Ingegneria del Software - Appunti presi a lezione

Perché Ingegneria del Software

Per realizzare sistemi complessi e risolvere i problemi che si verificano nello sviluppo degli stessi, ad es.:

- ritardo dei progetti rispetto ai termini prefissati
- sforamento budget
- scarsa affidabilità
- scarse prestazioni
- manutenzione ed evoluzione difficile
- alta percentuale di progetti software cancellati

Nasce negli anni '60, durante il passaggio dal software sviluppato informalmente ai grandi progetti di sistemi commerciali. Periodo di passaggio da **programmazione individuale** a **programmazione di squadra**, che quindi necessitava un nuovo approccio ingegneristico, con strumenti opportuni.

Crisi del Software risolta usando l'**Ingegneria del Software**

Secondo l'IEEE, **Software Engineering**: approccio sistematico allo sviluppo, all'operatività, alla manutenzione e al ritiro del software. Software è un **prodotto con il proprio ciclo vitale** e necessita di un **approccio sistematico**.

Temi di Ingegneria del Software

- **Processo Software**
  - **Organizzazione e gestione dei progetti**
  - **Composizione dei gruppi di lavoro**
  - S**trumenti di pianificazione, analisi e controllo**
  - **Cicli di vita del software**,
  - **Definizione e correlazione delle attività**.
- **Realizzazione di Sistemi Software**
  - **Strategie di analisi e progettazione**: tecniche per comprendere e risolvere un problema. Top-down, bottom-up, progettazione modulare, OO
  - **Linguaggi di specifica e progettazione**: UML, Reti di Petri, Z, OMT...
  - **Ambienti di Sviluppo**: strumenti analisi, progettazione e realizzazione
- **Qualità Software**
  - **Modelli di Qualità**, definizione caratteristiche di qualità
  - **Metriche Software**, unità di misura, scale di riferimento...
  - **Metodi di Verifica e Controllo**, verifica dei criteri di progettazione, controllo qualità, valutazione processo

Modelli del ciclo di vita del processo software

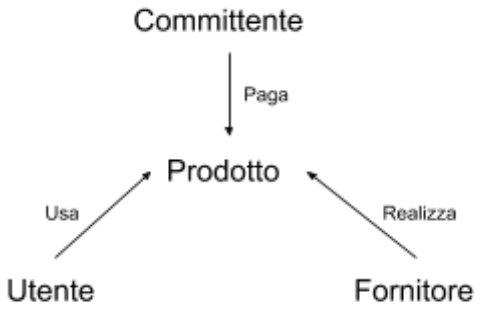
Che cos'è il processo software in termini di **processo aziendale**? Il ciclo di vita del software va da quando il software viene **concepito** a quando diventa **obsoleto**. Come in ogni processo aziendale, il processo di produzione del software ha **aspetti specifici** della produzione del software e **aspetti generali** della produzione industriale.

Un **processo** è un insieme di attività correlate che trasformano ingressi in uscite (**ISO 9000**).

Modellare un processo significa suddividere in attività e di ognuna dire **cosa**, **quali prodotti**, **quando**.

Queste attività vanno poi **organizzate per ordine** e vanno definiti i **criteri** per terminare ogni attività e passare alla successiva. Il processo è, quindi, **indipendente** dal software realizzato e denota solo le attività generiche, non applicate.

Organizzazione del lavoro per realizzare il software



Evoluzione dei modelli di ciclo di vita

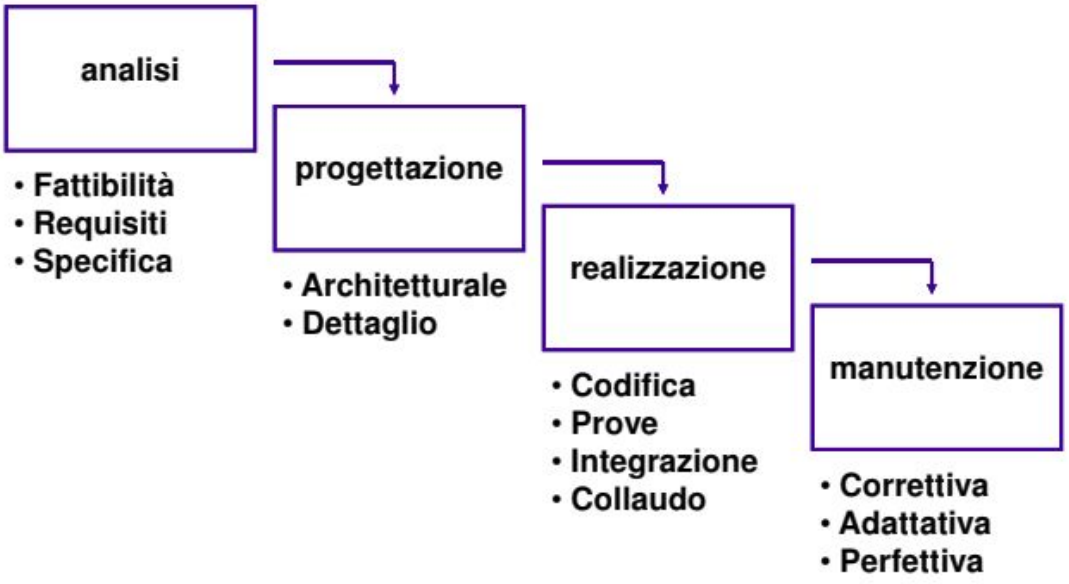
Da modelli molto **rigidi**, che riguardavano esclusivamente il software e con passi ben definiti e direzionati, siamo passati a modelli più **elastici** comprendenti anche indicazioni sui **rapporti fra le persone**, con il **cliente** eccetera.

Code&Fix

Il primo, è un non-modello poiché ha attività **non identificate** né **organizzate**, i progetti non sono gestiti ma si scrive codice e si corregge, ciclicamente.

Modelli prescrittivi

Modello a Cascata, 1970 Royce



Il grosso vantaggio di questo modello è che è stato il primo a fissare terminologia delle fasi del progetto software.

Le fasi sono sequenziali, senza possibilità di tornare indietro: quindi, in caso di eventi particolari come il cambio dei requisiti, **si ripartiva da capo**. Inoltre è un modello **estremamente documentato**, cioè **document driven**, poiché ogni fase produce dei documenti che la concretizzano e che sono necessari ad iniziare la fase successiva. Questa enorme produzione di documenti è considerata un **difetto**. Oltretutto è **inflessibile**, molto **burocratico**, **poco realistico** e non accetta il **cambiamento dei requisiti**.

- **Analisi**

    - **Fattibilità**: se sei in grado di **fare le cose** e se non sono già state fatte o **poter usare cose già fatte**, poterle fare meglio ecc...
    - **Requisiti**: capire **cosa deve essere realizzato nel sistema**. Usare tecniche per capire **effettivamente cosa va realizzato**. Un grosso investimento in questa fase paga tantissimo nelle fasi successive, poiché limita errori e incomprensioni delle specifiche, base fondamentale del progetto effettivo.
    - **Specifica**: come scrivere i requisiti nel linguaggio formale/informale, da passare poi ai progettisti.
- **Progettazione**
    - **Architetturale**: architettura in linea di massima, componenti da usare, tecnologie ecc...
    - **Dettaglio**: definire i componenti nel dettaglio, quasi fino a definire le singole classi.
- **Realizzazione**
    - **Codifica**: cioè la scrittura del codice
    - **Prove**: testing, test case ecc...
    - **Integrazione**: mettere insieme i pezzi prodotti e verificarne funzionamento
    - **Collaudo**: verifica finale prima del rilascio
- **Manutenzione**
    - **Correttiva**: correggere bug e rilasciare patch
    - **Adattiva**: adattare il software al cambiamento del dominio o dei requisiti
    - **Perfettiva**: reengineering del sistema, cambio della UI, incremento efficienza ecc...

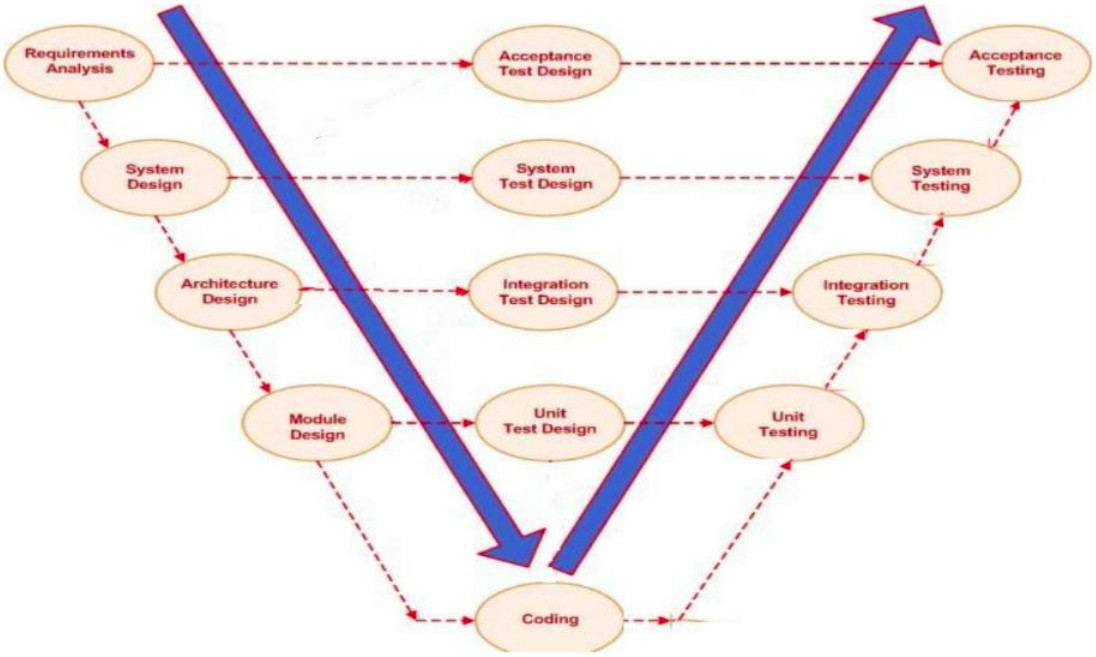
Iterativo incrementale

Questo modello è nato per soddisfare la necessità di adattabilità ai cambiamenti di soluzione, delle tecnologie e dei requisiti. La filosofia è di **ritardare la realizzazione** dei componenti che **dipendono criticamente da fattori esterni** come tecnologie, hardware sperimentale eccetera. Le iterazioni sono **pianificate** e la realizzazione è incrementale, per venire incontro alla necessità di uscire con qualcosa velocemente. Due modelli:

- **Requisiti stabili**
  Le fasi di **progettazione di dettaglio** e di **realizzazione** sono iterate realizzando incrementalmente le funzionalità e rilasciando mano a mano una versione incompleta che, alla fine delle iterazioni, sarà la versione completa del software
    - **Analisi e progettazione**, fatte una volta e per bene
    - **Progettazione di dettaglio**
    - **Realizzazione**
    - **Versione incompleta**
- **Requisiti instabili**, **modello evolutivo**
  Viene realizzato incrementalmente, iterando le fasi di **analisi e progettazione** e **realizzazione**, un **prototipo**, volto a ricevere feedback dal cliente per capire cosa cambiare, cosa aggiungere eccetera
    - **Analisi preliminare**
    - **Analisi e progettazione**
    - **Realizzazione**
    - **Prototipo** usa e getta

Modello a V

Della Hughes Aircraft, 1982. Altro modello sequenziale che però evidenzia l'attenzione rivolta alle fasi di test, progettati con anticipo con conseguente velocizzazione della produzione.
Sequenziale, incentrato sul testing. L'idea è

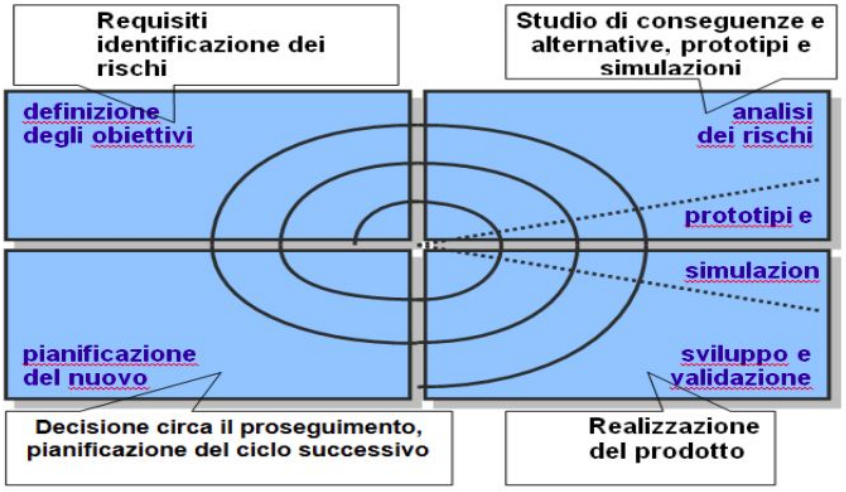


Plan-Do-Check-Act

Modello di sviluppo economico
Qualunque sviluppo industriale passava per queste fasi

Spirale

Nato nel 1988 ed ispirato al Ciclo di Deming del 1950: **plan**, **do**, **check**, **act**, ciclicamente. Evidenzia gli aspetti gestionali dello sviluppo, **pianifica** le fasi e analizza i rischi (**risk driven**). Questo modello è applicabile anche ai cicli tradizionali

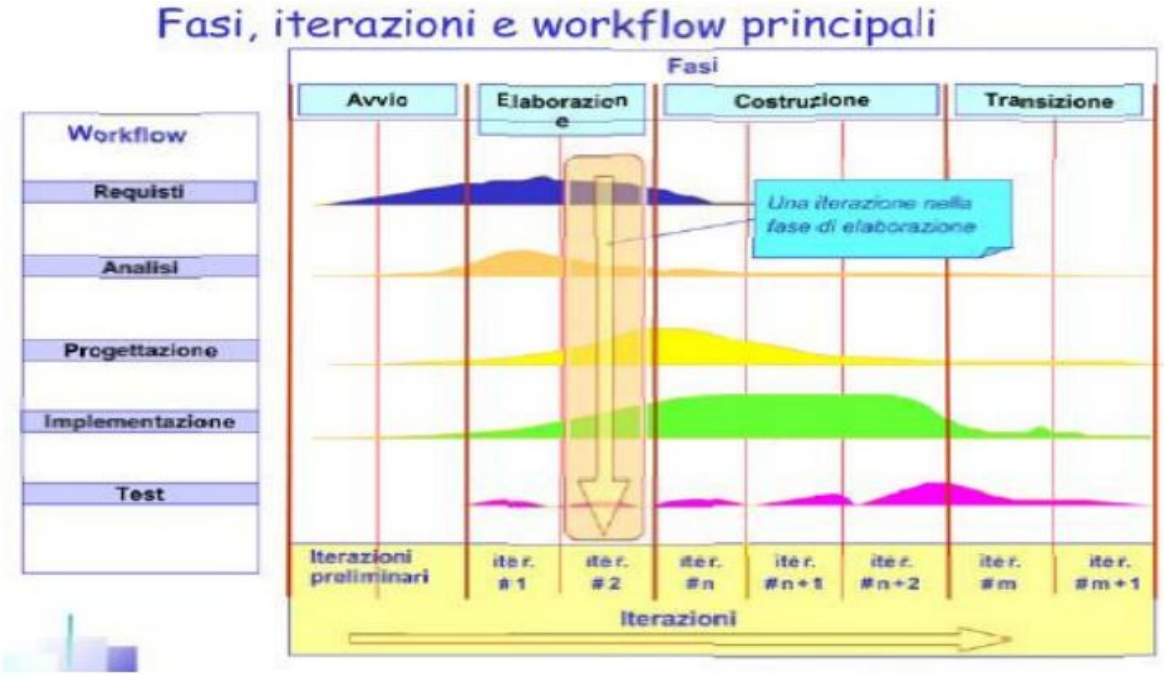


.

Unified Process

Guidato dai **casi d'uso** e dall'**analisi dei rischi**. La **raccolta requisiti** e i passi successivi sono guidati dallo **studio degli use case**, forniti dall'utente. Questo modello è particolarmente incentrato sull'architettura. Soprattutto nelle **prime fasi**, si concentra molto sull'architettura **in linea di massima**, lasciando i dettagli alle fasi successive. In tal modo è possibile avere da subito una **visione generale** del sistema **facilmente adattabile** al **cambiamento dei requisiti**.

Modello incrementale diviso in quattro fasi: **avvio**, **elaborazione**, **costruzione** e **transizione**. Il **workflow** è suddiviso in queste fasi: **requisiti**, **analisi**, **progettazione**, **implementazione** e **test**.



Modelli agili

Coinvolgono il più possibile il committente e sono adatti a progetti con meno di 50 persone. Una metodologia agile si basa su principi del Manifesto di Snowbird, di Febbraio 2001.

Il programmatore che smette di divertirsi a scrivere codice scrive schifezze. Centrale nei modelli agili è lo **sviluppatore**, la gerarchia aziendale è diversa e **priva di un capo**, generando così **ambienti più collaborativi** e volti ad un continuo confronto con il cliente.

Manifesto di Snowbird: *comunicazione*, *semplicità*, *feedback* e *coraggio*.

- **Comunicazione**: le persone e le interazioni sono **più importanti** dei processi e degli strumenti: tutti possono parlare con tutti. La **miglior risorsa di un progetto** sono **le relazioni e le comunicazioni tra gli attori**. Collaborare con i clienti oltre il contratto, perché la collaborazione diretta offre risultati migliori dei rapporti prettamente commerciali.
- **Semplicità**: analisti devono mantenere la **descrizione formale il più semplice e chiara possibile**. Si dà maggiore importanza al software funzionante che alla documentazione e si produce un codice **semplice e avanzato tecnicamente** riducendo la documentazione al **minimo indispensabile**.
- F**eedback**: nuove versioni del software vanno rilasciate ad **intervalli frequenti** ed il testing va eseguito **sin dal primo giorno**.
- C**oraggio**: dare in uso il sistema il **prima possibile** e **implementare cambiamenti richiesti man mano**. Bisogna essere **pronti a rispondere ai cambiamenti** più che aderire al progetto. Buttare via tutto e **rifare da capo se necessario**.

Extreme programming

Si basa su un insieme di prassi: **pianificazione flessibile**, basata su **scenari proposti da utenti** e **coinvolgendo programmatori**, e **rilasci frequenti**, due-quattro settimane.
In linea di massima: *rilascio qualcosa*, *ascolto feedback* e *proseguo lavoro*
Metafora condivisa: usare **metafore per descrivere funzionalità** del sistema software.
Progetti semplici, comprensibili a tutti. Verifica di unità e di sistema (basati sugli scenari). **Test Driven Development**: i casi di test denotano le specifiche.
Il cliente rimane **sempre a disposizione**.
Programmazione a coppie: un terminale, **driver scrive** il codice e **navigatore controlla** il lavoro in maniera **attiva**. **No lavoro straordinario**, perché a mente lucida si scrive codice migliore
Collettivizzazione del codice: **accesso libero**, **integrazione continua** per vedere se i componenti si funzionano bene quando vengono uniti e **standard di codifica**, cioè convenzioni comuni nello scrivere codice come particolari indentazioni, metodi di denominazione delle variabili eccetera..
Code Refactoring: modificare il codice senza cambiarne il comportamento. *Se un metodo necessita di un commento, riscrivilo!* **Codice auto-esplicativo**.
Daily Stand-Up Meeting, riunione ad inizio mattina in piedi in cui si fa il punto della situazione.

Scrum

Deriva dalla terminologia del rugby, basato sulla **terminologia della complessità**. Questo è un processo adottato per gestire e controllare lo sviluppo software. **Iterativo**, **incrementale** e volto allo sviluppo e alla gestione di ogni tipologia di prodotto. Fornisce, alla fine di ogni iterazione, un **set di funzionalità potenzialmente rilasciabili**. *Tutto si riduce a un mare di post-it*.
Fase iniziale in cui viene **deciso** tutto ciò che va **realizzato**, con ogni componente scritto su un post-it. Ogni post-it viene suddiviso in quattro categorie, a seconda dello stato del componente che denota: **non fatto**, **in progress**, **in test**, **finito**. Questi post-it compongono la **Product Backlog List**, cioè la lista di tutti i componenti conosciuti.
Successiva è la fase di **gioco**, cioè la parte agile dell'approccio Scrum. Questa fase è ciclica e produce qualcosa ad ogni ciclo. Ogni ciclo è breve (una settimana, un mese al massimo) ed è detto **sprint**. Lo sviluppo è quindi suddiviso in una serie di **sprint**, ognuno comprendente tutte le fasi tradizionali del ciclo software. Lo **scrum** vero e proprio è la riunione di 15 minuti fatta ogni giorno, in cui i membri dei vari team rispondono a delle domande di base come "*Cosa hai realizzato dall'ultimo scrum?*", "*Hai incontrato ostacoli?*", "*Cosa farai prima del prossimo scrum?*".
Ci sono tre ruoli principali nel modello scrum:

- **Product Owner**
  Si tratta di una **persona singola** (il cliente, un suo portavoce o un product manager) che ha **responsabilità economica**, **definisce le priorità** e decide gli **aspetti economici** del progetto.
- **Membro del team**
  Ogni team è composto da **7-9 persone** che lavorano a **stretto contatto**, **consegnano ciò che riescono a realizzare** in uno sprint e **scelgono il componente da realizzare** in uno sprint. **Non fanno più componenti contemporaneamente**, **non attendono gli altri team**, puntano a **ridurre gli sprechi** e si **auto-organizzano**, senza un team manager, quindi **senza specializzazione** dei membri singoli (al di là delle abilità personali).
- **Scrum master**
  Ha responsabilità ma non a livello formale. Si tratta di un membro che **guida i team**, **rimuove le barriere di produttività** e fa **scudo dalle interferenze esterne** senza essere autoritario. Il suo obiettivo è **tirare fuori il meglio dalle persone** con ogni mezzo possibile, dalle gite in barca ai discorsi alle serate nei pub. Un suo comportamento tipo è chiedere "*Vedo che …, cosa dovremmo fare?*".

L'idea fondamentale è la **felicità del programmatore**.

Non sono modelli alternativi l'uno all'altro. Definiti in tempi diversi, con mentalità diverse e obiettivi diversi, ma **ogni modello aggiunge qualcosa al precedente**, anche rimuovendo difetti.

Studio di Fattibilità

Lo **studio di fattibilità** è la fase preliminare nella quale si **stabilisce l'opportunità** o meno di realizzre il software. Si basa su una descrizione sommaria del sistema software e delle necessità dell'utente.
le informazioni necessarie per lo studio di fattibilità coinvolgono principalmente

- **Committente**, cioè gli utenti finali del sistema
- **Responsabile del progetto**, il commerciale (colui che si occupa di vendere in un'azienda) e l'analista

Lo studio si basa sulla **valutazione dei costi e dei benefici** di una possibile attività di produzione.
La **fattibilità tecnologica** riguarda:

- Gli **strumenti per la realizzazione** (il software, le libreria ecc...)
- Le **soluzioni algoritmiche e architetturali**
- l'**hardware**
- Il **processo** da seguire (la prototipazione, i progetti esplorativi, la ricerca ecc...)

Lo studio di fattibilità consiste anche nell'analizzare gli **aspetti economici e di mercato**:

- Valutare un **confronto tra il mercato attuale e quello futuro**
- Valutare il **costo della produzione** e **la redditività dell'investimento**

Analisi dei Requisiti

L'**attività di analisi dei requisiti** consiste nello **studiare e definire il problema da risolvere**, cioè riguarda il definire ***COSA*** va fatto, ***NON COME***. In questa fase bisogna:

- capire **cosa deve essere realizzato**
- documentare **cosa deve essere realizzato**
- **negoziare con il committente/fornitore**

L'analisi dei requisiti è una delle fasi più cruciali, poiché aver definito dei **requisiti** risultati poi **incompleti** è una delle cause principali dell'abbandono di un progetto software. Quindi è importante spendere del tempo in questa fase per avere una serie di **implicazioni economiche** e di **qualità** altamente positive nel resto della realizzazione del progetto. Questo perché correggere requisito sbagliato:

- durante l'analisi dei requisiti costa 1
- durante la progettazione 5
- codifica, 10
- test unità, 20
- test accettazione, 50
- operazione, 100

Diagramma dei casi d'uso
granularità: quanto spezzetto
caso d'uso incluso: comportamenti comuni a più casi d'uso (es. assegnazione e modifica tessera prevedono aggiornamento diritti d'accesso). si specifica con linea tratteggiata che unisce l'incluso all'includente con lo stereotipo <<include>> come etichetta

Prodotti dell'attività di analisi

Documento o **modello analitico** di:

- descrizione del **dominio**
- descrizione dei **requisiti**

Opzionalmente vengono prodotti, anche in parallelo, il **manuale utente** e i **casi di test**.

Dominio

Il **dominio** è il **campo di applicazione del prodotto**, cioè il **processo che si vuole automatizzare** e il suo **contesto**.

Requisito

Un **requisito** è una condizione o una capacità **necessaria all'utente** per risolvere il problema o che deve essere **soddisfatta dal sistema** per soddisfare un contratto. Si dividono in:

- **Funzionali**: le **funzionalità che il sistema deve realizzare**, in termini di:
    - **azioni** che deve compiere
    - come il software **reagisce** a specifici tipi di input
    - come si **comporta** in situazioni particolari

  Sono i requisiti a cui è dato il maggior valore
- **Non funzionali**, o **di qualità**: sono **vincoli sul progetto**, descrivono le proprietà del sistema software in relazione a determinati servizi o funzioni, ad esempio:
    - **efficiente**, nel tempo e nello spazio (tempi di risposta, dimensioni...)
    - **affidabile**, di sicurezza (safe e secure)
    - quale **standard di processo**
    - **vincoli legislativi**, **interoperabilità** con altri sistemi
    - **requisiti fisici**
    - **robustezza**
    - **fault tollerante**

Buona norma tenere i due tipi di requisiti ben separati.
Ad esempio: "*sistema deve validare pin inserito entro 3 secondi*"

- **Funzionale**: il sistema deve validare il pin
- **Non funzionale**: la validazione deve avvenire entro 3 secondi

Esistono **approcci diversi** ma **non mutualmente esclusivi**:

- basati su **linguaggio naturale** (glossario, **specifica dei requisiti**)
- basati su **linguaggio formale** (**metodo jackson**, **UML**)

Documento dei Requisiti

- **Acquisizione** tramite
    - interviste, con committente e utenti finali strutturate o non
    - questionari per i futuri utenti
    - osservazione dei futuri utenti al lavoro
    - studio di documenti
    - produzione di prototipi
    - casi d'uso

  eseguito con i committenti, gli utenti e i responsabili marketing
- **Elaborazione** in cui i requisiti vengono espansi e raffinati.
    - Produzione del **documento dei requisiti**, che **descrive il dominio** basandosi sull'**uso del linguaggio naturale**.
- **Negoziazione**
- **Convalida**
- **Gestione**

Struttura del Documento dei Requisiti

- **Introduzione**, **perché il sistema è desiderabile** e **come si inquadra negli obiettivi** più generali del cliente
- **Glossario**, spiegazione dei termini e dei concetti tecnici usati
- **Definizione dei Requisiti Funzionali**, cioè dei servizi richiesti
- **Definizione dei Requisiti Non Funzionali**, cioè dei vincoli operativi del sistema e dei vincoli sul processo di sviluppo
- **Architettura**, la strutturazione in sottosistemi
- **Specifica di system and software requirements**, la specifica dettagliata dei requisiti funzionali

- **Modelli astratti** del sistema, modelli formali/semi-formali ciascuno illustrante un solo punto di vista (controllo, dati, funzioni ecc...)
- **Previsioni** di successive **evoluzioni del sistema**, per esempio di hardware o di requisiti
- **Appendici** comprendenti un'individuazione ed eventuale descrizione della piattaforma hardware, i requisiti di database, il manuale utente, i piani di test...
- **Indici**, con il **lemmario**: lista di termini, con puntatori ai requisiti che li usano

Glossario

Definizioni dei termini chiave del dominio. Viene usato come **strumento per**: **comprendere** e **documentare il dominio**, **condurre le interviste**, **validare i requisiti**.

Validazione dei Requisiti

Per validare un documento dei requisiti già strutturato si usano più tecniche:

- **Deskcheck**
  - **Walkthrough**, cioè la lettura sequenziale dei documenti
  - **Ispezione**, cioè la lettura strutturata dei documenti (tecnica del lemmario, ricerca di rimozioni, distorsioni, generalizzazioni)
- **Prototipi**

Difetti da evitare:

- **Omissioni**, cioè la mancata presenza di un requisito (**incompletezza**). Di questo me ne accorgo **parlando col cliente**, col **prototipo**...
- **Inconsistenze**, contraddizioni tra requisiti o tra requisiti e ambiente operativo
- **Ambiguità**, requisiti con significati multipli
- **Sinonimi** e **omonimi**
- **Presenza di dettagli tecnici**
- **Ridondanza**, può esserci **solo tra sezioni diverse**

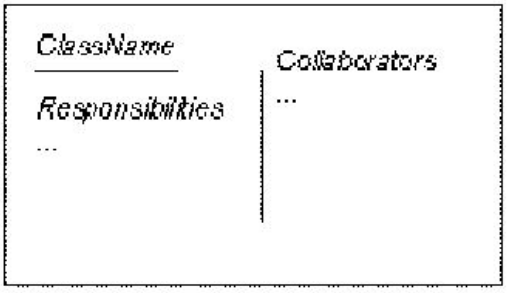
PNL Programmazione neurolinguistica di Noam Chomsky

I **filtri linguistici** sono i processi attraverso i quali costruiamo la nostra mappa del mondo:

- **Generalizzazione**, processo con il quale, partendo da un'esperienza specifica, la si decontestualizza traendone un significato universale
- **Cancellazione**, processo di selezione dell'esperienza, prestare attenzione solo ad alcuni pezzi, escludendone altri
- **Deformazione**, si tratta di una percezione distorta della realtà, cioè interpretare il mondo esterno secondo le proprie mappe.

CRC Cards

Per abbozzare rapidamente un modello di dominio. Una card è grande circa 10x15cm



es. sistema di controllo degli ascensori:

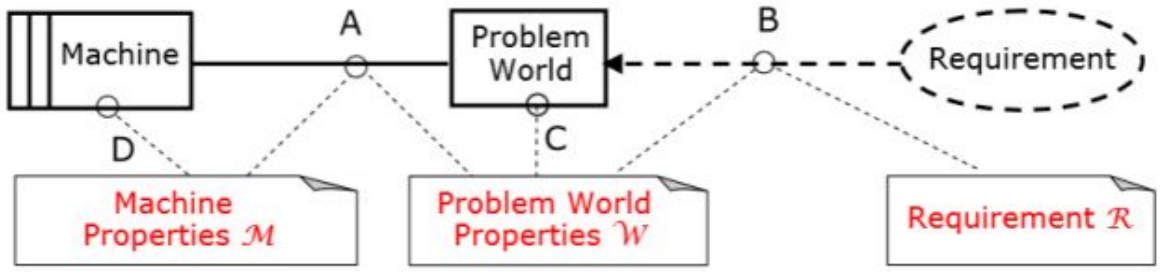
- Classi: cabina, pulsantiera, motore

Definite le classi si può continuare la discussione, scrivendo sulle card le responsabilità

- Per es. la pulsantiera deve
    - “ricevere” dall’utente il piano di destinazione
    - mostrare il piano corrente
    - comunicare la destinazione al motore

Il Metodo Jackson

Michael Jackson not the singer http://mcs.open.ac.uk/mj665/
Insieme delle “scatolette” che permettono di comunicare ai progettisti cosa desiderano.
Il problema della correttezza:
$\mathcal{M}, \mathcal{W} \models \mathcal{R}$
M, W soddisfa R con: M **macchina**, W **world** in cui opera, R **requisiti**
Quindi il **sistema** specificato da **M** in un **mondo** descritto da **W** soddisfa i **requisiti** indicati da **R**
Macchina: **hardware e software**, una macchina per un problema
Mondo: **tutto ciò che non è la macchina**, contesto/dominio in cui la macchina opera ed **esiste indipendentemente dalla macchina**
Interfacce: i **fenomeni condivisi** tra macchina e mondo.
La soluzione sta nella macchina, il problema sta nel mondo. La macchina opera attraverso le interfacce.
Un **requisito** è una **condizione del mondo**, parla il linguaggio del contesto del problema. La macchina deve realizzarli (es. ascensore arriva su richiesta).
Il mondo ha caratteristiche e componenti **indipendenti dalla macchina** (es. il motore dell’ascensore).



La macchina e il mondo **collaborano** per risolvere problema.
Fenomeni A: condivisi tra macchina e mondo
Fenomeni B: menzionati nei requisiti
C, D **fenomeni privati** (non condivisi) della macchina e del mondo

es. requisito biblioteca:

- *solo membri possono prendere in prestito*
- *reminder per chi non restituisce in tempo*
- *raccolta multe e tasse*
- *catalogo aggiornato*

Libreria
Proprietà di un libro
mondo: n copie, titolo, autore, isbn...
macchina: id

Ascensore
Proprietà
mondo: ascensore, pulsanti, porte

macchina: accensione/spegnimento motore, apertura/chiusura porte

Definire i confini del sistema software, cosa da realizzare e cosa è già soddisfatto

requisiti o proprietà del mondo?

- *ascensore non si muove se porte aperte,* ***requisito***
- *piano n raggiunto da n-2 solo tramite n-1,* ***proprietà***
- *freccia in alto è illuminata l'ascensore non si muove verso il basso,* ***requisito***
- *pulsante di richiesta al piano spento quando ascensore si ferma al piano* ***requisito***
- *le porte al piano e porte dell'ascensore si aprono e chiudono insieme* ***requisito***
- *nel display che indica posizione ascensore è illuminato un solo piano alla volta,* ***requisito*** *scritto male, mescola proprietà (no contemp su 2 piani) con requisito di dire esattamente a che piano è.*

Context diagrams

Descrivono le **relazioni** tra: il **contesto del problema** (dominio applicativo), la **soluzione software**, e le **interfacce tra dominio e soluzione**. Il **dominio** viene **organizzato in sotto-domini**:

- **Machine domain**: **hardware e software da realizzare**
  Rappresentato come un rettangolo con due righe verticali.
  Un solo machine domain nel context diagram
- **Problem domain**: Il **mondo reale esterno** alla macchina
  Strutturato in sotto-domini
  - **Given domain**: **dominio dato**, non può essere modificato.
    Rappresentato come un rettangolo (*è l'utente, il libro, non la catalogazione*)
  - **Designed domain**: **rappresentazione del mondo del sistema**, costruito costruendo il sistema.
    Rappresentato da un rettangolo con una linea verticale.

  Consigli su come strutturare il mondo: **alcune infrastrutture fanno parte del mondo** (internet, rete cellulare, linee fisiche che collegano negozi e banche) ed **altri possono essere progettati e sono quindi parte del sistema** (formato degli sms mandati da un dispositivo di monitoraggio antifurto di una macchina)

Problem diagrams

Estendono i **context diagrams introducendo i requisiti**: mostrano le relazioni tra i requisiti e i sotto-domini.
I requisiti vengono rappresentati con ovali tratteggiati collegati da: **linee tratteggiate** se il requisito **parla del mondo** *la linea legge*, oppure da **frecce tratteggiate** se il requisito **impone vincoli sul mondo** *la freccia scrive*

Decomporre il problema

Dato un problema, **la macchina è ciò che dobbiamo sviluppare**, **il mondo è ciò che è dato**.
La macchina è una vista parziale delle funzioni software. Ogni problema ha una macchina specifica

1 macchina > 1 problema

La macchina di un problema può essere il dominio di un altro problema.
Si possono scomporre problemi in sottoproblemi: un **sottoproblema** è un **problema più semplice** e viene chiamato **componente del problema**.
Bisogna inoltre distinguere le interfacce tra domini e le interfacce utente. Un'**interfaccia fra domini** è **l'insieme di fenomeni condivisi** (eventi, stati, valori, ...)

- **tutti i domini di un'interfaccia condividono i fenomeni**
- **solo un dominio può causare un determinato fenomeno**
- **i fenomeni possono trasportare informazioni**

I **fenomeni** sono di due tipi:

- **Individuali**
  - **eventi**, istanza di qualcosa che accade
  - **entità**, qualcosa che persiste nel tempo

    - **valori**, un elemento di una data sorta algebrica
- **Relazioni**
    - **stati**, rapporti tra entità e valori
    - **verità**, predicati tra gli individui (statici)
    - **ruoli**, rapporti tra eventi e individui

Safety: software non fa danni
Security: segretezza, privacy ...

UML

Attività di analisi e progettazione modelli

Attività centrate sul modello (diversamente dalla realizzazione che è centrata sul codice). Verifica il modello rispetto ai requisiti e si revisiona il modello per risolvere i problemi.
Modello: astrazione di un sistema (o del dominio) usata per specificarne struttura e comportamento. Contiene la conoscenza sul "cosa" e sul "come" di un sistema, fondamentale per un processo di progettazione collaborativo e centrale nel moderno sviluppo software.
es. *il codice è un modello del calcolo eseguito dalla macchina*.

UML, Unified Modeling Language: linguaggio di modellazione grafico basato su sistemi software.
Si tratta di un **linguaggio generale**, non specifico per definire applicazioni software object oriented, i cui **diagrammi** sono generalmente **facilmente comprensibili**. Il suo principio base è che un sistema software possa essere visto come un insieme di oggetti che collaborano. Considera due aspetti fondamentali del sistema: la **struttura statica**, cioè gli oggetti necessari e le loro relazioni, e il **comportamento dinamico**, quindi come gli oggetti collaborano.
UP, Unified Process: specifica un processo di sviluppo del software, indica le attività che devono essere eseguite, i manufatti, eccetera. Descrive le attività e "raccomanda" l'uso di UML ma non lo necessita.

La **modellazione** riguarda ogni fase del processo di sviluppo. Essendo **applicabile a più tipi di progetti e domini** è anche **indipendente da linguaggio di sviluppo e dal modello di processo**. Quindi abbiamo una unificazione a livello di linguaggio, non di metodo.
Obiettivi della modellazione:

- **visualizzazione**, comunicazione e comprensione
- **specifica e documentazione**, descrizione del sistema in tutti i suoi livelli
- **realizzazione**, supporto all'automazione di codifica

Concetti di UML

Modello: astrazione di (parte di) un sistema.
Modello statico: descrive elementi del sistema e le loro relazioni.
Modello dinamico: descrive il comportamento del sistema nel tempo.
Progetto (o **disegno**): l'insieme dei modelli.
Vista: descrizione di un aspetto di un modello.

Caratteristiche dei modelli

Un modello deve essere **tollerante a inconsistenza e incompletezza**, possedere **meccanismi di strutturazione** (package), di **personalizzazione mediante stereotipi** e **strumenti di supporto disponibili sul mercato**.

Classificatori vs Istanze

Classificatore: modella un concetto che descrive istanze (es. *una classe modella oggetti*). In UML i classificatori sono: classi, attori, casi d'uso, componenti, nodi, ...

Alcuni diagrammi possono essere a livello di classificatore o di istanza.
Un classificatore è uno spazio di nomi e gli elementi contenuti hanno un nome unico.
Per esempio una classe è un classificatore di oggetti, un package è un classificatore di package, classi, diagrammi....

Annotazioni

note, {vincoli}
Alcuni dettagli del modello sono espressi in forma testuale. Commenti e vincoli sono annotazioni nei diagrammi, collegate ad elementi del modello con una linea tratteggiata. I vincoli sono elementi del modello, i commenti no.

Stereotipo

Primitiva di UML comune ad ogni diagramma: rende un diagramma più informativo arricchendo la semantica dei costrutti UML. Viene scritto tra virgolette ed abbinato ad un elemento del modello, ad esempio *<<import>>, <<interface>>, ...*
Forniscono quindi un significato aggiuntivo ai costrutti UML. Possono essere usati per adattare UML a particolare ambiti e piattaforme di sviluppo e sono definiti nei profili, che costituiscono uno dei principali meccanismi di estensione UML.

Diagramma dei casi d'uso

Contenuti: requisiti

- **modello statico**
  - diagramma dei casi d'uso
- **modello dinamico**
  - narrazione associate ai casi d'uso

casi d'uso > requisiti funzionali, quelli non funzionali si raccontano a parole con il diagramma dei requisiti.
Il diagramma dei casi d'uso cattura il comportamento di un sistema, visto dall'esterno.
Un **attore** è un un'entità esterna al sistema, che interagisce con esso in un determinato ruolo: utente, altro sistema, ...
Un **caso d'uso** è una **funzionalità o servizio offerto dal sistema a uno o più attori** e viene espresso come un insieme di **scenari**.
Uno **scenario** è una **sequenza di interazioni tra sistema e attori**.

Sintassi e semantica:

- **attore**, omino con nome in UpperCamelCase
  Utente o sistema in un particolare ruolo.
- **associazione**, senza nome, linea da omino a ovale. Associazione molti a molti
  Rappresenta l'interazione
- **caso d'uso**, ovale con nome (verbo) in UpperCamelCase. I casi d'uso sono senza attori solo se inclusi (included).
  Rappresenta un compito (task) che gli attori eseguono con l'ausilio del sistema. Un caso d'uso è iniziato solo da un attore (principale), eventualmente il tempo.
- **sistema**, rettangolo con o senza nome che contiene i casi d'uso

visual paradigm community edition programma da scaricare

Diagramma delle classi

In UML le classi sono classificatori che rappresentano le classi di oggetti e, nei diagrammi, sono indicate con quadrati contenenti:

- **Nome** dell'entità, in UpperCamelCase
- **Attributi**, indicati con *nome: tipo*, in lowerCamelCase

- **Metodi**, indicati con *nome(parametri : tipo):tiporitornato*, in lowerCamelCase

Se non interessanti, attributi e metodi possono essere **omessi**, mentre se sono statici vengono indicati sottolineati.
Una **classe** quindi **cattura un concetto del dominio del problema**, descrive un **insieme di oggetti con caratteristiche simili**. Un **oggetto** è un'entità istanziata nel tempo, con uno **stato** ed un **comportamento**.
L'identità di un dato oggetto è quindi vista a livello di istanza, con gli attributi che ne definiscono lo stato e le operazioni che ne definiscono il comportamento.

Attributi

Sintassi: *visibilità nome:tipo[molteplicità]=valoreiniziale {proprietà}*
Il nome ed il tipo sono obbligatori.
es.

+ colore:Saturazione[3] {>=0}
+ nome:String[0..1] {Ordered}
+ n:int=0 {>=0}
- punto:int[2]
- seq:int[10] {>=3, <=33, ordered}

Gli attributi e i metodi hanno un certo grado di **visibilità**:

- \- privato: solo le operazioni della classe possono vedere e usare questo elemento
- \+ pubblico: accessibile ad ogni elemento che può vedere e usare la classe
- \# protected: accessibile ad ogni elemento discendente
- ~ package: accessibile solo agli elementi dichiarati nello stesso package

Operazioni

Sintassi: *visibilità nome(listaParametri):tipoRitorno {proprietà}*
Il nome e la lista dei parametri sono obbligatori, la lista può essere vuota
ed.

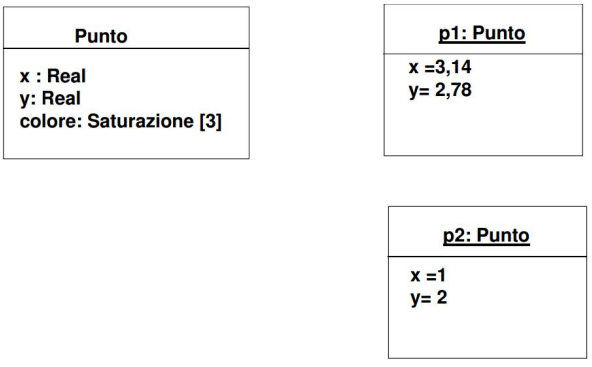
+ sum(a:int, b:int):int
+ sum(a:int, b:int=10):int
- gra(): Gra

Oggetti

Vengono indicati seguendo uno schema simile a quello delle classi:

- *nomeoggetto:nomeclasse*
- attr1:tipo=valore
  attr2:tipo=valore
  ...
  Il tipo ed il valore possono essere omessi, l'intera lista dei singoli attributi è opzionale.

Es.



Classi di analisi

Una **classe di analisi** corrisponde ad un concetto concreto del dominio, ad esempio quelli descritti nel glossario. Normalmente ogni classe di analisi viene raffinata in una o più classe di progettazione.
Una classe di analisi ha le seguenti caratteristiche:

- **Astrae** uno specifico elemento del dominio
- Possiede un **numero ridotto di responsabilità** (**funzionalità**)
- Non riguarda classi **onnipotenti**: attenzione al chiamare classi “sistema”, “controllore”...
- Non è una **funzione travestita da classe**
- Non possiede una **gerarchia di ereditarietà troppo profonda** (>=3)
- **Coesione e disaccoppiamento**:
    - Responsabilità simili in un'unica classe
    - Limitare le interdipendenze tra le classi

Deve contenere operazioni e attributi di alto livello, solo quelli veramente utili, senza inventare niente e limitando la specifica di tipi, valori ecc...

Identificazione delle classi

Problema classico delle prime fasi di sviluppo.
Approccio **data driven**: si identificano tutti i dati del sistema e si dividono in classi. (**identificazione dei sostantivi**)
Approccio **responsibility driven**: si identificano le responsabilità e si dividono in classi. (**CRC Cards**)

Analisi nome-verbo
Sostantivi → classi o attributi
Verbi → responsabilità o operazioni
Passi:

- individuazione delle classi
- assegnazione degli attributi e delle responsabilità alle classi
- individuazione delle relazioni tra classi

Problemi ricorrenti:

- tagliare le classi inutili, quindi trattare i casi di sinonimia
    - Elencare i sostantivi ed eliminare quelli riconosciuti come sinonimi, eventi, inutili (estranei al sistema), metalinguaggio (sistema), attributi (nome del libro...)
- individuare classi nascoste, cioè implicite del dominio del problema che possono anche non venire mai menzionate esplicitamente

Relazioni

Una **relazione** rappresenta un legame tra due o più oggetti, normalmente istanze di classi diverse ma non è sempre vero.
In matematica, una relazione binaria tra A e B è un sottoinsieme del prodotto cartesiano AxB. In UML abbiamo notazioni grafiche per: prodotti cartesiani (a livello di classificatori), coppie (a livello di istanza), anche triple, quadruple, n-arie usando il rombo.
Si hanno tra classi: associazione (aggregazione, composizione), generalizzazione.
Tra oggetti: collegamenti.
Si hanno relazioni di dipendenza: uso, realizzazione, istanza.

Associazione binaria

ClasseA ruoloDiA nome ruoloDiB Classe B

Ruoli e nome sono opzionali, ma almeno uno è presente (**raramente entrambi**). Il ruolo serve a **parlare meglio dell'oggetto associato**, soprattutto se sono oggetti della stessa classe.

Nome e ruoli in **lowerCamelCase**. Il **nome** è normalmente un **verbo** e il **ruolo** è un **sostantivo**.

Si può indicare con **una freccia** il **verso di lettura** dell'associazione e la **molteplicità**, cioè il **numero di oggetti coinvolti** in un'associazione **in un dato istante**.

1, 0..n, 1..*, n..*, *

Nell'aggregazione si rappresenta una relazione parte - tutto

Nella composizione le parti non hanno senso senza il tutto.

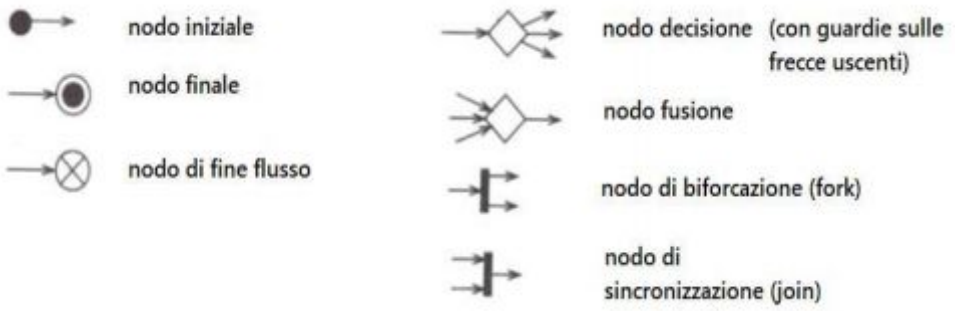
Dominio Dinamico

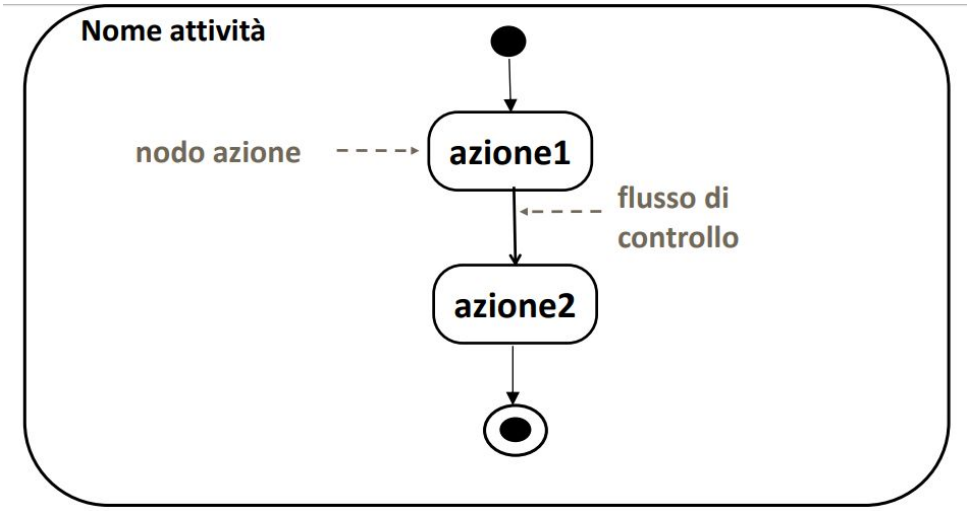
Diagrammi di Attività

Modellano il **flusso di lavoro** (workflow, business model) di una **computazione** (software) o di un **processo/attività** (business). Un'attività descrive la coordinazione su un insieme di azioni. Essa è centrata su:

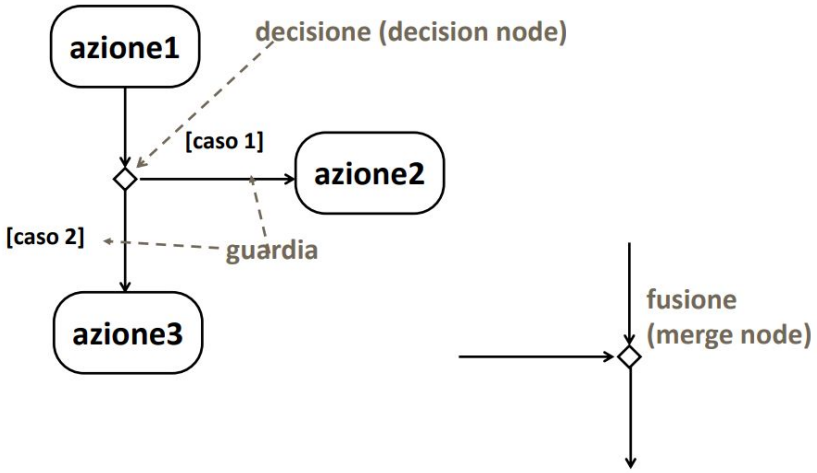
- sequenza e concorrenza delle azioni
- condizioni che abilitano

Gli antenati di questi diagrammi sono i diagrammi di flusso e le Reti di Petri.

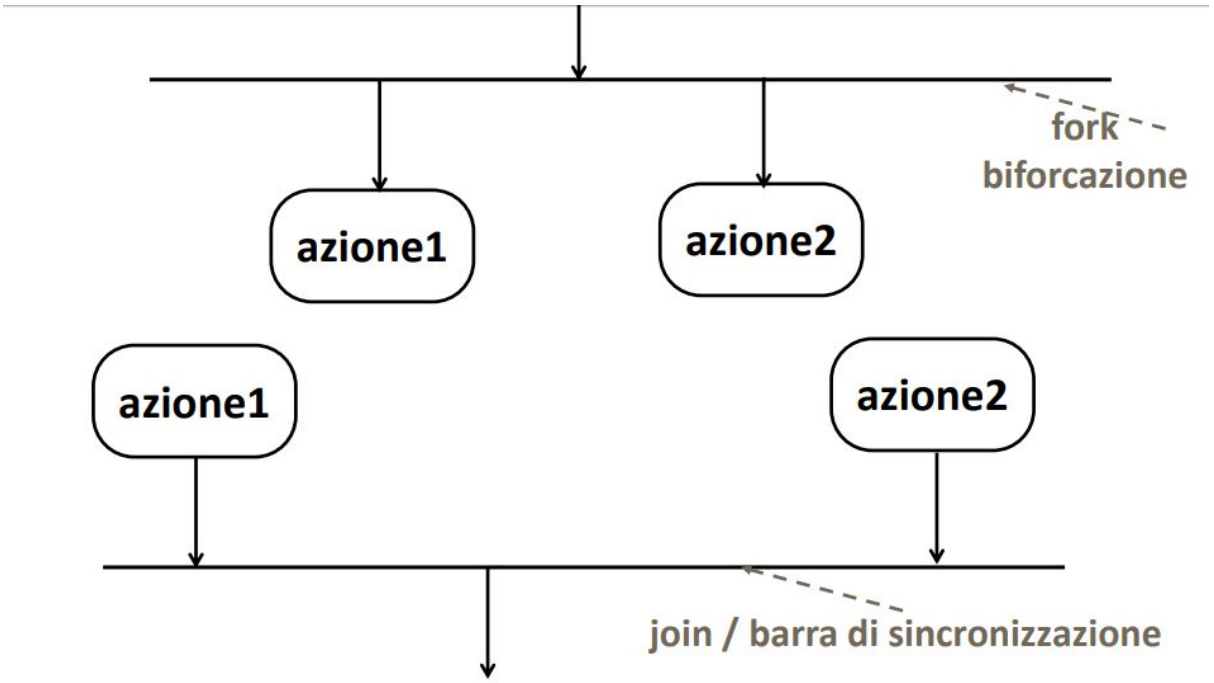




Solo **una freccia entrante** e **una uscente per azione**



Il flusso (token) **prende uno dei cammini**, e le guardie devono **coprire tutte le possibilità**, usando in caso [else].
Non è necessario, ma è bene che le guardie siano **mutualmente esclusive**, altrimenti si avrebbe un comportamento ambiguo.
Dato un nodo decisione, non è obbligatorio che ci sia un nodo fusione corrispondente, potrebbe esserci un nodo di fine flusso.



La **fork moltiplica i token**:

- dato un token in ingresso, ne **produce uno per ogni uscita** = ogni "braccio" viene eseguito

La **join consuma i token**:

- si **attende un token per ogni freccia entrante**, si **consumano tutti** e ne **esce solo uno**

Non c'è vincolo di una join per ogni fork.

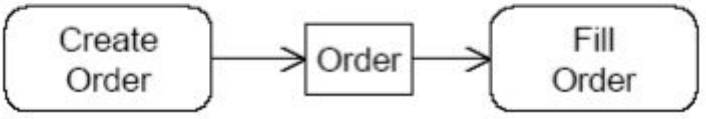
Le **azioni possono avere input e output**: tali dati sono scambiati tramite oggetti

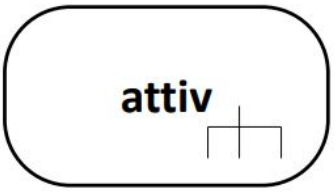
Gli oggetti:

- possono **avere uno stato**, ad es. [paid]
- hanno **nome maiuscolo e non sottolineato**

Un'azione può **includere** (cioè **richiamare**) **un'altra attività** secondaria:

- si usa il **rastrello** (**rake**) per dire che l'azione include una sotto-attività
- la sotto-attività si descrive in un diagramma a parte

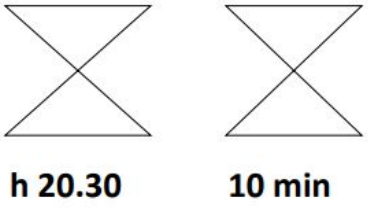




Accettazione di un evento esterno

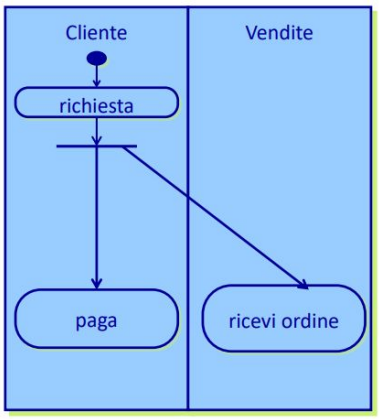
Invio di un segnale

Accettazione di un evento temporale



Nell'accettazione di un evento esterno o temporale, **l'arco entrante non è necessario**: quando arriva l'evento si genera un token. In caso di arco entrante invece, l'azione è abilitata quando arriva un token e attende l'evento esterno per farlo transitare.

Una **partizione** è utile per **dividere i gruppi le azioni in gruppi**. Spesso corrisponde alla divisione in unità operative di un modello business. Permettono di **assegnare la responsabilità delle azioni**.

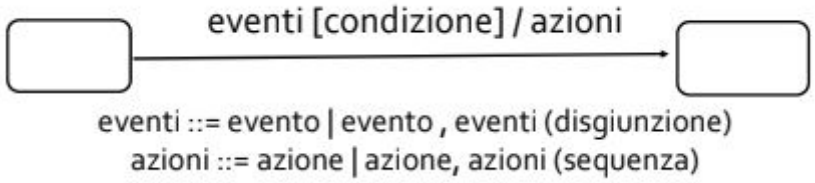


Macchina a Stati
Una macchina a stati descrive il comportamento dinamico delle istanze di un classificatore.
Per costruirla dobbiamo **individuare gli stati significativi in cui si può trovare un oggetto**. Bisogna inoltre descrivere **come da ciascun stato si può transitare in un altro**.
Le transizioni avvengono in risposta al verificarsi di un evento, tipicamente **messaggi** o **eventi generati internamente**.
Una macchina a stati è rappresentata da un grafo di stati e transizioni, associata ad un classificatore.
Uno **stato**:

- è un insieme di valori di un oggetto (di alcune variabili significative)
    - astrazione dello stato concreto (valore di tutte le variabili)
    - rappresenta uno stato significativo
    - caratterizzato dal dare la stessa risposta qualitativa ad eventi che possono accadere
- ha un nome unico
- può essere composito

Rappresentati con rettangoli arrotondati, disco nero (stato da cui partire), disco nero bordato (stato finale, terminazione).

Una **transizione** collega tra loro due stati, rappresentata da una freccia. L'uscita da uno stato definisce la risposta dell'oggetto all'occorrenza di un evento, viene presa solo se la condizione è vera e comporta l'esecuzione delle azioni specificate.
eventi ::= evento | evento, eventi (disgiunzione)
azioni ::= azione | azione, azioni (sequenza)



Un **evento**

- è l'occorrenza di un fenomeno collocato nel tempo e nello spazio
- occorre istantaneamente
- se è modellato ha delle conseguenze
- viene ignorato se arriva in uno stato che non ha transizioni
- ammesso il non determinismo: un evento può dar luogo a più transizioni. (se più transizioni escono dallo stesso stato ne viene scelta una non deterministicamente)

Tipi di evento:

- **Operazione** o segnale op(a:T)
  Transizione abilitata **quando l'oggetto** (in quello stato) **riceve una chiamata di metodo o un segnale** con parametri *a* e tipo *T*. I parametri sono opzionali
- **Evento di Variazione** when(exp)
  Transizione abilitata **appena l'espressione diventa vera**. Può indicare un tempo assoluto o una condizione su variabili.
- **Evento Temporale** after(time)
  Transizione abilitata **dopo che l'oggetto è stato fermo *time* in quello stato**.

Transizione interna: **risposta** ad un evento **che causa solo l'esecuzione di azioni**.
Attività interna (do activity): **eseguita in modo continuato mentre l'oggetto si trova in quello stato** (senza necessità di un evento scatenante). Al contrario di tutte le altre azioni, che sono atomiche, **questa consuma del tempo e può essere interrotta**.

Il diagramma di macchina a stati parla **di evoluzione nel tempo**.
Il diagramma di attività parla **di una serie di azioni da eseguire.**

Diagrammi di sequenza
Per il **modello dinamico**, si usano per **descrivere scambi di informazioni e messaggi fra i componenti** del sistema, **organizzati in sequenza temporale**.
Come sono fatti?
Gli oggetti partecipanti allo scambio di messaggi sono rappresentati con **linee di vita** formate da **un rettangolo** (ruolo nell'interazione e/o tipo dell'oggetto) e una **linea verticale** che è proprio la linea di vita. La linea di vita è tratteggiata quando l'oggetto è inattivo, continua e doppia quando è attivo.
Le linee di vita **rappresentano lo scorrere del tempo perché scambio messaggi è una sequenza cronologica**.

I **messaggi** possono essere

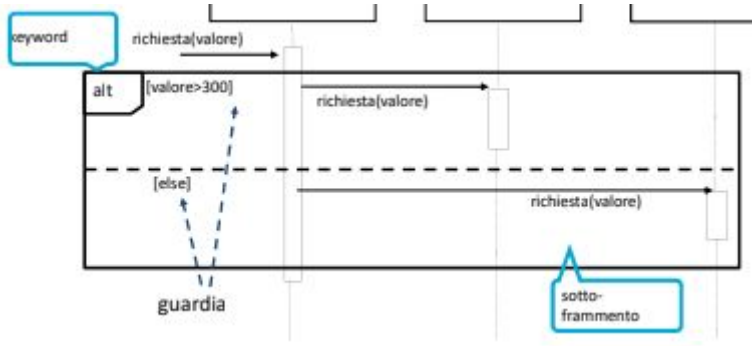
- **sincroni** cioè l'oggetto che ha spedito rimane in attesa di una risposta, congelato rappresentati con la freccia piena
- **di return** (opzionali)
- **asincroni**
    - eventualmente con esplicito consumo di tempo

Nella semantica UML **il messaggio è istantaneo**. Se voglio dire che quel messaggio impiega un tot di tempo o deve impiegare meno di x secondi ho possibilità di indicarlo: il messaggio che impiega tempo ha la linea e la freccia rivolta verso il basso (parte dal momento in cui parte, arriva nel momento in cui arriva quindi più in basso perché il tempo scorre dall'alto in basso)
I messaggi **rappresentano**

- invocazione di **operazione**
- **segnali**

attributo = **nomeMessaggio**(arg1, arg2, ...) : valore di ritorno
solo il *nomeMessaggio* è obbligatorio
Alcuni partecipanti possono essere aggiunti **dinamicamente all'interazione** oppure **cancellati**.

Frame condizionale



Senza guardia = [true]
Più guardie vere = viene scelta una via in maniera non-deterministca
Tutte le guardie false = **frame saltato**

Frame iterativo

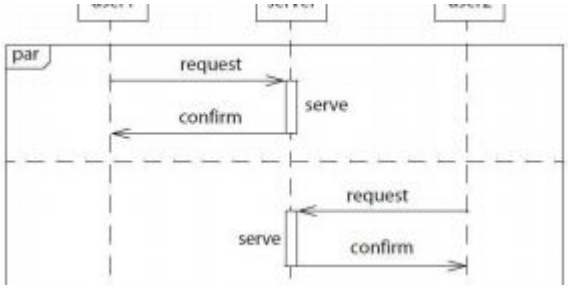
Si itera **almeno *min* volte e non più di *max* volte indipendentemente dal valore della condizione**.
Per le prime *min* volte eseguo la sequenza **senza guardare la guardia**, **tra *min* e *max* si valuta la guardia ad ogni passaggio**, esce se la guardia è falsa o se *max* volte eseguite.

loop(0, *) [condizione] while
loop(1, *) [condizione] do-while
loop(n, n) for

Frame opzionale

Identico al frame condizionale ma privo del "ramo else", cioè **le interazioni contenute vengono eseguite solo se la guardia è vera**, altrimenti si salta il frame.

Frame parallelo

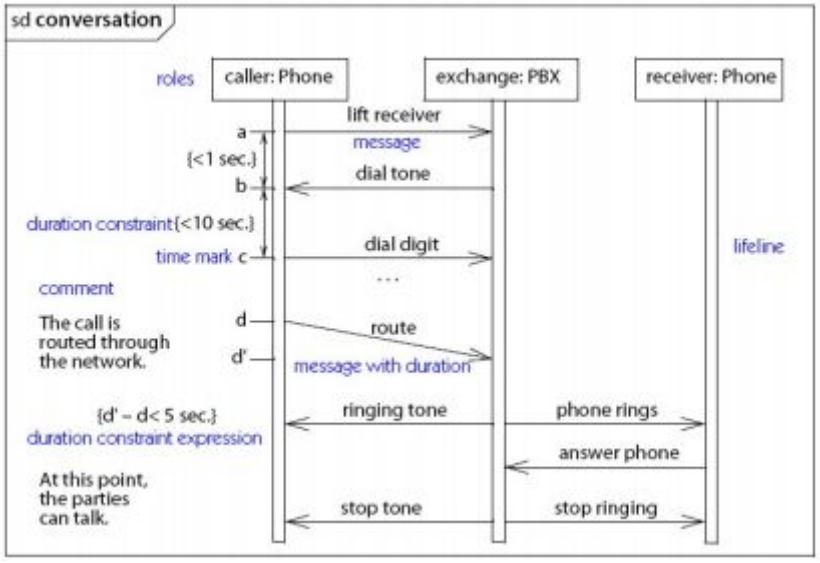


Le **interazioni contenute nei sottoframmenti** vengono **eseguite in parallelo**. Semantica a interleaving.
Nell'esempio, le richieste possono arrivare in un momento qualsiasi.

Inclusione di interazione

Inclusione di una interazione definita altrove

Vincoli di durata



Posso indicare **vincoli temporali qualitativi ma anche quantitativi**, ad esempio quanto tempo passa da quando alzo la cornetta al segnale della centrale.

Gates

Punto sul bordo del diagramma a cui è collegato un messaggio, in ingresso o uscita (il nome del gate è quello del messaggio)