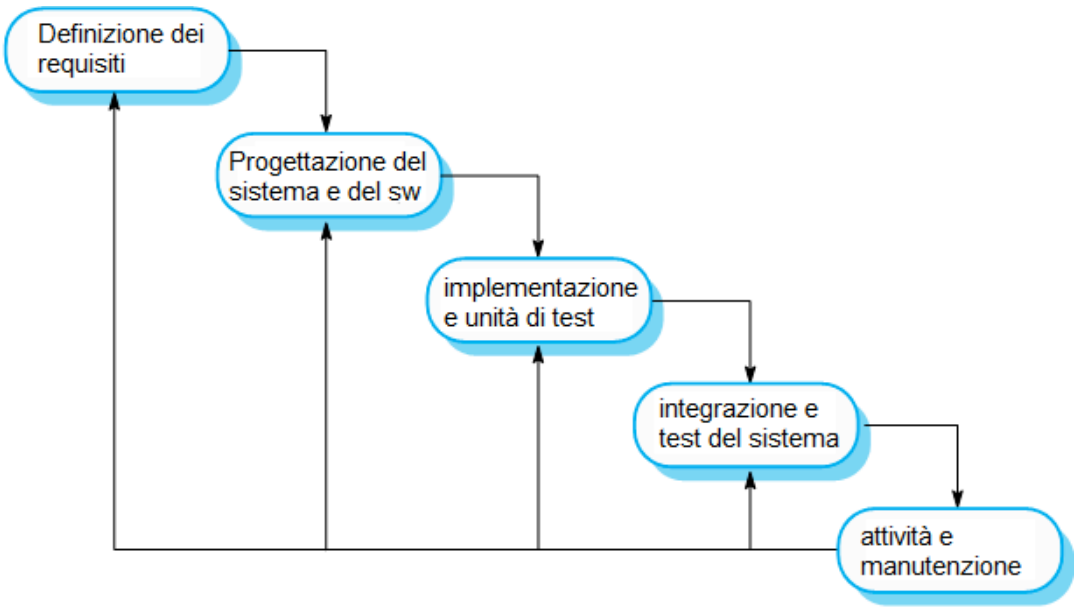
Modelli del processo (di sviluppo) software

- **Plan-driven**: attività pianificate all'inizio, guardo le tempistiche a ogni fine attività (Siamo in tempo? In ritardo? In anticipo?). Ottimo per progetti grossi con molte persone coinvolte, anche distanti fra loro.
- **Agile:** Programmazione incrementale, più facilmente modificabile e utile se i clienti cambiano spesso requisiti (non metto scadenze fisse, cambio le tempistiche lungo il progetto, fisso però l'ordine delle attività). Ottimo per poche persone che lavorano vicine, più facile reagire a un cambiamento esterno.

NB: di solito c'è un uso combinato plan-driven e agile.

Modelli di processo (di sviluppo) software

- **Modello a cascata:** è plan-driven quindi rigido, specifico a cascata tutte le attività e fasi (una conseguente all'altra).

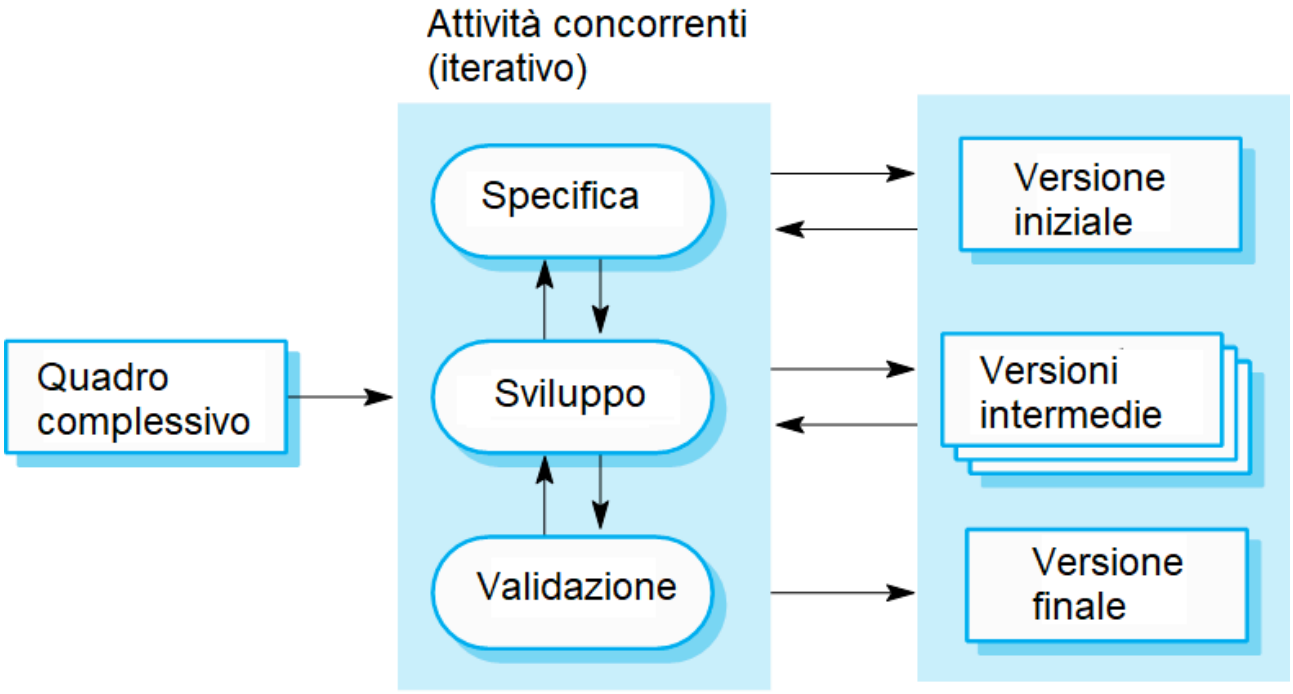


Ogni attività deve terminare prima dell'inizio della successiva. I risultati di ogni attività (output) servono come input dell'attività successiva. Solo quando arrivo all'ultima attività posso avere la possibilità di fare retroazione.

Svantaggi: approccio rigido ed impossibile da seguire se ho requisiti che vengono capiti pian piano e che cambiano in fretta o cambiano in corso d'opera (questo modello funziona solo quando la parte dei requisiti è assestata e fissa, quindi poco reale). In più se rimango bloccato con un'attività, blocco tutto il processo, l'utente finale deve aspettare parecchio tempo prima di vedere un prodotto finito.

Vantaggi: ho fin dall'inizio una chiara scomposizione dei compiti che rimane fissa dopo la definizione dei requisiti, molto utile quando ho processi grossi o dislocati in vari posti.

- **Sviluppo incrementale**: plan-driven/agile, ho delle consegne intermedie periodiche che pian piano vengono raffinate sulla base del confronto con l'utente. Prima di passare all'incremento successivo, faccio la validazione dell'incremento appena sviluppato e consegnato (incremento ciclico di sviluppo e validazione). La validazione può essere fatta dall'utente finale.



Vantaggi: l'utente vede quasi subito il prodotto (NON è completo e definitivo) e così collabora con l'affinazione delle specifiche attraverso il feedback (maggior coinvolgimento). Riesco a sopportare in modo più rapido il cambio di requisiti. Inoltre i requisiti possono essere verificati in situazioni reali dall'utente.

Svantaggi: gestire la documentazione, non c'è monitoraggio accurato, meno controllabile perché ci sono continue aggiunte. Può produrre degrado del codice e del software, perdita del controllo dello sviluppo.

Due esempi visti di processo di sviluppo agile e incrementale sono:

EXTREME PROGRAMMING - XP

Tecnica sviluppata negli anni '90 che produce un approccio estremo al processo iterativo. E' una produzione di software molto rapida: faccio nuove versioni più volte al giorno e devo far vedere gli incrementi del progetto al cliente ogni 2-4 settimane. Ogni incremento/sviluppo deve essere testato e i test devono funzionare per ogni incremento: solo quando i test hanno successo posso passare all'incremento successivo.

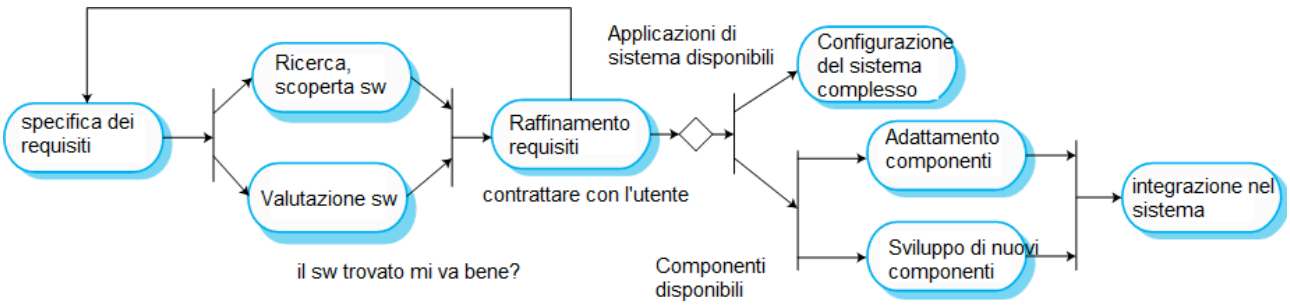
Ha come pratiche importanti:

- Test driven-development o test-first: specifico tutto prima quello che deve essere testato, poi progetto i test e poi sviluppo. In questo modo semplifico tantissimo, sistemo la comunicazione tra moduli prima ancora di iniziare a svilupparli. Inoltre, se ci sono parti di requisiti che non sono chiari, quando creo i test per ogni singola unità vengono fuori di sicuro. Questo approccio non piace agli sviluppatori.
- Refactoring (miglioramento dell'implementazione): devo continuare a migliorare il codice affinchè abbia una performance ottimale, se non integro bene le parti migliorandole, perdo la funzionalità e la prestazione. Devo essere sempre disposto a tornare indietro per semplificare o migliorare anche se non serve nell'immediato.
- Pair programming : prevede che gli sviluppatori lavorino due alla volta e producano un solo codice. Lo scopo della programmazione a coppie è ottenere un software superiore alla somma dei due software sviluppati separatamente. La coppia deve avere: lo stesso livello di competenza, sapere come discutere e responsabilità. Le coppie non sono fisse perchè così la responsabilità è condivisa in tutto il gruppo, incrementa le conoscenze di tutti. In questo contesto è richiesta una competenza più elevata che in altri contesti.
- Proprietà collettiva: tutto il codice prodotto dal gruppo è responsabilità di tutti.
- Cliente on site: all'interno del gruppo.

SCRUM

- TEAM: gruppo di persone piccolo (max di 7 persone) che andrà a sviluppare il software o parte di esso.
- OWNER: proprietario del progetto.
- MASTER: persona che si interfaccia tra il team e il proprietario, ha il compito di decidere quali sono le funzionalità che servono e tradurre le funzionalità in user stories (traduce le funzionalità espresse dal proprietario in un linguaggio più capibile dal team).
- USER STORY: funzionalità tradotte dallo scrum master per gli sviluppatori.
- INCREMENTO: componente del software che porterà ad un miglioramento del progetto finale. Si può spezzare il progetto finale in piccole parti funzionanti che andranno consegnate al cliente. Permette di evitare sorprese sia per lo sviluppatore sia per il proprietario e di cambiare direzione in corso d'opera.
- VELOCITA': si fa una stima di tutti i task che devono essere svolti dal back log per capire se ci si sta dentro alla finestra temporale richiesta.
- BACK LOG: elenco degli incrementi di un progetto più o meno spezzettato. Deve essere mantenuto dagli sviluppatori e dal proprietario del progetto. Il BACK LOG è una serie di blocchi che definisce cosa c'è da fare nel progetto.
- SPRINT: All'inizio di ogni sprint si fa una riunione (INIZIO SPRINT, 30 min) dove lo scrum master e il team decidono quali sono le funzionalità che andranno implementate nelle successive 2-4 settimane. In questa riunione si elencano i task da eseguire presi dal back log (SPRINT BACK LOG). Sempre in questa fase il team assegna una stima di risorse (ore o minuti di lavoro): la stima deve avere sia un minimo che un massimo. Alla fine dello sprint si andrà a consegnare una feature completa per il cliente. Si arriverà ad avere un prodotto finito e quindi un incremento dello sviluppo del software.
- MEETING DI FINE SPRINT: al termine dello sprint ci sarà un ulteriore meeting di 30 minuti in cui si discutono i vari problemi trovati durante lo sviluppo, le aggiunte e le rimozioni di alcune feature, le modifiche da fare, ecc.
- SCRUM MEETING: ci sarà una riunione di 10 minuti ogni giorno a inizio giornata, possibilmente in piedi, in cui si va a mantenere o gestire lo sprint back log, oppure si discute delle possibili feature ulteriori da implementare in futuro (back log).

- **Integrazione e configurazione**: plan-driven/agile, ho riuso del codice che viene assemblato da componenti configurabili già esistenti (parti generiche poi specificate per singolo utente).



È sempre più frequente, riusi e raffini quello che c'è già. Hai una struttura complessa che si fa una sola volta (backend/semi-lavorato) e poi lo configuri per lo specifico utente (frontend). Quando faccio l'analisi dei requisiti non guardo solo le richieste dell'utente ma guardo quanto le caratteristiche richieste che sono d'accordo con moduli base già pre-esistenti.

Vantaggi: non parto da 0, sviluppo più veloce e partenza più rapida

Svantaggi: ho un compromesso rispetto ai requisiti, spesso al ribasso. Il riuso e l'estensione non è così semplice, devo studiare codice complesso fatto da altri. Il software visto che non è mio può cambiare con degli aggiornamenti e non andare più bene con le mie estensioni.

NB: I sistemi molto grandi sono sviluppati usando un processo che incorpora elementi da tutti questi modelli.

GESTIONE DEL CAMBIAMENTO

Se avvengono dei cambiamenti ho un costo legato al fatto che devo re-implementare tutto (devo ripartire con l'analisi dei requisiti, capire cosa vuole l'utente). Si devono ridurre questi costi e ho due modi per farlo:

1. **Cerco di evitare il cambiamento (prevenzione):** durante il processo software includo delle attività che possono anticipare i possibili cambiamenti prima di dover rifare il tutto (cerco di anticipare).
2. **Tollero il cambiamento:** vado avanti in modo **incrementale** per cui, è vero che se c'è un cambiamento lo devo gestire, ma procedendo in maniera incrementale non devo fare delle modifiche enormi.

PROTOTIPI: per la gestione del cambiamento possiamo sviluppare prototipi (anticipano il cambiamento). Un prototipo è la versione iniziale del sistema che voglio sviluppare, viene usata per confermare i requisiti nella fase di elicitazione. Fa vedere solo le cose più importanti e serve per raffinare i requisiti ed evitare ambiguità. Riesco a far capire all'utente cosa può fare il sistema, così può dirmi cosa vuole con maggiore accuratezza.

Lo posso usare anche: nella fase di progettazione (per requisiti che si prestano a più di un'interpretazione propongo diverse soluzioni e vedo come l'utente reagisce a queste opzioni), nella fase di validazione (siccome contiene gli elementi più importanti del sistema, inizio a validare tutte le parti principali del sistema in modo sequenziale), nello studio di fattibilità.

Vantaggi: migliora la stabilità, l'usabilità del sistema, la qualità del progetto e la manutenibilità. Tengo l'utente vicino allo sviluppo del progetto. Riduco il tempo di attesa per l'utente, lo sforzo e i costi di sviluppo.

Processi di sviluppo di prototipi:

- Obbiettivi
    - Stabile gli obiettivi del prototipo
    - Definire le funzionalità del prototipo
    - Sviluppo del prototipo
    - Valutazione del prototipo

Mi concentro sugli aspetti FUNZIONALI, non guardo l'affidabilità, la sicurezza o la performance.

NB: THROW-AWAY PROTOTYPES: I prototipi sono usa e getta, non sono la base del tuo sistema e non sono documentati. Questa decisione viene presa perchè estendere i prototipi con modifiche e aggiunte porta ad una degradazione delle prestazioni, fa aumentare i costi invece che diminuirli.

Capitolo 3 - Specifica dei requisiti – Oliboni

L'ingegneria dei requisiti è il processo di stabilire i servizi che il cliente abituale richiede ad un sistema e i vincoli in base ai quali funziona ed è sviluppato. Si divide in ricerca, analisi, documentazione e verifica dei requisiti .

*Un **processo** è un insieme o sequenza di attività (non solo processo tecnico).*

*I **requisiti di sistema** sono la descrizione dei servizi forniti da esso e dei vincoli operativi. Le definizione di **requisito** può variare da una astratta di alto livello di un servizio o di un vincolo di sistema fino a un dettaglio di specifica funzionale (basso livello, matematico). Ho questa doppia descrizione perchè i requisiti possono servire a una doppia funzione:*

- Può essere la base per un'offerta per un contratto - quindi deve essere aperto all'interpretazione (alto livello).

- Può essere la base per il contratto stesso - quindi deve essere definito in dettaglio (basso livello).

NB. Stesura dei requisiti: in questa fase conta cosa l'utente chiede al sistema (non mi interessa il come dopo verrà fatto).

***Stackholders**: qualunque persona coinvolta quindi sono figure diverse con competenze diverse. Dedurre i requisiti dagli stackholders non è facile e immediato, hanno un idea generale dei requisiti che vogliono, e gli esprimono sulla base delle loro competenze e i loro interessi (es. utenti finali, gestori di sistema, proprietari del sistema).*

Tipi di requisiti

- **REQUISITI UTENTE:** dovrebbero descrivere i requisiti funzionali e non funzionali in modo comprensivo per gli utenti del sistema, senza approfondire e senza preparazione tecnica. Dovrebbero specificare solo il comportamento estero del sistema, non dovrebbero descrivere caratteristiche di progettazione. Dovrebbero essere scritti in linguaggio semplice (non tecnico, non da sviluppatore, corredati da diagrammi) e non devono far riferimento all'implementazione. Alto livello di astrazione. Un requisito utente può essere espanso in più requisiti di sistema.

- **REQUISITI DI SISTEMA:** sono versioni espanse dei requisiti utente con una formulazione dettagliata e strutturata di funzioni, servizi e vincoli. Sono utilizzati dagli ingegneri del software come base di partenza per la progettazione, possono essere una parte dell'implementazione. Dovrebbero descrivere il comportamento del sistema e i vincoli operativi, comunque scritti in linguaggio naturale o al massimo con nozioni semiformali (descrivono COSA non COME). Livello di astrazione più basso. **NB:** i requisiti utente devono essere scritti in termini non tecnici e semplici mentre i requisiti di sistema hanno aspetti più tecnici.

- **Requisiti funzionali:** descrivono i servizi (funzionalità) che il sistema deve fornire, indicano in che modo il sistema dovrebbe reagire a particolari input e come il sistema dovrebbe comportarsi in particolari situazioni. Posso anche affermare esplicitamente cosa il sistema NON deve fare (es. quando un utente richiede l'estratto conto deve fare una serie di azioni)

  Dipendono dal tipo di sistema richiesto (ma anche dagli altri sistemi con cui si interfaccia, es. richiesta a un db già esistente) possono avere diversi livelli di dettaglio.

- **Requisiti non funzionali:** vincoli sui servizi e sul processo di sviluppo. Si applicano al sistema completo (es: rispettare un certo standard come il real time)

  Si riferiscono al sistema nel suo complesso e a proprietà del sistema come: affidabilità, tempi di risposta, occupazione spazio e altri vincoli.

  Si dividono in:

  - Requisiti del prodotto: Requisiti che specificano che il prodotto consegnato deve comportarsi in un modo particolare (es. velocità di esecuzione e affidabilità).

- Requisiti organizzativi: sono una conseguenza di certe politiche e procedure dell'organizzazione (es. standard di processo utilizzati e requisiti di implementazione).
- Requisiti esterni: derivano da fattori esterni al sistema e il suo processo di sviluppo (es. requisiti legislativi).

- **Requisiti di dominio:** derivano dal dominio applicativo del sistema (es. procedura interna a una banca).

  Impongono nuovi requisiti, quindi sono vincoli ulteriori ad altri requisiti già esistenti (requisiti funzionali, es particolari requisiti legati a calcoli, documenti soggetti a copiright).

Comprensibilità: *ci deve essere uno scambio di informazioni nelle due branche specifiche, ad esempio fra informatica e medicina. Ci possono essere REQUISITI IMPLICITI (requisiti dati per scontati perché il cliente ha certe competenze) fa parte del processo capire se ci sono questo tipo di requisiti.*

Il linguaggio naturale può causare diversi problemi: mancanza di chiarezza (es. discorsi troppo lunghi), confusione sui tipi di requisiti (non facilmente distinguibili in linguaggio naturale) e mescolanza i requisiti.

I requisiti devono essere COMPLETI (tutti i servizi devono essere definiti) e COERENTI (non devono esserci contraddizioni).

Accorgimenti per minimizzare le incomprensioni:

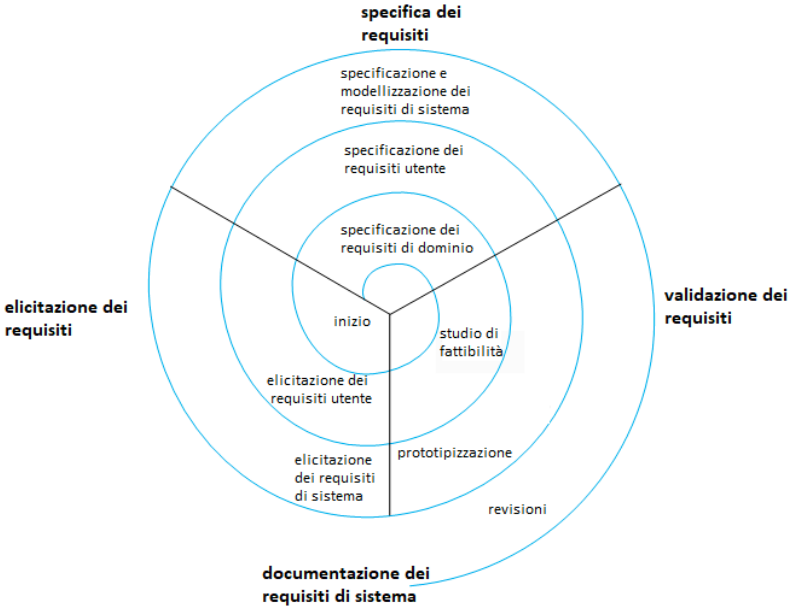
1) Scrivere tutte le definizioni dei requisiti usando un certo formato standard.
2) Usare il linguaggio in maniera coerente, distinguere bene i requisiti obbligatori (il sistema DEVE…) e i requisiti desiderati/graditi(il sistema DOVREBBE…).
3) Usare diversi stili di testo.
4) Evitare l'uso del gergo informatico.

Processi di sviluppo dell'ingegneria dei requisiti

Ci sono 4 attività comuni a tutti i processi di sviluppo di ingegneria dei requisiti:

1. Deduzione (elicitazione) dei requisiti.
2. Analisi dei requisiti.
3. Validazione dei requisiti.
4. Gestione dei requisiti.

Modello di attività di deduzione e analisi dei requisiti



La deduzione (elicitazione/scoperta) dei requisiti coinvolge personale tecnico che lavora con i clienti (stackholders) per trovare i servizi che il sistema dovrebbe fornire e i suoi vincoli.

Posso usare vari strumenti:

- **Intervista:** strumento molto importante, posso fare interviste chiuse (elenco predeterminato di domande) o aperte e conta molto il fattore umano. Le interviste sono buone per ottenere una comprensione generale di ciò che le parti interessate fanno e in che modo potrebbero interagire con il sistema.

  Svantaggi: non sono buone per la comprensione dei requisiti di dominio, ho terminologia differente o omissioni dovuti al lavoro dello stackholder.

- **Scenario:** è informale e per lo stackholder è facile perché ti racconta la sua giornata. La descrizione deve: partire dall'inizio del lavoro dell'attore, descrivere cosa succede normalmente e cosa succede se qualcosa va male, descrivere lo stato finale (output).

  Vantaggi: puoi vedere le iterazioni con gli altri attori.

- **Casi d'uso:** tecnica basata su scenari che mi permette di identificare gli attori e i requisiti. Per i casi d'uso usiamo I DIAGRAMMI DI SEQUENZA (sequence diagram: notazione UML) che sono usati per completare le informazioni. Una serie di casi d'uso dovrebbe descrivere tutti i possibili interazioni con il sistema.

- **Etnografia:** tecnica di osservazione, guardo gli attori nel loro ambiente lavorativo per dedurre requisiti non esplicitati ma che loro considerano scontati.Studi etnografici hanno dimostrato che il lavoro che svolgono è di solito più ricco e più complesso di quanto suggerito.

  Vantaggi: vedo i requisiti derivati dalla cooperazione di attività fra individui.

  Svantaggi: vedo il lavoro attuale ma non i futuri cambiamenti.

La validazione dei requisiti è importante per evitare errori ed evitare un notevole costo dopo, dimostra che i requisiti che ho raccolto definiscono in maniera corretta il sistema che il cliente vuole.

Tipi di controllo

- Validità: descrizione corretta delle funzionalità richieste.
- Consistenza: no conflitti tra requisiti.
- Completezza: i requisiti devono includere tutte le funzionalità e i vincoli descritti.
- Realismo: tecnologia, tempo e budget (posso implementare tutto con i soldi che mi danno, ce la faccio nei tempi previsti).
- Verificabilità: i requisiti sono verificabili? (la verifica deve essere chiara, ho requisiti netti ad esempio: ho massimo 2 errori al giorno).

Tecniche di validazione

- **Revisione dei requisiti**: i requisiti vengono sistematicamente analizzati da un team di revisori (può essere un singolo basta che la revisione sia periodica).
- **Prototipizzazione**: creo un prototipo e lo faccio usare.
- **Generazione dei casi di test:** faccio test per testare che il sistema segua i requisiti.

Gestione dei requisiti

Gestione dei cambiamenti dei requisiti (cambiamenti veri e propri, valuto l'impatto), evoluzione nel tempo e affinando la descrizione.

REQUISITI DURATURI: relativamente stabili di dominio (ad esempio l'ambito medico non cambio)

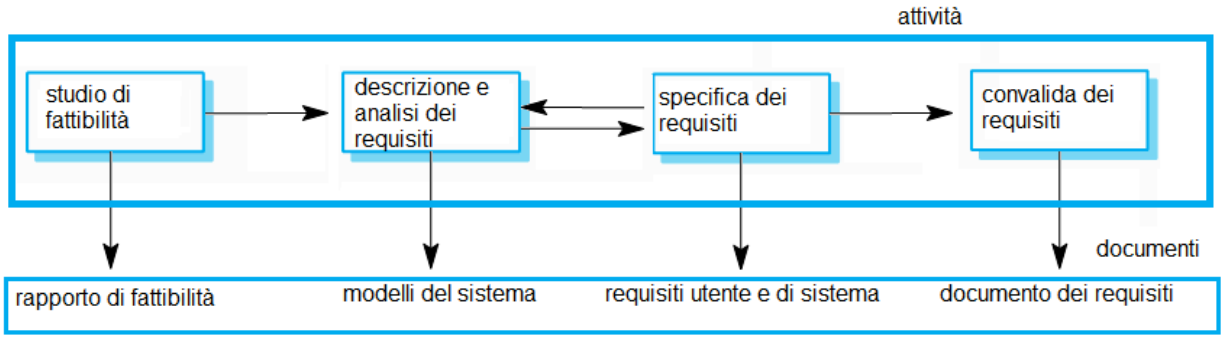
REQUISITI VOLATILI: requisiti che possono cambiare durante lo sviluppo o dopo (ad esempio politiche di governo. Si possono dividere in:

- Requisiti mutabili: possono cambiare a causa della dinamicità ambientale
- Requisiti emergenti: emergono durante lo sviluppo
- Requisiti consequenziali: derivano dall'individuazione del sistema
- Requisiti di compatibilità con i sistemi già presenti

La gestione dei requisiti serve perché l'ambiente aziendale cambia anche dopo l'installazione e con esso anche i requisiti iniziali cambiano, ci vuole quindi un'evoluzione e una gestione dei requisiti. Ho bisogno quindi di **politiche di tracciabilità** che definiscono, registrano e indicano come conservare la relazione tra i requisiti.

Ci sono tool a supporto della gestione ed elaborazione di grandi quantità di dati, vengono usati per la tracciabilità dei requisiti e del progetto stesso.

L'INGEGNERIA DEI REQUISITI PRODUCE VARI DOCUMENTI



1) **Rapporto di fattibilità:** relativamente semplice e poco costoso (alto livello)
2) **Modelli del sistema:** quale è il sistema da mettere insieme? Posso usare il riuso? In che modo?
3) **Requisiti utente e di sistema:** scrittura vera e propria dei requisiti del sistema
4) ***Documento dei requisiti:*** *conferma da parte del cliente, in caso torno indietro*

Documento dei requisiti

Dichiarazione ufficiale di quello che poi gli sviluppatori dovrebbero fare (COSA non COME)

Deve includere i requisiti utente e di sistema, a un insieme di lettori/utenti molto vario:

- Clienti: specificano i requisiti, leggono il documento per vedere se ci siamo capiti ed eventualmente modificano i requisiti
- Manager: pianificano il costo per lo sviluppo del sistema à quindi devono sapere sia il cosa sia il come
- Ingegnere di sistema: chi davvero svilupperà e testerà o farà manutenzione.

Quindi questo documento è un compromesso che deve essere fruibile a tutti.

Esempio di struttura del documento dei requisiti

Questo documento deve essere strutturato in :

1) PREFAZIONE: dovrebbe definire i lettori attesi del documento. Descrive la cronologia delle versioni, le motivazioni per la creazione di una nuova e un riepilogo delle modifiche apportate in ciascuna.
2) INTRODUZIONE: descrive le necessità del sistema e come interagirà con gli altri sistemi, come sistema si inserisce all'interno degli obbiettivi strategici dell'organizzazione.
3) GLOSSARIO: definisce i termini tecnici usati nel documento (un minimo di contaminazione ci sarà sempre, ma uso solo i termini tecnici spiegati nel glossario).

4) DEFINIZIONE DEI REQUISITI UTENTE: descrivere i servizi forniti agli utenti e i requisiti di sistema non funzionali. Devo usare il linguaggio naturale e diagrammi.
5) ARCHITETTURA DEL SISTEMA: descrizione ad alto livello dell'architettura prevista, suddivisione delle funzioni nei vari moduli. Sottolineo che componenti sto riutilizzando.
6) SPECIFICHE DEI REQUISITI DEL SISTEMA: requisiti funzionali e non funzionali, eventualmente interfacce
7) MODELLI DI SISTEMA: delinea uno o più modelli del sistema mostrando le relazioni del sistema con l'ambiente: modelli a oggetti, diagrammi di flusso, ecc.
8) EVOLUZIONE DEL SISTEMA: scrivo se parto da 0 o riuso e scrivo quali sono le priorità (prima i requisiti DEVO e poi i requisiti DOVREI).
9) APPENDICI: descrizioni tecniche, vincoli hardware.
10) INDICI: indice alfabetico, indice dei diagrammi, indice delle funzioni, ecc.

Capitolo 4 - Modellizzazione del sistema orientata agli oggetti – UML

Un oggetto *è l'unità di informazione che rappresenta un oggetto fisico o astratto e che serve al nostro sistema.* ***SONO SINGOLI ENTI, NON GRUPPI***. Un oggetto ha uno stato con degli attributi e un comportamento con dei metodi, può essere privato o pubblico (information hiding e encapsulation).

Un tipo *è una struttura di tanti oggetti che si assomigliano, rispondono allo stesso insieme di stimoli. È una astrazione per descrivere lo stato e il comportamento. Gli oggetti hanno un tipo.*

Nella modellazione orientata agli oggetti distinguo: tipi atomici e costruttori di tipo (record, set, bag e list).

NB: tipo (prof) = classe (Java) !!!

Es.

- **Tipo per il prof**

TYPE DOCENTE:

RECORD(CF: STRING, NOME: STRING, COGNOME: STRING, STUDENTI: SET(STUDENTE)
)

- **Classe in Java**

```
Class Docente{
    String cf;
    String nome;
    String cognome;
    TreeSet<Studente> studenti = new TreeSet<>();
}
```

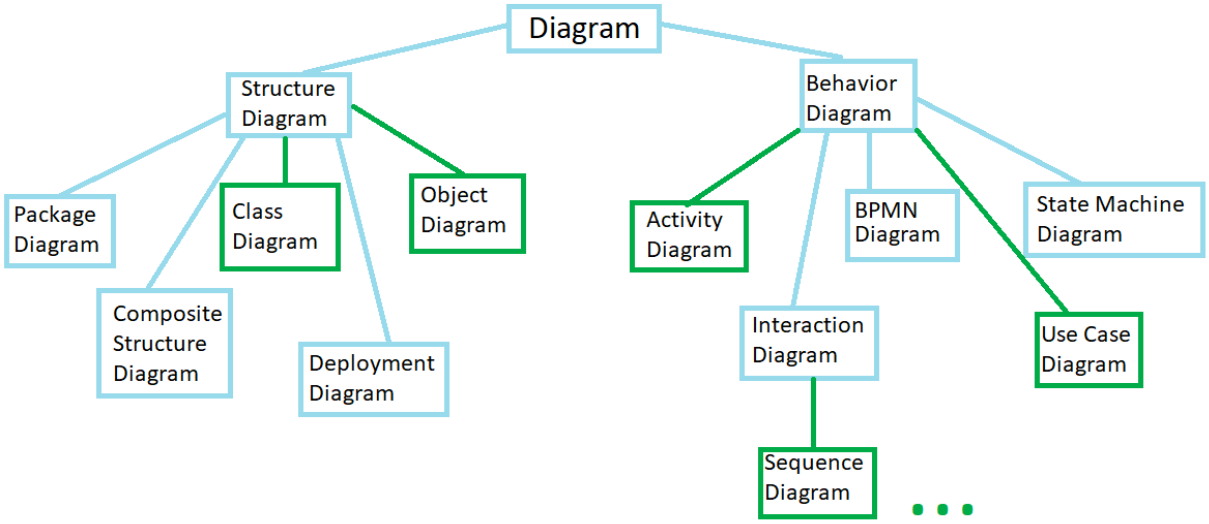
NB: Il tipo visto da Combi è un tipo di dato strutturato comparabile a una struct in C e a una classe in Java, racconta la struttura dello stato e l'assegnatura (signature??? Firma?) dei metodi, interfaccia dell' utente.

Gli oggetti vengono raccolti e implementati in classi quindi: la classe non è come quella di java, ma specifica l'implementazione dei metodi, implementazione di un tipo, in più gestisce l'interazione fra oggetti dello stesso tipo.

Come tecnica di progettazione e documentazione abbiamo usato **UML** che uno strumento per la definizione di diagrammi. I diagrammi che abbiamo visto a lezione sono:

- use case diagram
- class diagram
- activity diagram
- sequence diagram

Vari tipi di Diagram



BPMN (Business Process, Model & Notation) : racconta e descrive i processi organizzativi.

Behavior Diagram o diagramma di comportamento : racconta l'evoluzione.

Deployment Diagram: spiega come verrà eseguita l'installazione su macchine reali.

Use case diagram

Sono usati per la rappresentazione: dei requisiti e della loro analisi. Non ha una sequenza di attività nel tempo.

ATTORI: rappresentati con degli omini, ci possono essere delle **gerarchie fra loro**. Non sono necessariamente umani (es. attori esterni posso essere altri sistemi software). Sono coloro che fanno partire i casi d'uso o che vengono coinvolti nell'esecuzione dei casi d'uso. Non identificano singoli specifici utenti ma fanno riferimento ad **un ruolo** che può appartenere a più persone fisiche o software.

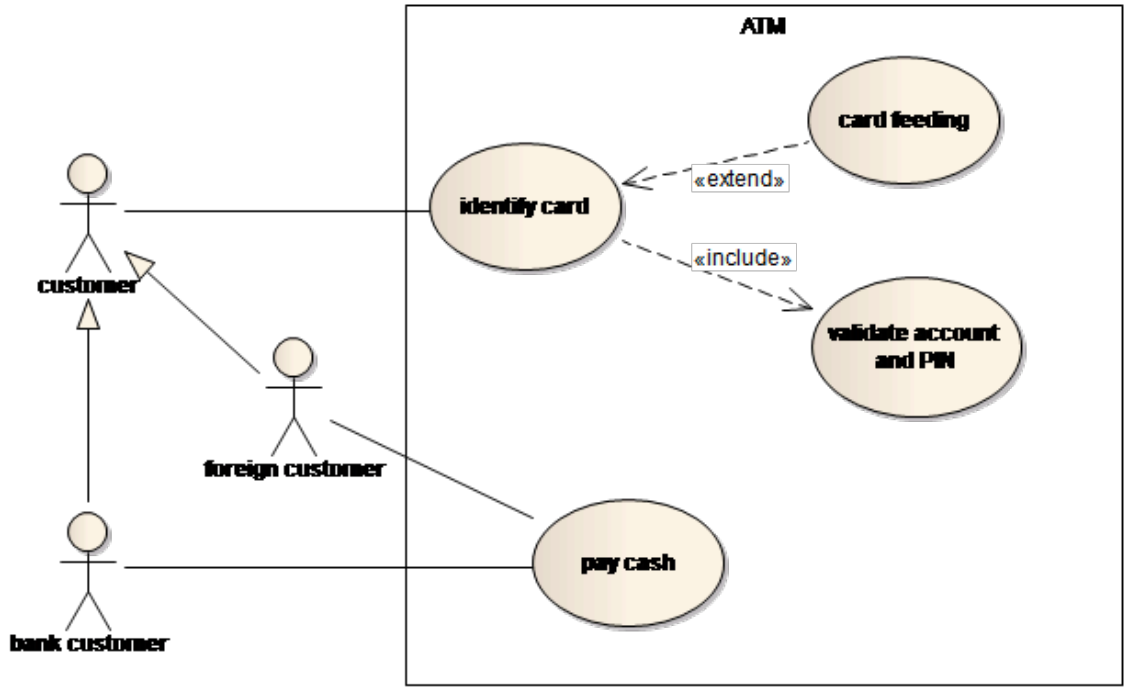
DIPENDENZE TRA I CASI D'USO

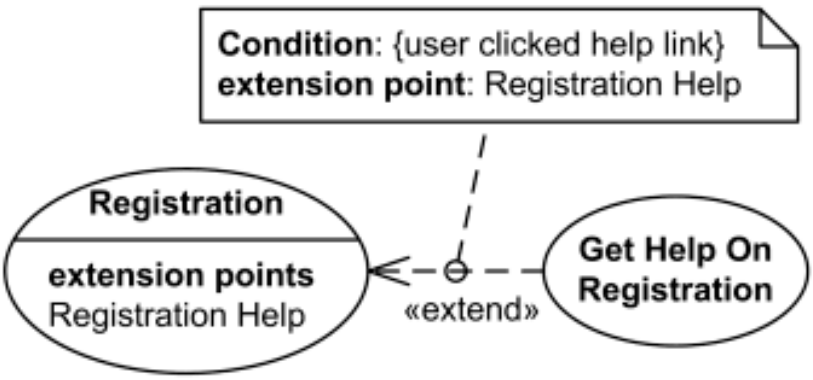
<<Extends>> : comportamento opzionale. Uno use case è una parte eccezionale/opzionale all'interno di un altro use case (il sistema probabilmente ha un if e quindi svolge questa azione a volte).

<<Include>> : stereotipo. Etichette con significato predefinito all'interno di un diagramma UML ("prevede sempre"). La freccia va verso l'incluso. Si usa per implementazioni obbligatorie (cose che il sistema DEVE fare per forza).

Extension Point: ulteriore informazione aggiunta agli extends (vedi es.)

Specifica degli use case: esiste della documentazione alfanumerica che racconta quelle **parti temporali** che non si riesce a rappresentare graficamente negli use case.





Class diagram

CLASSE: il nome della classe è l'unica cosa obbligatoria e deve essere scritto in UpperCamelCase. Gli attributi e i metodi sono opzionali e basta il loro nome, sono scritti in lowerCamelCase. Possono esserci degli ornamenti:

- La **molteplicità (cardinalità)** : può essere ad es. [0..1] oppure [0..*] oppure [5..5].

- Il **valore di default**

- La **visibilità**: + pubblica, - privata, # protected e ~ package.

Per gli **attributi** gli ornamenti sono così organizzati:

<visibilità> <nomeAttributo> :<tipo> <molteplicità/cardinalità>=<valore Default> <altro>

es. +nomeAttributo: ClasseJava [0..1] = 0

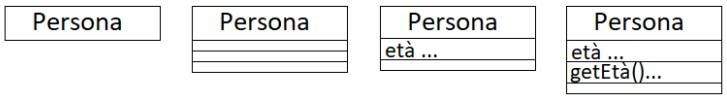
Per i **metodi** gli ornamenti sono così organizzati:

<visibilità> <nome Metodo> ({<nome Parametro: tipo parametro} {nomePar2: tipoPar2,..>})

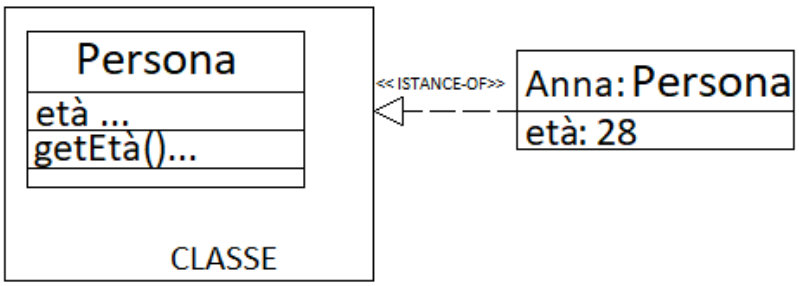
<tipoParametroRestituito>

es. +getOggetto(nome:String, cognome:String) Object

Esempi di rappresentazione di una classe (per il prof rimane comunque un tipo)



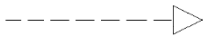
Esempio di rappresentazione di un'istanza della classe



Operazioni tra classi

1) Generalizzazione

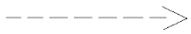
2) Realizzazione



es. una classe realizza un'interfaccia.

3) Associazione

4) Dipendenze



5) Aggregazione: tutte e due le classi coinvolte hanno spessa dignità.

6) Composizione: le parti dipendono da tutto e non possono esistere senza il tutto.

NAVIGABILITÀ

La navigabilità è la direzione dell'operazione. Viene indicata con le frecce e le croci ma mettere sempre questi simboli appesantisce il diagramma.

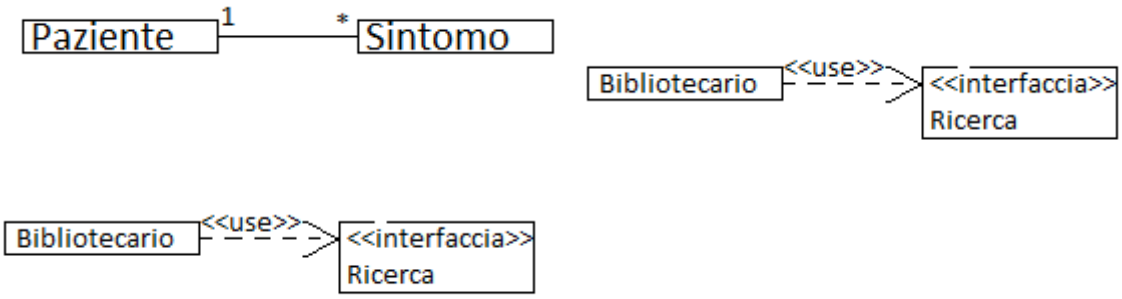
Si è arrivati a una convenzione:

- Le croci non si usano (dovrebbero indicare che la direzione inversa di una freccia non funziona).

- La mancanza sia della freccia sia della croce indica "navigabilità non specificata" cioè la direzione è in entrambi i sensi.

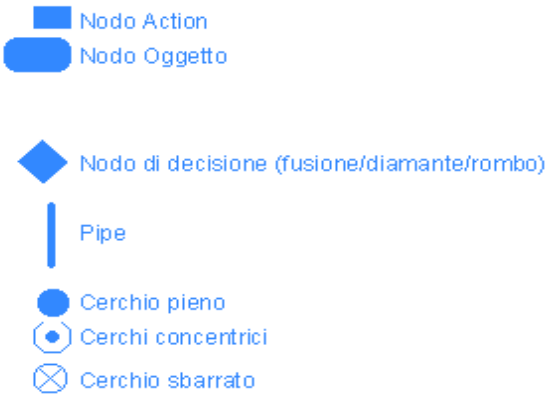
- Per specificare la direzione uso solo le frecce.

Esempi:



Activity Diagram

Va a rappresentare gli oggetti di nostro interesse (attività legate al tempo).



Viene usato a supporto: degli use case e delle attività di funzione (es. flusso di operazioni di un caso d'uso, i passi principali di un algoritmo, i passi principali di un utente per interagire con l'interfaccia grafica).

Componenti grafici

Nodo Action (angoli smussati): è la minima unità di comportamento, rappresenta una singola attività o azione.

Nodo Oggetto: unità di informazione, rappresentano gli oggetti software usati come input/output delle attività.

Nodi di controllo

Nodo di decisione e Fusione (Diamante o rombo): rappresenta una scelta (sempre a coppie)

Pipe: rappresenta l'esecuzione in contemporanea di più attività (sempre a coppie).

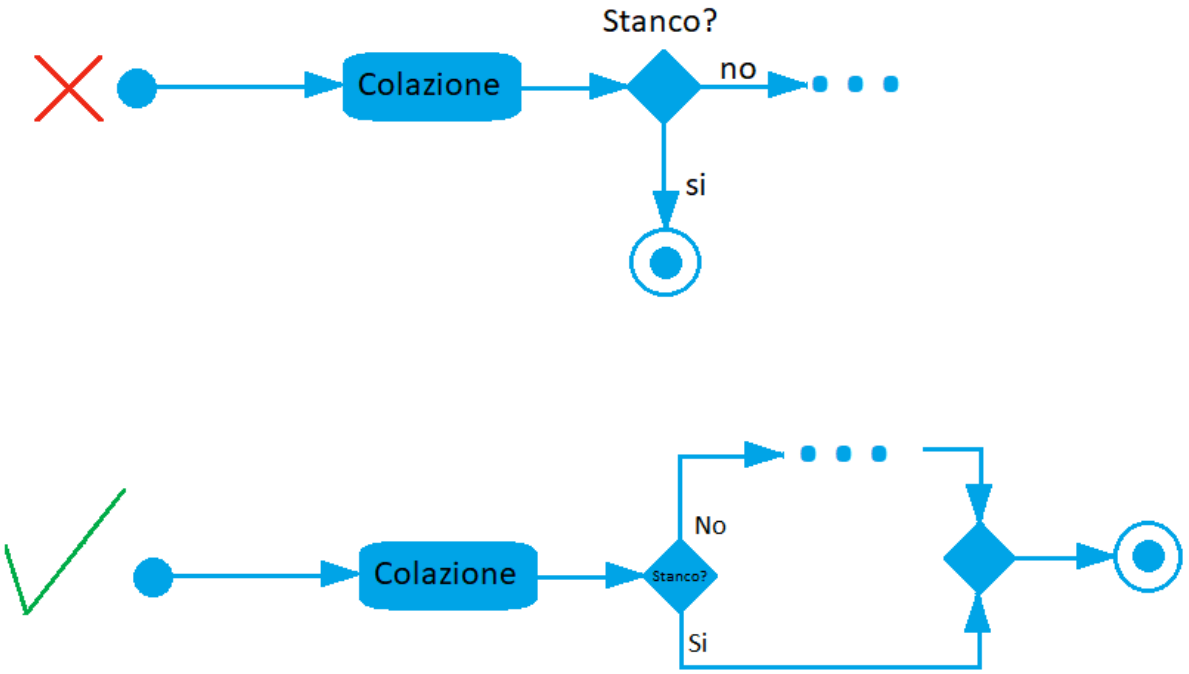
Cerchio Pieno: dove inizia l'azione del diagramma.

Cerchi concentrici: dove finisce tutta l'esecuzione.

Cerchio sbarrato (a x): dove si ferma/blocca l'azione.

NB: Semantica a token (gettone): se ho un gettone all'ingresso posso fare l'attività richiesta, se non lo ho niente azione. Quando consumo un gettone ne produco un altro in uscita.

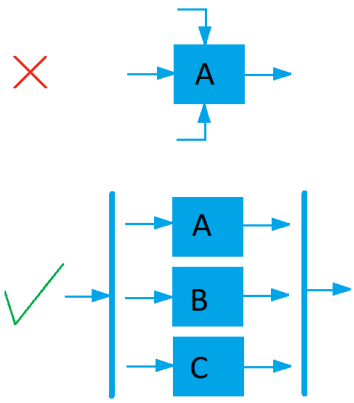
Esempio:



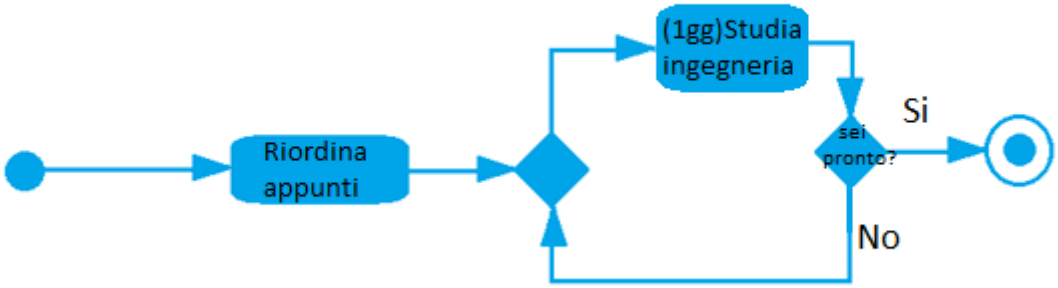
Il primo esempio è sbagliato perché il rombo è da solo, non è a coppia, e la semantica del token(gettone) non funziona. Il secondo esempio invece è il corretto modo per questo diagramma.

AMBIGUITÀ:

Cerchiamo di evitare ambiguità, nell'esempio qui sotto il primo diagramma non ha un diamante o una pipe e non si capisce quale dei due comportamenti produce quell'output. Uno dei due modi per correggere l'errore è mettere una pipe (potremmo volere anche un altro comportamento e in questo caso mettiamo un diamante). La pipe in ingresso è la FORK e la pipe in uscita è la JOIN.



Esempio di ciclo



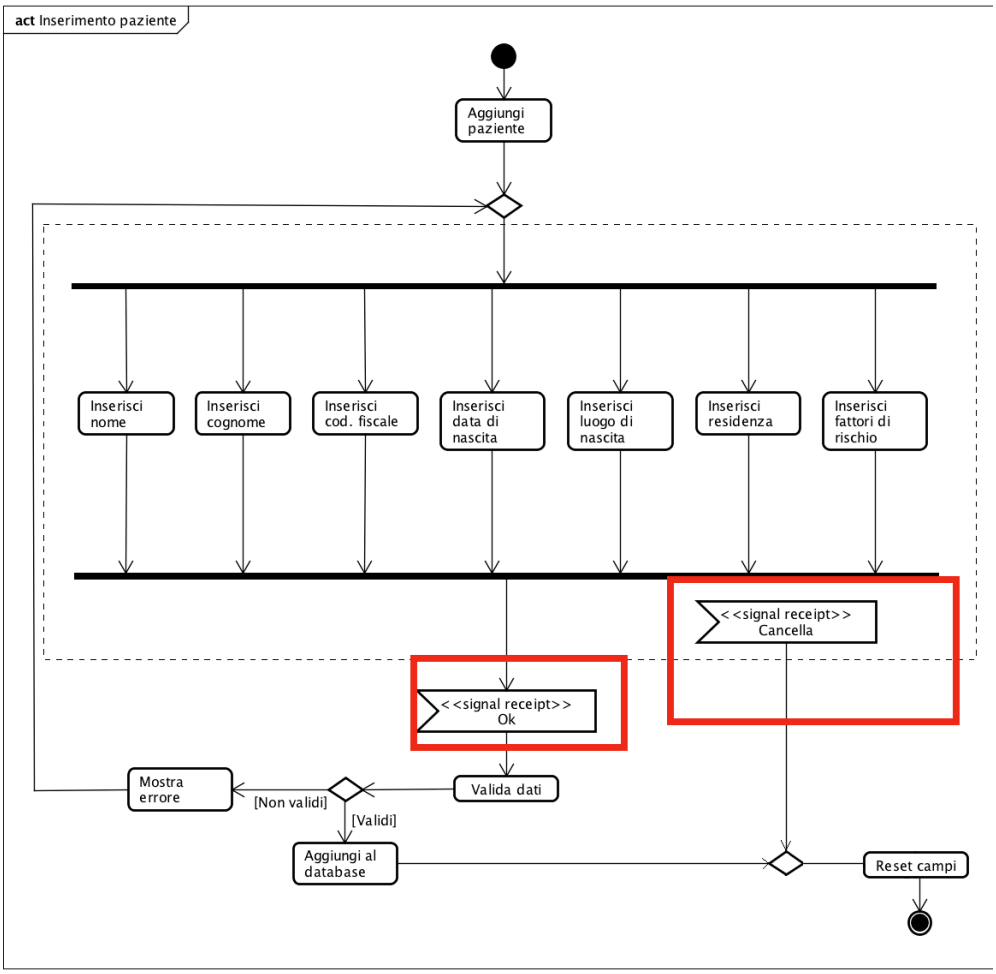
Segnali e eventi

Simbolo per mandare un segnale, asincrono e non broccante.

 Simbolo per ricevere un evento, attivo e bloccante.

Simbolo per accettare, evento temporale.

ESEMPIO:



Sequence diagram

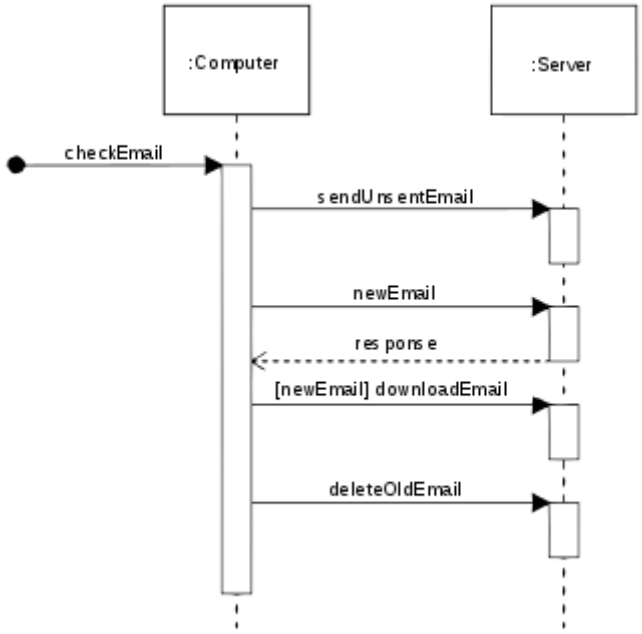
Servono per gli scampi di messaggi fra entità. Le entità sono attori e istanze di classi.

TIPI DI FRECCETTE

1) Freccia sincrona, l'entità che la manda rimane in attesa di risposta.

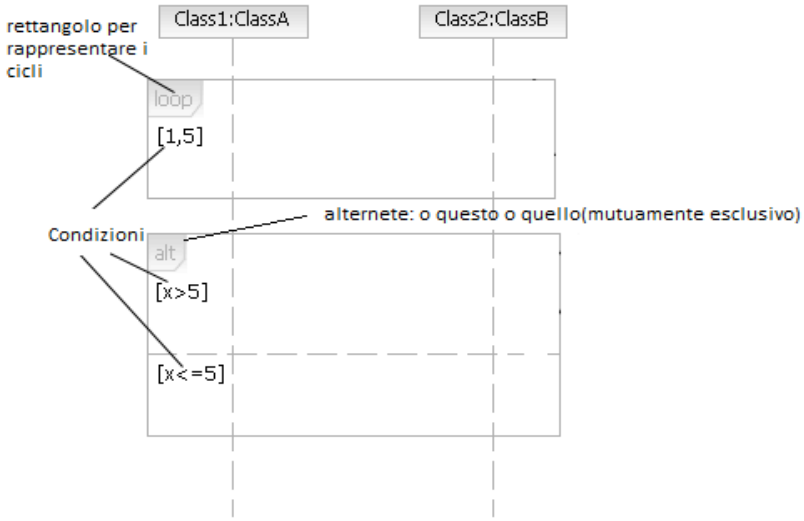
2) Freccia asincrona, non attendo risposta.

ESEMPIO



FRAMMENTO COMBINATO: un rettangolo che comprende tutti gli attori coinvolti e svolge una certa azione.

1) ALT: alternate, o questo o quello, mutuamente esclusivo, un if else
2) OPT: if/then, non ci sono linee tratteggiate, non c'è l'else
3) LOOP: for e while
4) BREAK
5) REF: reference a un altro sequence diagram che non ripeto.



Capitolo 5 - Progettazione architetturale

La progettazione architetturale riguarda la comprensione di come dovrebbe essere organizzato un sistema software e la progettazione della struttura generale di quel sistema.

La progettazione architetturale è il collegamento tra la vera e propria progettazione e l'ingegneria dei requisiti, in quanto identifica i principali componenti strutturali in un sistema e le relazioni fra loro.

L'output del processo di progettazione architetturale è un modello architetturale che descrive come è il sistema organizzato cioè un insieme di componenti comunicanti.

NB: facciamo quindi molto **riuso di modelli** già esistenti che sono già testati e collaudati, così da non partire da zero.

Si può dividere in:

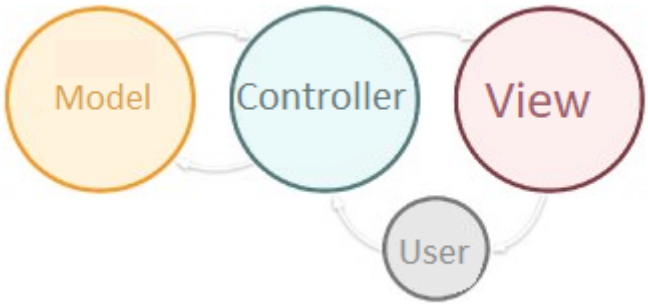
1) IN PICCOLO: guardo l'architettura dei singoli programmi cioè come un singolo programma è scomposto in componenti (in moduli/sottoparti). In pratica facciamo affidamento a PATTERN ARCHITETTURALI.

2) IN GRANDE: guardo l'architettura di sistemi aziendali complessi che includono altri sistemi e programmi. Essi sono distribuiti su diversi computer, possono essere di proprietà di diverse società. Qui facciamo affidamento a DESIGN PATTERN cioè alla progettazione dei singoli gruppi di classe e delle singole classi.

Pattern Architetturali

Un modello architetturale è una descrizione stilizzata di progettazione che è stata provata e testata. Guardo com'è scomposto un programma nelle sue parti. Ogni modello ha dei pregi e dei difetti, è bene sapere quando sono utili e quando non lo sono.

Pattern MVC – Model View Controller



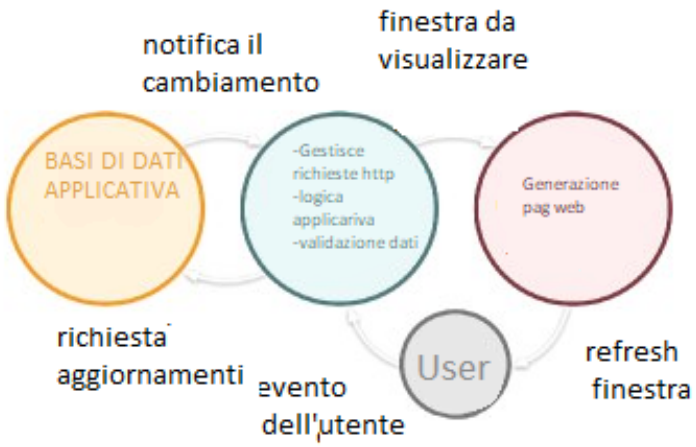
Questo modello separa il programma in tre componenti:

- **Model:** gestisce la base di dati e le sue operazioni (può essere anche non una base di dati vera e propria).
- **View:** gestisce l'interfaccia utente e l'interazione con l'utente.
- **Controller:** implementa la logica applicativa, gestisce le azioni dell'utente e le elabora mandandole a View. Collega e gestisce View e Model.

Lo uso quando: ci sono più modi per visualizzare e interagire con i dati. Usato anche quando i requisiti futuri per l'interazione e la presentazione dei dati sono sconosciuti.

Vantaggi: posso cambiare View e Model senza dover cambiare le altre due parti.

Svantaggi: ho codice aggiuntivo e più complesso anche quando Model e View sono semplici.



Pattern a strati (o layered)

Utilizzato per organizzare il sistema in un set di livelli, ciascuno dei quali fornisce una serie di servizi. Supporta lo sviluppo incrementale dei sottosistemi perchè quando cambia un livello solo lo strato adiacente è interessato. Tipico modello delle architetture di rete (studiate reti, CAPRE!).

Lo uso quando: costruisco nuove strutture su sistemi esistenti, quando lo sviluppo si sviluppa su più squadre con responsabilità di ogni squadra per un livello di funzionalità o quando c'è un requisito per la sicurezza a più livelli.

Vantaggi: permette la sostituzione di interi livelli senza toccare gli altri. Le strutture ridondanti (ad es. L'autenticazione) possono essere in ogni strato per aumentare l'affidabilità del sistema.

Svantaggi: il modello risulta spesso artificiale, la netta separazione tra gli strati è spesso difficile e uno strato di alto livello potrebbe dover interagire direttamente con strati di livello inferiore anziché attraverso il livello immediatamente sotto di esso. Diminuiscono le prestazioni perchè una richiesta deve essere elaborata ad ogni livello.

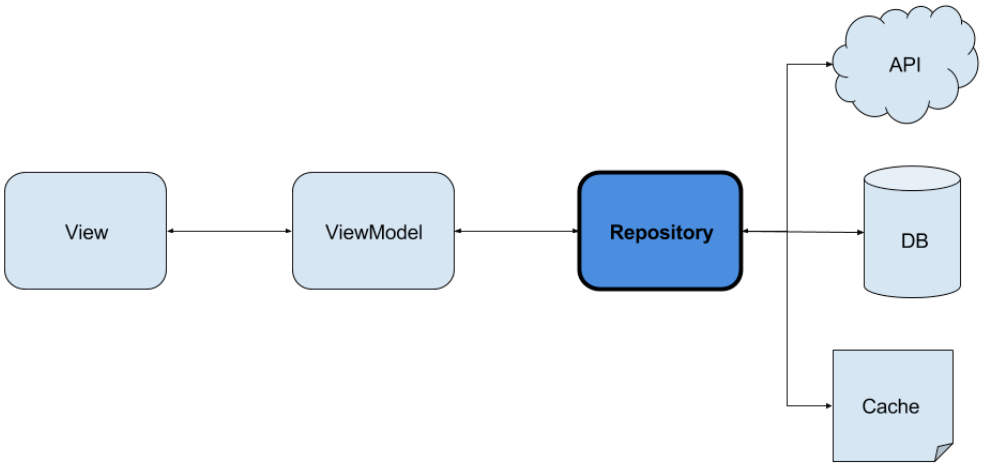
Pattern repository

Posso avere dei sottosistemi che devono scambiare dati per compiere le loro operazioni. Posso conservare i dati condivisi in un database o repository centrale, i dati possono essere accessibili da tutti i sottosistemi. Oppure ogni sottosistema mantiene il proprio database e passa i dati esplicitamente ad altri sottosistemi. Quando devono essere condivise grandi quantità di dati, il pattern repository è più comunemente usato perchè è un meccanismo efficiente di condivisione dei dati.

Lo uso quando: ho un sistema con un grande volume di informazioni che deve essere tenuto per tanto tempo.

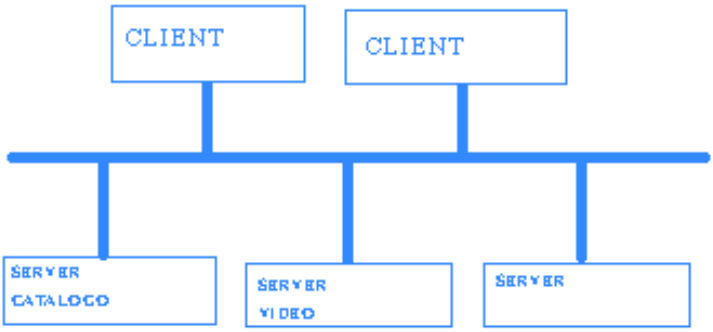
Vantaggi: i sottosistemi sono indipendenti fra loro e possono non sapere dell'esistenza degli altri. I cambiamenti fatti da un sottosistema vengono propagati velocemente, c'è consistenza essendo tutto in un solo posto.

Svantaggi: tutti i sottosistemi dipendono dal repository, una falla nel repository dà problemi a tutto il sistema. La comunicazione con il repository può essere gestita male. La distribuzione del repository su vari computer può essere difficile.



Pattern client server

Modello di sistema distribuito, i dati e l'elaborazione è distribuita su una gamma di componenti (server standalone). I client chiamano i server per questi servizi, la rete consente ai client di accedere ai server. Può essere implementato su un singolo server.Possiamo avere un'architettura client server nella rete e un'architettura MVC nel server.



Lo uso quando: i dati in un database condiviso devono essere accessibili da una vasta gamma di posizioni. Potrebbe anche essere utilizzato quando il carico su un sistema è variabile poichè i server possono essere replicati.

Vantaggi: i server possono essere distribuiti attraverso una rete. Una funzionalità generale (ad esempio una stampa servizio) può essere disponibile per tutti i clienti.

Svantaggi: ogni servizio è un singolo punto di errore, suscettibile ad attacchi Dos o a errori del server. Le prestazioni potrebbero essere imprevedibili perché dipende dalla rete e dal sistema. Possono esserci problemi di gestione se i server sono di proprietà di diverse organizzazioni.

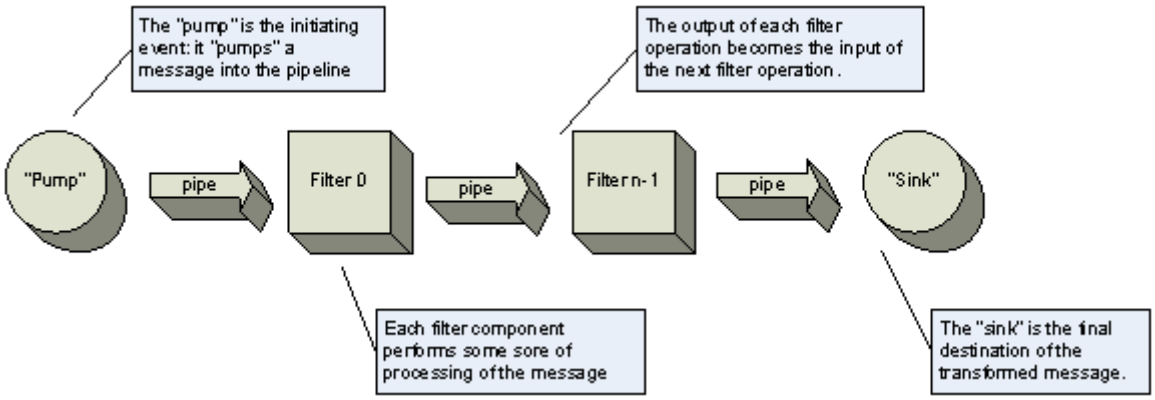
Pattern Pipe & filter

Vogliamo processare dei dati, ogni componente che processa (filter) è discreto, i dati passano da un componente all'altro (pipe). I componenti elaborano i loro input per produrre output. Quando le elaborazioni sono sequenziali, questo modello diventa un batch sequential che è ampiamente utilizzato nell'elaborazione dei dati di sistemi.

Lo uso quando: ampiamente utilizzato in applicazioni di elaborazioni di dati (sia batch che transaction-based), quando abbiamo un'elaborazione separata in blocchi (stages). Distinguiamo le varie elaborazioni per ottenere una maggiore manutenibilità del sistema.

Vantaggi: è facile da capire, supporta il riuso di trasformazioni. Questo stile combacia con tanti processi business. L'evoluzione è semplicemente aggiungere trasformazioni. Può essere implementato sia come sequenziale che in parallelo (conccorrenziale?).

Svantaggi: il formato per il trasferimento di dati va concordato, quindi i formati di input e output vanno rispettati. Questo può portare all'impossibilità del riutilizzo di certe trasformazioni. Non adatto per i sistemi interattivi.



Design Pattern

Progettazione dei singoli gruppi di classi e delle singole classi.

Una progettazione Object-oriented di processi (di sviluppo) spazia un vario numero di modelli di sistema. Questi modelli richiedono molti sforzi sia per lo sviluppo che per la manutenzione, per i piccoli sistemi questo potrebbe non essere conveniente, tuttavia, per sistemi di grandi dimensioni sono un importante meccanismo di comunicazione. Le attività comuni nel processo di sviluppo sono:

- **definire il contesto e le modalità di utilizzo del sistema:** la comprensione delle relazioni tra il software e il suo ambiente esterno è essenziale, consente anche di stabilire il confini del sistema.

- **progettare l'architettura del sistema:** una volta che abbiamo capito l'interazioni tra il sistema e il suo l'ambiente, possiamo identificare i componenti principali che lo compongono e le loro interazioni, così possiamo organizzarli in un modello architetturale (PATTERN ARCHITETTURALI).

- **identificare le principali classi:** identificare le classi per i vari pattern non è così facile, è un processo iterativo e probabilmente la prima volta sbagli.

- **sviluppare i modelli di design:** guardiamo molto al comportamento del sistema per identificare i partecipanti di un possibile pattern (DESIGN PATTERN). Possiamo usare uno scenario in cui classi, attributi e metodi vengono identificati.

- **specifica delle object interfaces**: gli oggetti(le classi) possono avere diverse interfacce, esse devono essere specificate in modo che le classi e gli altri componenti possono essere progettati in parallelo.

DESIGN PATTERN: un design pattern è un modo per riutilizzare la conoscenza astratta su un problema e la sua soluzione (sono buone pratiche, buoni progetti e pregressa esperienza), il modello contiene la descrizione del problema e l'essenza della sua soluzione. Dovrebbero essere sufficientemente astratti da essere riutilizzati in diversi impostazioni, di solito si avvalgono di caratteristiche object-oriented come ereditarietà e polimorfismo.

MODELLI DI DESIGN PATTERN:

- **Strutturali:** descrivono la struttura statica del sistema in termini di classi di oggetti e relazioni.

- **Dinamici:** descrivono le interazioni dinamiche tra oggetti.

Singleton (Creazionale)

Nome e descrizione: singolo, usato per assicurare che una classe abbia una sola istanza ed un unico punto di accesso globale.

Descrizione del problema/quando viene usato: c'è la necessità di garantire l'esistenza di un unico oggetto di una classe (es. in un sistema ci possono essere tante stampanti ma solo una classe PrintSpooler).

Soluzione: le classi Singleton vengono progettate con i **costruttori privati** per evitare la possibilità di istanziare un numero arbitrario di oggetti della stessa. Esiste un **metodo statico** con la responsabilità di assicurare che nessuna altra istanza venga creata oltre la prima, restituendo contemporaneamente un riferimento all'unica esistente. La classe contiene poi tutti i metodi, le proprietà e gli attributi tipici dell'astrazione per cui è stata concepita.

CODICE:

```
public class Singleton {
        // istanza allocata in modo pigro
        private static Singleton instance = null;
        // specifica delle proprieta' caratteristiche dell'oggetto ...
        // costruttore privato, proibisce ad altri di creare oggetti
        private Singleton() {}
        // metodo statico
        public static Singleton getInstance() {
                if ( instance == null )
                instance = new Singleton();
                return instance;
        }
}
```

Observer Pattern (Comportamentale)

Nome e descrizione: osservatore, separa la visualizzazione dello stato dell'oggetto dall'oggetto stesso, consente di fornire vari display. Quando lo stato dell'oggetto cambia, tutti i display sono automaticamente notificati e aggiornati per riflettere il modificare.

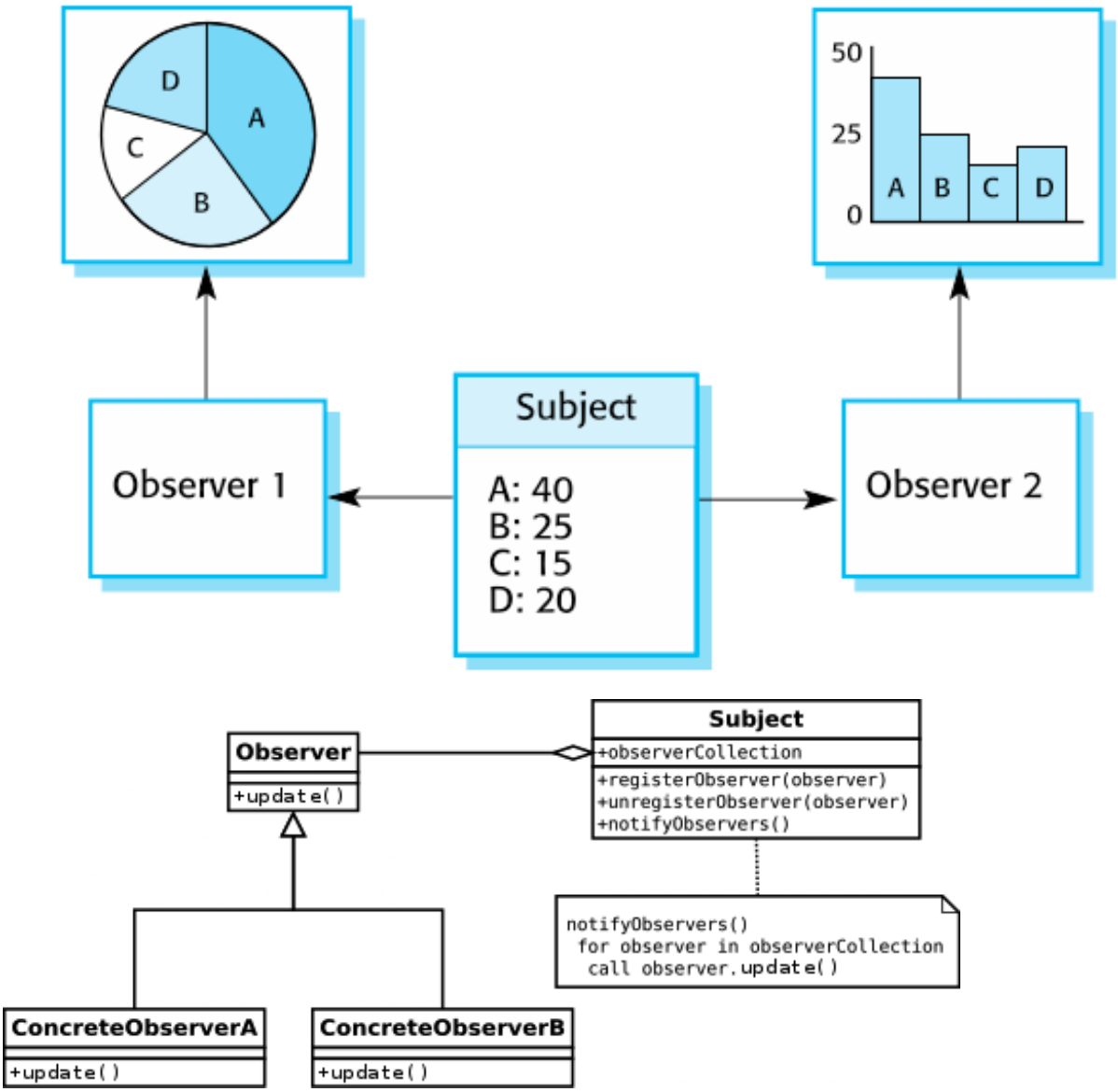
Descrizione del problema/quando viene usato: utilizzato quando sono necessari più display di stato e l'oggetto può non conoscere i formati dei display utilizzati, ad esempio un display grafico e un display tabellare. Quando lo stato cambia, tutti i display devono essere aggiornati. (es. interfaccia utente Swing).

Soluzione:

Ho bisogno di due interfacce: Subject (soggetto osservato) e Observer. E di due classi concrete: ConcreteSubject e ConcreteObject, che ereditano gli attributi degli oggetti astratti correlati. Gli oggetti astratti, **Subject e Observer**, includono operazioni generali che sono applicabili in tutte le situazioni.

ConcreteSubject: mantiene lo stato da visualizzare, eredita le operazioni da Subject consentendogli di aggiungere add() e rimuovere remove() gli Observer (ciascun Observer corrisponde a un display) e quando lo stato cambia manda una notifica notify() a tutti gli Observer.

ConcreteObserver: mantiene una copia dello stato di ConcreteSubject e definisce il metodo update() di Observer, questo metodo gli consente rimanere aggiornato. Ogni volta che lo stato viene aggiornato, automaticamente aggiorna il display.



NB: Java fornisce già implementate le classi per realizzare il pattern Observer.

Vantaggi: utile quando i requisiti possono cambiare nel tempo e si richiede un display diverso dal precedente.

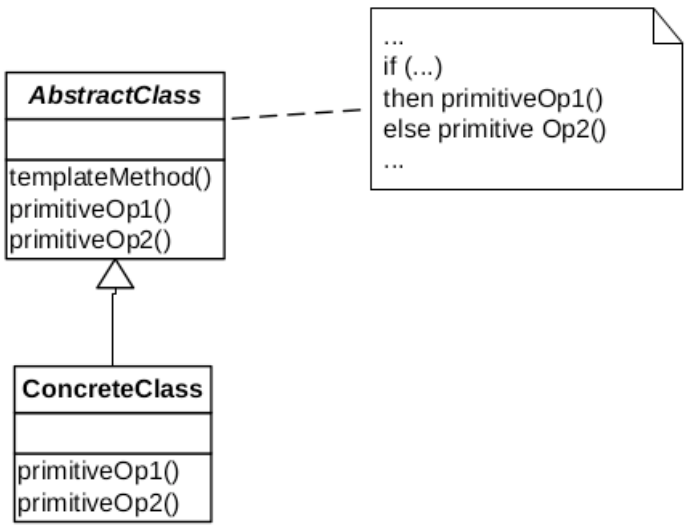
Svantaggi: le ottimizzazioni per migliorare le prestazioni del display non sono pratiche.

CODICE:

```
public interface Subject {
        void add(Observer anObserver);
        void remove(Observer anObserver);
        void notify();
}
public interface Observer {
        void update();
}
public class ConcreteSubject implements Subject {
        private int state;
        private LinkedList observers;
        public ConcreteSubject(int initialState){
                state = initialState;
        }
        public int getState() {
                return state;
        }
        public void setState(int aState) {
                state = aState;
        }
        public void Notify() {
                Iterator iter=observers.iterator();
                Observer   obs;
                while(iter.hasNext()) {
                        obs=(Observer)iter.next();
                        obs.update();
                }
        }
        public void add(Observer anObserver) {
                observers.add(anObserver);
        }
        public void remove(Observer anObserver) {
                observers.remove(anObserver);
        }}
public class ConcreteObserver implements Observer {
        private ConcreteSubject subject;
        public ConcreteObserver(ConcreteSubject aSubject){
                subject = aSubject;
                subject.add(this);
        }
        public void Update(){
                System.out.println("Io sono l'observer " +
        this +
                " ed il nuovo stato del mio subject e' " +
                subject.getState());
        }
}
```

Template pattern (Comportamentale)

Nome e descrizione: modello/sagoma, voglio definire la struttura (template) di un algoritmo all'interno di un metodo final (quindi non sovrascrivibile, si chiamerà template) in una classe astratta, delegando alcuni passi dell'algoritmo alle sottoclassi concrete. Le sottoclassi ridefiniscono alcuni passi dell'algoritmo senza dover implementare di nuovo la struttura(template) dell'algoritmo stesso (Operazioni hook). Principio Hollywood: "don't call me, I'll call you" cioè è la classe padre che chiama le operazioni ridefinite nei figli e non viceversa.



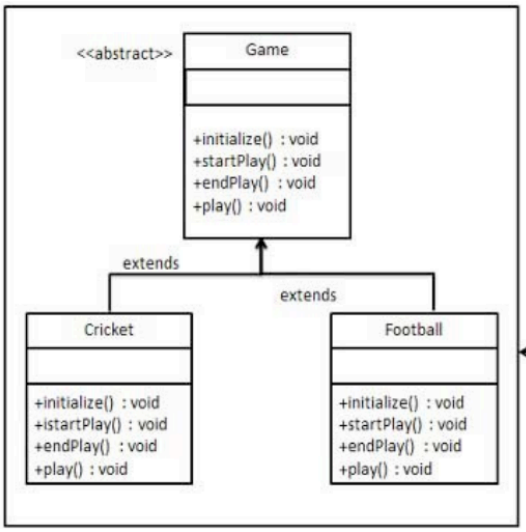
Descrizione del problema/quando viene usato: quando voglio che la super classe sia più forte, avendo un metodo che obblighi le sottoclassi a rispettare un certo modello.

Soluzione: creiamo una **classe astratta** che ha un **metodo final e metodi astratti**. All'interno del metodo final chiamo i metodi astratti, si chiama template perchè essendo final non può essere sovrascritto e fa da modello alle sottoclassi. Le sottoclassi concrete che creo estendono quella astratta e fanno l'override dei suoi metodi astratti.

CODICE:public abstract class **AbstractClass** {

```
public abstract class AbstractClass {
        abstract void primitiveOp1();
        abstract void primitiveOp2();
        //template method
        public final void templateMethod(){
                primitiveOp1();
                primitiveOp2();
        }
}
```

```
public class ConcreteClass extends AbstractClass {
        @Override
        void primitiveOp1() { … }
        @Override
        void primitiveOp2() { … }
}
```



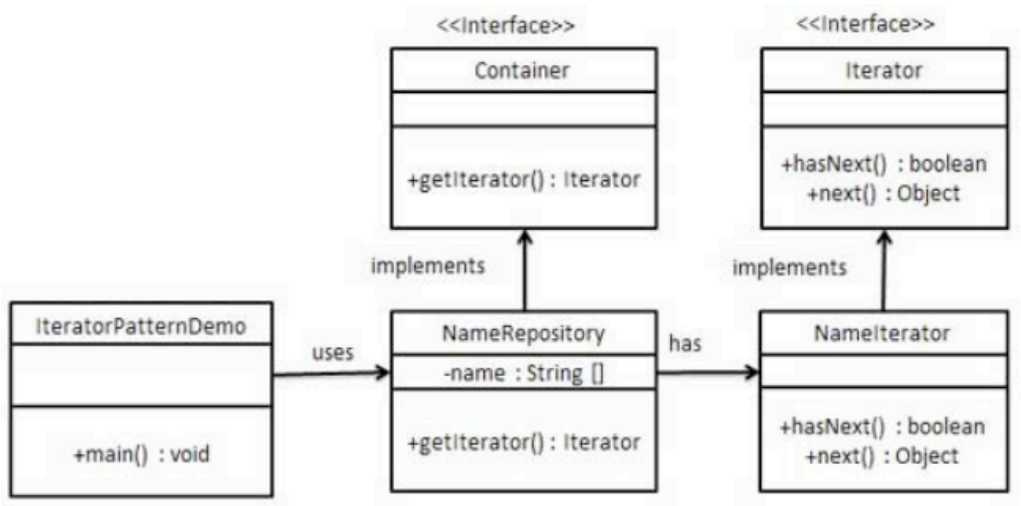
Vantaggi: puoi aggiungere sottoclassi senza preoccuparti del modello.

Svantaggi: se devi cambiare la classe astratta fai fatica.

Iterator pattern (Comportamentale)

Nome e descrizione: iteratore, utilizzato per ottenere un modo per accedere agli elementi di una raccolta in sequenza, senza alcun bisogno di conoscere la sua rappresentazione sottostante.

Descrizione del problema/quando viene usato: ho un set di oggetti non predefinito, devo esaminarli uno ad uno sequenzialmente (es. in un negozio di cibo voglio cercare i cd per nome, iterator per far vedere a video).



Soluzione: creiamo un'**interfaccia Iterator** che narra il metodo di navigazione e un'**interfaccia Container** che ritorna l'iteratore. Le classi concrete dell'interfaccia Container sono responsabili di implementare l'interfaccia Iterator (hanno un **inner class** che implementa Iterator) e di usarlo.

CODICE:public class **ConcreteContainer** implements Container {

public class **ConcreteContainer** implements Container {

```
public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};
@Override
public Iterator getIterator() {
        return new NameIterator();
}
// inner class che implementa Iterator
private class ConcreteIterator implements Iterator {
        int index;
        @Override
        public boolean hasNext() {
                if(index < names.length){
                        return true;
                }
                return false;
        }
        @Override
        public Object next() {
                if(this.hasNext()){
                        return names[index++];
                }
                return null;
        }
}
}
```

```
public interface Iterator {
        public boolean hasNext();
        public Object next();
}
public interface Container {
        public Iterator getIterator();
}
```

Factory pattern (Creazionale)

Nome e descrizione: offre uno dei modi migliori per creare un oggetto, creiamo l'oggetto senza esporre all'esterno com'è venuta la creazione.

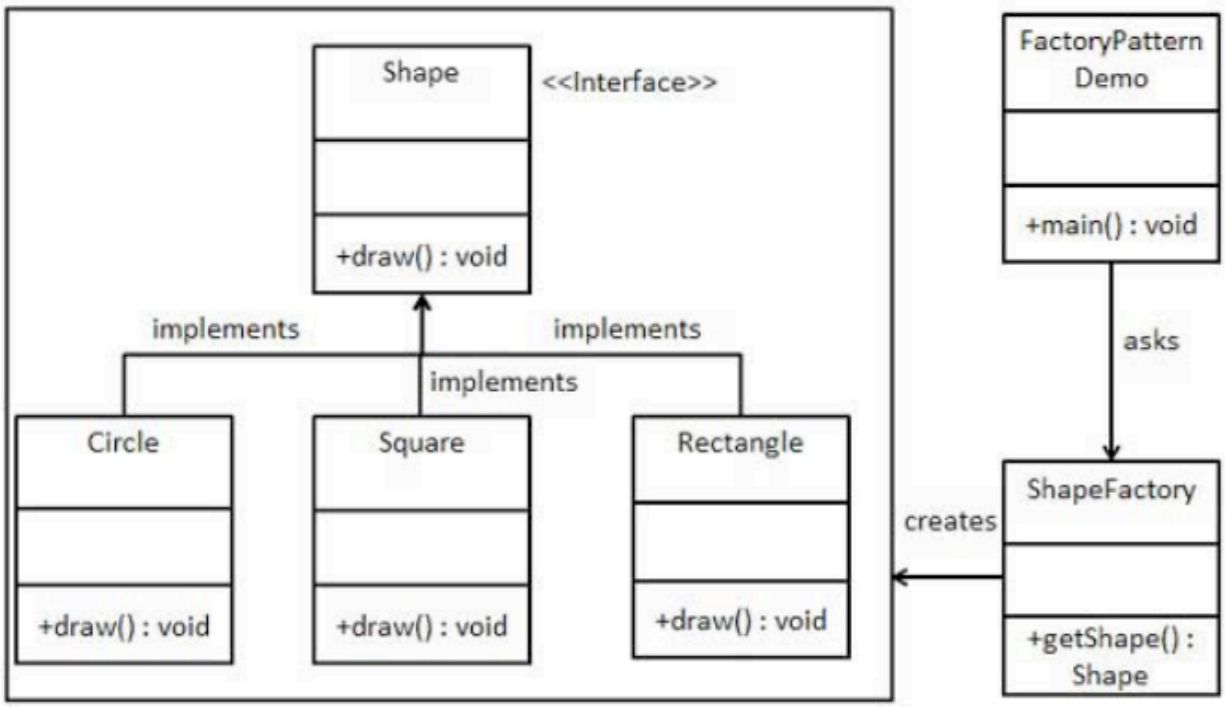
Descrizione del problema/quando viene usato: voglio la creazione di oggetti ad alto livello ma non mi interessa la gerarchia interna. Questo pattern ti offre un servizio nascondendotelo, gli oggetti all'esterno sono indistinguibili.

Soluzione: creiamo: un'interfaccia, delle classi concrete che la implementano e una classe Factory che ritorna con il metodo getShape(String shapeType) l'oggetto creato con la sottoclasse che vuoi tu.

CODICE:public class **Factory** {

```
public class Factory {
// questo metodo ti ritorno l'oggetto creato con la sottoclasse che vuoi tu
        public Shape getShape(String shapeType){
                if(shapeType == null){
                        return null;
                }
                if(shapeType.equalsIgnoreCase("shape1")){
                        return new Shape1();
                } else if(shapeType.equalsIgnoreCase("shape2")){
                        return new Shape2();
                } else if(shapeType.equalsIgnoreCase("shape3")){
                        return new Shape3();
                }return null;}}
```

```
public interface Shape {
        void op();
}
public class Shape1 implements Shape {
        @Override
        public void op() { ... }
}
```



Abstrac factory pattern (Creazionale)

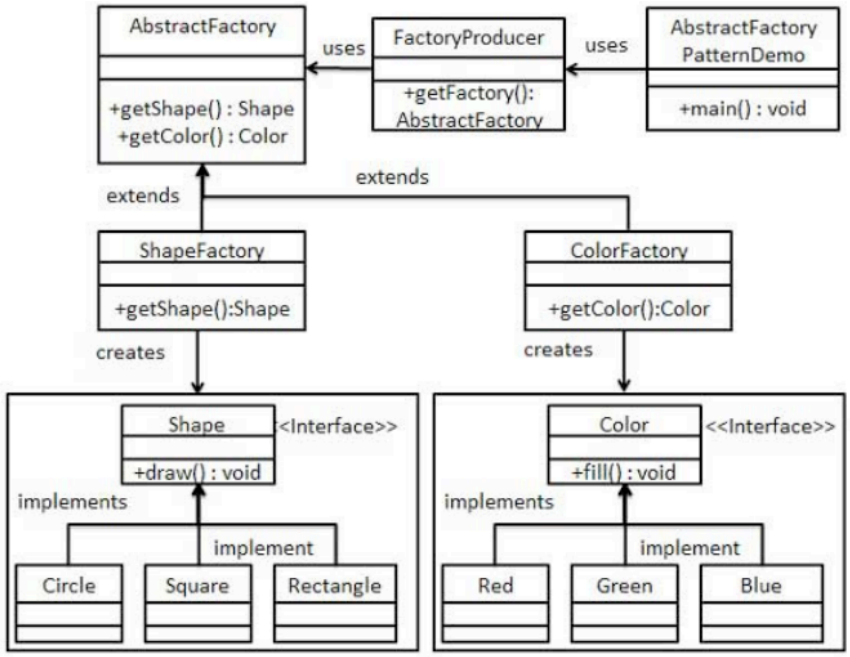
Nome e descrizione: una super-factory che crea altre factory, cioè factory di factory. Un'interfaccia è responsabile della creazione di una factory di oggetti correlati senza specificare esplicitamente le loro classi. Ogni factory generata può dare gli oggetti secondo il factory pattern.

Descrizione del problema/quando viene usato:

Soluzione: Creiamo: due interfacce Shape e Color, delle classi concrete che le implementano e una classe astratta AbstractFactory. Le classi factory ShapeFactory e ColorFactory estendono AbstractFactory.

La classe FactoryProducer crea ???.

AbstractFactoryPatternDemo, la nostra classe demo utilizza FactoryProducer per ottenere un AbstractFactory oggetto. Passerà le informazioni (CIRCLE / RECTANGLE / SQUARE per Shape) a AbstractFactory per ottenere il tipo di oggetto di cui ha bisogno. Passa anche le informazioni (ROSSO / VERDE / BLU per Colore) a AbstractFactory per ottenere il tipo di oggetto di cui ha bisogno.



CODICE:

```
public interface Shape {
        void draw();
}
public class Rectangle implements Shape {
        @Override
        public void draw() {...}
} ...
public interface Color {
        void fill();
}


public class Red implements Color {
        @Override
        public void fill() {...}
}
public abstract class AbstractFactory {
        abstract Color getColor(String color);
        abstract Shape getShape(String shape) ;
}
```

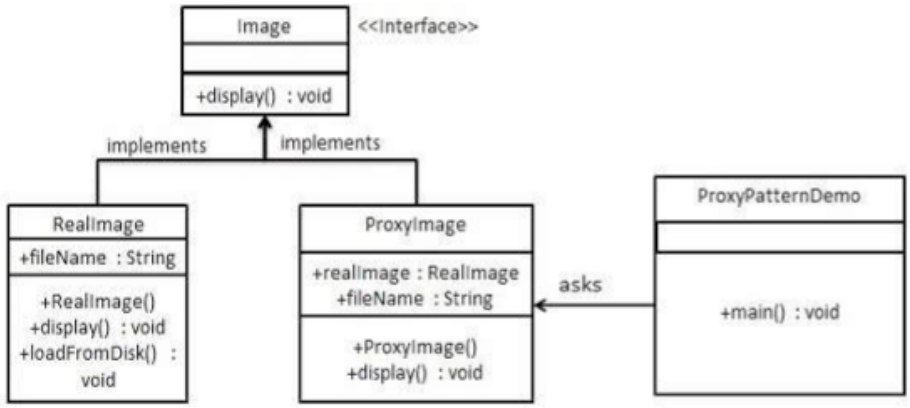
```
public class ShapeFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE"))
        {
            return new Circle();
        }else if
        (shapeType.equalsIgnoreCase("REC")){
            return new Rectangle();
        }else if
        (shapeType.equalsIgnoreCase("SQ")){
            return new Square();
        }
        return null;
    }
    @Override
    Color getColor(String color) {
        return null;}}
```

```
public class ColorFactory extends AbstractFactory {
        @Override
        public Shape getShape(String shapeType){
                return null;
        }@Override
        Color getColor(String color) {
                if(color == null){
                        return null;
                }
                if(color.equalsIgnoreCase("RED")){
                        return new Red();
                }else if(color.equalsIgnoreCase("GREEN")){
                        return new Green();
                }else if(color.equalsIgnoreCase("BLUE")){
                        return new Blue();
                }
                return null;
        }
}
```

Proxy pattern (Strutturale)

Nome e descrizione: per evitare **operazione pesanti e ripetitive.** Il proxy fa da rappresentante dell'oggetto pesante.

Descrizione del problema/quando viene usato: una classe rappresenta la funzionalità di un'altra classe. Creiamo un oggetto che ha un oggetto originale per interfacciare la sua funzionalità con il mondo esterno.



Soluzione: creiamo un'interfaccia Image e delle classi concrete che la implementano. ProxyImage è una classe proxy per ridurre l'impatto di caricamento sulla memoria dell'oggetto RealImage. ProxyPatternDemo, la nostra classe demo, utilizzerà ProxyImage per ottenere un oggetto Image da caricare e visualizzare.

CODICE:

```
public interface Image {

        void display();}
}
public class RealImage implements Image {
        private String fileName;
        public RealImage(String fileName){
                this.fileName = fileName;
                loadFromDisk(fileName);
        }
        @Override
        public void display() {
                System.out.println("Displaying " + fileName);
        }
        private void loadFromDisk(String fileName){
                System.out.println("Loading " + fileName);
        }}
```

```
public class ProxyImage implements Image{
        private RealImage realImage;
        private String fileName;
        public ProxyImage(String fileName){
                this.fileName = fileName;
        }
        @Override
        public void display() {
                if(realImage == null){
                        realImage = new RealImage(fileName);
                }
                realImage.display();
        }
}
```

Facade pattern (Strutturale)

Nome e descrizione: coinvolge una singola classe che fornisce metodi semplificati richiesti dal client e delega le chiamate ai metodi delle classi.

Descrizione del problema/quando viene usato: nasconde la complessità del sistema e fornisce un'interfaccia al client, è simile alla factory.

Soluzione: Creiamo un'interfaccia Shape e delle classi concrete che la implementano. ShapeMaker è la nostra classe facade, utilizza le classi concrete per delegare a loro le chiamate dell'utente. FacadePatternDemo, la nostra classe demo, utilizzerà la classe ShapeMaker per mostrare i risultati.

CODICE:

```
public class ShapeMaker { //facade class
        private Shape circle;
        private Shape rectangle;
        private Shape square;
        public ShapeMaker() {
                circle = new Circle();
                rectangle = new Rectangle();
                square = new Square();
        }
        public void drawCircle(){
                circle.draw();
        }
        public void drawRectangle(){
                rectangle.draw();
        }
        public void drawSquare(){
                square.draw();
        }}
```

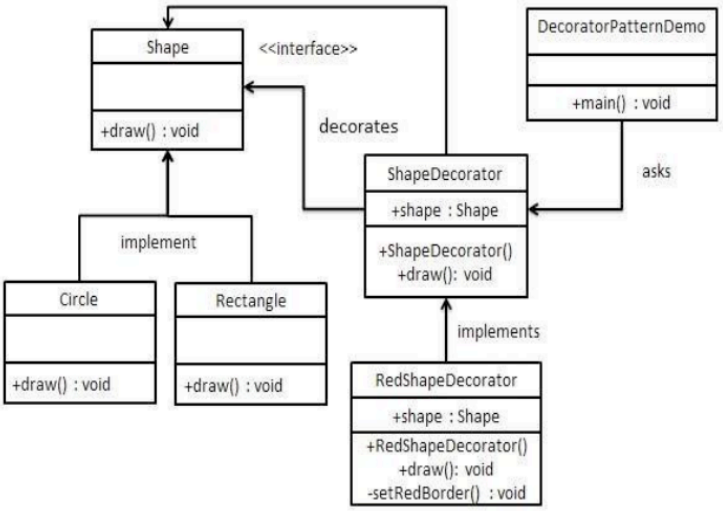
```
public interface Shape {
        void draw();
}
public class Rectangle implements Shape {
        @Override
        public void draw() {
                System.out.println("Rectangle::draw()");
        }
} ...
```

Decorator pattern (Strutturale - wrapper)

Nome e descrizione: aggiungere nuove funzionalità/caratteristiche alla classe senza toccare la classe iniziale.

Descrizione del problema/quando viene usato: consentire all'utente di aggiungere nuove funzionalità a un oggetto esistente senza alterarne la struttura.

Soluzione: creiamo una classe decoratore che avvolge la classe originale e fornisce ulteriori funzionalità mantenendo intatti i metodi della classe. Mostriamo un'esempio: decoreremo Shape con del colore senza alterare la classe. Creiamo un'interfaccia Shape e delle classi concrete che la implementano. ShapeDecorator è una classe decoratrice astratta che implementa l'interfaccia Shape e ha un oggetto Shape come sua variabile di istanza. RedShapeDecorator è una classe concreta che implementa ShapeDecorator. DecoratorPatternDemo è la nostra classe demo e utilizzerà RedShapeDecorator per decorare gli oggetti Shape.



CODICE:

```
public abstract class ShapeDecorator implements Shape {
	protected Shape decoratedShape;
	public ShapeDecorator(Shape decoratedShape){
		this.decoratedShape = decoratedShape;
	}
	public void draw(){
		decoratedShape.draw();
	}
}
public class RedShapeDecorator extends ShapeDecorator {
	public RedShapeDecorator(Shape decoratedShape) {
		super(decoratedShape);
	}
	@Override
	public void draw() {
		decoratedShape.draw();
		setRedBorder(decoratedShape);
	}
	private void setRedBorder(Shape decoratedShape){
		System.out.println("Border Color: Red");
	}
}
```

```
public interface Shape {
	void draw();
}
public class Rectangle implements Shape {
	@Override
	public void draw() {
		System.out.println("Shape:
Rectangle");
	}
}...
```

Capitolo 6 - Test del software

Il test del software serve per mostrare che il software fa quello per cui è stato realizzato rispetto ai requisiti e per scoprire eventuali difetti. Uso dati artificiali, sia dati comuni che a dati "strambi" o estremi .

NB: Il test rivela la presenza di errori , **ma non la loro assenza.** Se non trovo errori non posso dire che non ho errori.

Validazione e verifica del software

Validazione: il software fa quello per cui è stato realizzato (stiamo costruendo il software giusto?).

Verifica: test di possibili difetti (Stiamo costruendo bene il software?).

Sono importanti :

- Lo scopo del software.
- Le aspettative dell'utente.
- Contesto commerciale.

Abbiamo una parte di test e una parte di ispezione.

Ispezione: analisi statica del software (il software non è funzionante è solo l'analisi dei risultati), attraverso la documentazione, c'è un ispezione della struttura del codice sia sintattica che semantica.

TEST: analisi dinamica del software (software in funzione) e demo (uso comune ma anche casi estremi beta e alfa tester).

DOCUMENTAZIONE

È l'ispezione della STRUTTURA del codice sia SINTATTICA che SEMANTICA (hai usato le strutture viste a lezione? Sono usate bene?).

Ispezione del software

Si analizza:

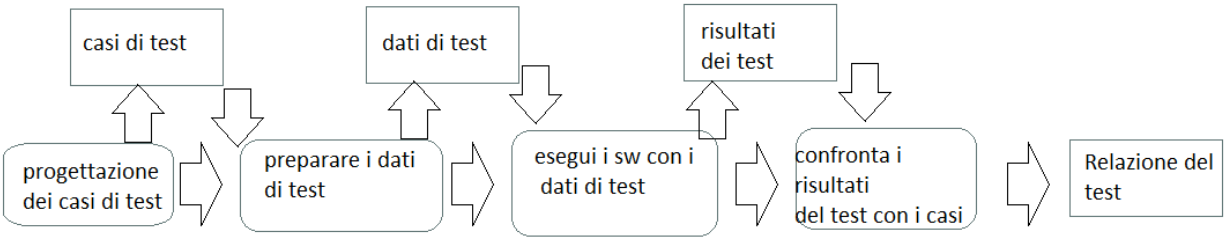
- Requisiti utente (prototipo del sistema).
- Architettura software.
- Schema della BASE DI DATI(non in questo corso).
- Codice vero e proprio.

TEST

Vantaggi delle ispezioni: ispezione più completa del test, riesco ad evitare **ERRORI CHE POSSONO COPRIRE ALTRI ERRORI** ove il test invece fallisce. Posso applicare la validazione anche con **software INCOMPLETO** e lavorare su aspetti di **QUALITà DEL software.**

Queste cose non ispezionano tutto: no run time, no memoria, ecc.

Produzione della relazione di TEST



Fasi di test

1) Test di sviluppo: test fatto da noi.

   Si dividono in: test di unità, test di architettura / interfaccia e test di sistema.

2) Test del sistema(rilasciato): test fatto da personale tecnico ma non da chi lo ha sviluppato (colleghi).

3) Test utente: si divide in alfa test e beta test.

Sviluppo guidato da test

Peso ai test da fare prima di scrivere codice, è u **SVILUPPO INCREMENTALE** usato di solito negli agili ma può essere usato anche nei plan driven.