

# INGEGNERIA DEL SOFTWARE

Le economie di tutte le nazioni sviluppate sono dipendenti da un software. Molti sistemi sono controllati da software.

## Cos'è l'ingegneria del software?

L'**ingegneria del software (IS)** è interessata a *teorie, metodi e strumenti per lo sviluppo di software professionali*. La spesa in software rappresenta una frazione significativa dell'economia in tutti i paesi sviluppati (molto più alta che in hardware).

La parola **ingegneria** sta ad indicare un'*applicazione* che ha a che fare con le caratteristiche che affrontano i problemi di ingegneria: si tratta di *costruire/progettare* qualcosa di complesso (a partire dalla progettazione, approccio al problema da vari ambiti, documentazione,...).

La parola **software** invece sta ad indicare il *motore trainante* di applicazioni (anche in ambito economico, culturale e scientifico).

**DEFINIZIONE di INGEGNERIA DEL SOFTWARE:** studio di metodologie, teorie e strumenti che servono a supportare e realizzare lo sviluppo di software professionali.

Devono garantire qualità diverse in ambiti differenti. Vanno oltre un approccio artigianale, il quale non è **scalabile** (= *soluzione che, una volta che la trovo realizzabile ed eseguibile per una certa dimensione, posso usarla per una dimensione superiore*).

Il software deve avere quindi queste **caratteristiche**:

- Software professionale
- Uso di risorse commisurato al problema in esame (uso di criteri di specificità)

I **prodotti software** si possono dividere in:

- 1) **Prodotti software generici**: sistemi operativi, Office, videogiochi,... Sono sistemi autonomi commercializzati e venduti a chiunque li voglia comprare. È lo sviluppatore a decidere cosa deve fare il software e a scegliere di modificarlo.
- 2) **Prodotti software specifici/personalizzati**: applicativi per gestire un reparto di terapia intensiva, ... Sono sistemi commissionati da uno specifico utente per andare incontro alle sue necessità. Le modifiche del software vengono decise tra utente e sviluppatore.

La differenza sta in chi li richiede e le caratteristiche richieste dall'utente. Per i software specifici bisogna stare attenti all'utilizzatore e a come devono essere caratterizzati.

**SUP** → software per la gestione dei sistemi di un'azienda (via di mezzo tra software generici e specifici). I software generici possono trasformarsi in specifici a seconda dei requisiti di un'azienda (per non ripartire da zero col codice, questo viene riutilizzato).

## Che cos è un SOFTWARE?

Un **software** è un *programma* che svolge un qualunque compito. Ma non solo! È un insieme di programmi informatici *con* tutta la *documentazione associata* al programma (questo è ciò che distingue un software artigianale da uno professionale).

## Quali sono le CARATTERISTICHE di un buon software?

- 1) **Manutenibilità:** il software deve essere gestito su una vita medio-lunga, si devono poter fare delle modifiche al software, si devono gestire gli errori,...
- 2) **Affidabilità:** il software deve adempiere ai suoi compiti senza intaccare proprietà di sicurezza, privacy, ...
- 3) **Usabilità:** il software deve essere testato affinché sia utilizzabile anche in più contesti e non ci siano rischi di utilizzo (stesso approccio da parte di applicazioni diverse: ad esempio, il browser web).

## Quali sono le ATTIVITA' 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,... 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,...

## 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.

## Quali sono le sfide dell'ingegneria del software?

- Sicurezza
- Mantenimento dei costi di produzione sotto controllo
- Eterogeneità (nelle piattaforme, nelle richieste, nelle fasi applicative,...)

### 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, bancario,...).

I restanti costi si trovano nella **progettazione/realizzazione**.

Un'altra sfida molto importante nell'ingegneria del software è il **tempo**: nessun utente deve attendere troppo affinché i requisiti richiesti vengano soddisfatti dal software.

Il software deve avere queste ulteriori **caratteristiche**:

- **Efficienza**: utilizzo del minor numero di risorse indispensabili nel minor tempo possibile e con poca memoria.
- **Efficacia**: un software è efficace se fa quello per cui è stato progettato ,ovvero, quando il software è in grado di incontrare il successo nel soddisfare i requisiti dell'utente.

Ci sono molti tipi diversi di **sistemi software** e non esiste un insieme universale di tecniche software applicabile ad ognuno di essi. I **metodi** e gli **strumenti** dell'ingegneria del software utilizzati dipendono dal tipo di applicazione che si sta sviluppando, dalle richieste dell'utente e dal background del team di sviluppo. Una delle caratteristiche richieste ad un buon sviluppatore è quella di saper muoversi da uno strumento all'altro in tempi ragionevoli.

I vari **tipi di sistemi software** sono:

- 1) **Sistemi stand-alone**: sistemi eseguiti su un computer locale; includono tutte le loro funzionalità necessarie e non hanno la necessità di essere connessi alla rete (sistemi in disuso).
- 2) **Applicazioni interattive basate su transazioni**: applicazioni eseguite su un computer remoto e accessibili dagli utenti dai loro terminali o PC. Si basano su scambi di informazioni per cui vale il principio di "o tutto o niente" (o la transazione va a buon fine completamente o viene annullata). Un esempio sono i siti e-commerce, le applicazioni bancarie,...
- 3) **Sistemi embedded**: sistemi di controllo software che controllano gli accessi degli utenti dai loro terminali. Sono sistemi software innestati dentro apparecchi di varia natura (es. bancomat, navigatore, ...). Di solito sono basati su web, ma non necessariamente.
- 4) **Applicazioni batch**: devono elaborare dati in grossi gruppi (es. stampare e produrre il saldo di tutti i clienti di una banca, elaborazione degli stipendi,...). Processano grandi numeri di dati input individuali per creare i corrispondenti output.
- 5) **Sistemi di intrattenimento**: sistemi di uso personale per divertire l'utente: ad esempio, videogiochi, film on demand, applicazioni di scommesse, ...
- 6) **Applicazioni di raccolta dati**: applicazioni il cui scopo principale è raccogliere dati in un unico punto, ad esempio da dei sensori (es. sito Arpav,...)

- 7) **Sistemi di sistemi:** sistemi software non indipendenti composti da un numero di altri sistemi software (software che si parlano). Ad esempio, Amazon (basato su transazioni, raccolta di dati, intrattenimento,...)

## **FONDAMENTI DELL'INGEGNERIA DEL SOFTWARE**

Le **caratteristiche fondamentali** nello **sviluppo del software** sono:

- **Metodologia/processo di sviluppo:** il modo in cui viene progettato un software deve avere un percorso ben definito di sviluppo. Devo avere già un'idea degli strumenti, dei passi da seguire,... prima di iniziare. Sono delle caratteristiche fondamentali delle metodologie adottabili.
- **Affidabilità e prestazioni:** caratteristica comune a tutti i sistemi software, i quali devono garantire i servizi richiesti dall'utente ed essere prestanti.
- **Specifiche e requisiti:** i requisiti devono essere opportunamente compresi ed essere integrati nel sistema che si sta creando.
  - **Stakeholders:** gli utenti di un sistema sono vari e utilizzano in maniera differente il sistema.
- **Riuso del software:** esistono dei componenti software creati non solo per un'unica applicazione: ci sono delle parti di codice riutilizzate in diversi software (si parla di *semilavorati* ) in maniera opportuna ai requisiti forniti dall'utente (es. sistemi web,...)

## **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 SW:** come si regola la proprietà intellettuale (brevetti, diritti d'autore, licenze, ...). Un ingegnere del software non può essere impreparato in questo ambito.
- **Uso scorretto del SW:** 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.

### **DILEMMI ETICI**

- **Non accordo con le politiche aziendali:** Il datore di lavoro si comporta in modo non etico e rilascia un sistema di "safety-critical" senza finire il collaudo del sistema, oppure fa delle richieste all'ingegnere software che non sono eticamente corrette. Si può proporre una soluzione alternativa per più volte e poi, se il datore continua a fare richieste sbagliate, cambio azienda.
- **Dipendenti non si comportano in modo responsabile:** sto sviluppando sistemi critici dal punto di vista della sicurezza e scopro che alcuni dipendenti lasciano passare degli errori che possono comportare danni anche gravi. Cosa faccio? Devo riprendere e controllare questi tipi di comportamenti e garantire che tutti lavorino in maniera opportunamente corretta e responsabile. Il controllo non è della persona ma del prodotto. Devo spingere verso un clima che garantisca comportamenti di qualità.

- **Sviluppo di strumenti potenzialmente pericolosi:** sto sviluppando software di controllo per aerei militari o per impianti militari. Lo faccio per un mondo migliore perché posso evitare la produzione di armamenti atomici non controllato,... E' sempre la direzione giusta o sto armando persone che potrebbero utilizzare i software da me prodotti in maniera malevola?

### CASI DI STUDIO (USE CASE)

- 1) **Sistema di infusione di insulina:** sistema che, attraverso opportuni sensori, rileva il livello di zucchero nel sangue e rilascia l'opportuna dose di insulina, quando necessario al paziente. Questo sistema, se funzionasse male, può comportare gravi danni al paziente o addirittura alla morte. È un sistema software critico dal punto di vista del paziente, deve agire in tempo reale (compatibile con il paziente).
- 2) **Sistema di cartelle cliniche psichiatriche:** non è un sistema real time nel senso che le visite psichiatriche sono pianificate in un tempo ragionevole → non devo reagire nel tempo di un secondo alle informazioni che vengono raccolte in queste cartelle psichiatriche. Spesso i contatti col paziente non sono di persona ma per contatto telefonico (cure, terapie, contatti,...). Inoltre, nel caso di particolari situazioni, devo poter reagire immediatamente col paziente (pazienti che potrebbero far del male ad altri).
- 3) **Sistema di stazioni metereologiche in zone disabitate:** è legata al fatto di avere in un territorio potenzialmente sconfinato delle stazioni in grado di acquisire una serie di parametri legati al tempo, e devo trasmetterle via satellite ad un centro che le raccoglie e le memorizza in modo permanente. Devo inoltre avere a bordo dei software in grado di fare auto-diagnostica (cambio di batterie, guasti dei sensori,...). Anche in questo caso non è un sistema real time.

### PROCESSI PER IL SOFTWARE

Per **processi software** si intende l'insieme strutturato di attività (**processo**) richieste per lo sviluppo di un sistema software. Le **attività** sono delle azioni/operazioni fatte da un qualche insieme di persone o persone che usano software di supporto.

I **modelli di processo** sono dei modi/strutture di base per lo sviluppo di diversi software. Pur essendoci vari modelli di processo e approcci, tutti contengono queste componenti (vedi appunti prima lezione):

- Specifica
- Progettazione (o sviluppo) e implementazione
- Validazione
- Evoluzione (manutenzione)

Nei vari modelli di processo cambierà come queste diverse fasi sono coordinate e articolate, ma saranno sempre presenti.

Nella specifica di processi software possono esserci:

- Il **dettaglio dei prodotti** di ogni attività (di ogni processo di questa attività racconto qual è l'output)
- I **ruoli e le risorse** necessarie
- **Pre e post condizioni** di ogni attività (posso iniziare la fase di progettazione solo se soddisfo delle pre-condizioni; alla fine della progettazione devo aver completato tutti i diagrammi e i documenti,.. ovvero le post-condizioni)

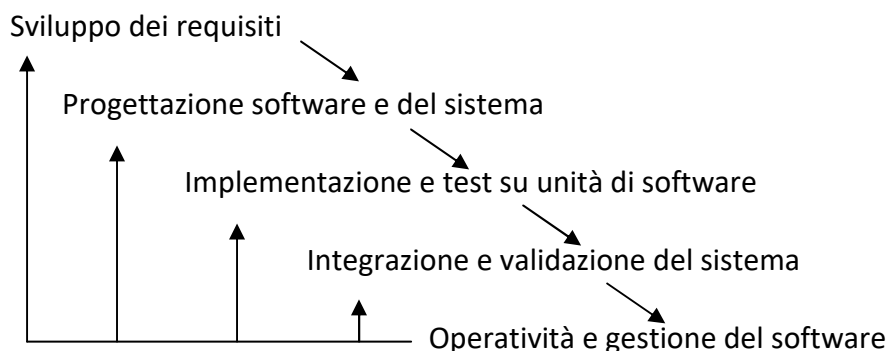
In generale, possiamo distinguere due **APPROCCI** alla definizione di processi software:

- 1) **Approccio PLAN-DRIVEN**: tutte le attività di cui si compone il progetto devono essere pianificate in anticipo. Poi, mentre eseguo le varie attività che compongono il software, verifico il proseguire del mio piano.
- 2) **Approccio AGILE**: non c'è una pianificazione iniziale del piano, ma questo viene costruito col proseguire della progettazione (in maniera incrementale) e può essere modificato passo passo osservando i risultati ottenuti in tempo reale.

In tutti i processi complessi di ingegneria del software, nella realtà, questi due approcci convivono ed esiste anche una via di mezzo. Questo tipo di approcci è ortogonale ai modelli di processi software.

### Quali sono i MODELLI DI PROCESSI SOFTWARE? (Questi approcci coesistono in tempo reale)

- 1) **Modello di sviluppo a cascata**: è il modello più rigido ed è plan-driven. Distinguo diverse fasi, una successiva all'altra, che vanno dalla specifica fino all'evoluzione. Tutte le fasi vengono considerate in modo sequenziale e per queste fasi alloco le risorse, il tempo e tutto il necessario per lo sviluppo del software. Le attività fondamentali del modello a cascata sono:



... dove ogni attività fa cadere la sua attività su quella successiva.

Letture della connessione tra le varie attività:

- Ogni attività connessa con la seconda, 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**.

Il **vantaggio** di questo modello è che, per progetti grandi, ho fin dall'inizio una chiara scomposizione dei compiti prefissata. Funziona inoltre solo quando la parte dei requisiti è assestata (astrazione non sempre vera).

Lo **svantaggio** di questo modello è che questo è un approccio rigido e diventa difficile star dietro a requisiti che vengono capiti pian piano, a star dietro a progetti che per altre condizioni esterne devono cambiare in corso d'opera (se lo applicassi in ambito medico rischio di far passare troppo tempo).

- 2) **Sviluppo incrementale**: fornire, con un approccio incrementale, delle consegne intermedie che pian piano vengono raffinate sulla base del confronto con l'utente. Sia con un approccio plan-driven sia con un approccio agile si utilizzano dei cicli e vado avanti ad aggiungere e raffinare parti nello sviluppo di software.

- Specifica
- Sviluppo
- Validazione

## DESCRIZIONE DEL PROBLEMA

Queste sono **attività** concorrenti (non c'è solo un susseguirsi delle fasi ma c'è anche un ritorno).

Rispetto ai **documenti** ci saranno:

- Versione iniziale di documenti del prodotto
- Versioni intermedie
- Versione finale

Quali sono i **vantaggi**?

- 1) Se i requisiti non sono così chiari all'inizio o se l'utente cambia i requisiti in corso d'opera, il tutto può essere fatto in modo più comodo rispetto all'approccio a cascata (*gestione "dinamica" dei requisiti*). Riesco a sopportare in modo più rapido il cambio di requisiti. Una cosa buona è mostrare un primo semi-lavorato (che deve centrare i requisiti fondamentali) di modo che l'utente sia in grado di dare delle specifiche più chiare su ciò che gli interessa.
- 2) Maggior *coinvolgimento del mittente* perché riesce a vedere passo passo lo sviluppo delle parti in modo incrementale, anche durante il processo stesso.
- 3) *Consegna più rapida del software al cliente*

Quali sono gli **svantaggi**?

- 1) Il processo non è così "visibile" come il processo a cascata (se io documentassi tutti i processi incrementali creerei una massa enorme di documenti che poi non servirebbero più perché relativi a versioni intermedie: quindi documento solo le parti più importanti. Il problema è che chi gestisce il software non vede i risultati fino alla fine del processo).
- 2) L'idea di aggiungere/modificare in modo incrementale nuove parti, dal punto di vista di qualità del software prodotto e, a volte, delle prestazioni, produce degrado del software. Potrei perdere il controllo dello sviluppo.

- 3) **Modelli basati sul riuso del software:** sempre più spesso non parto a sviluppare software da zero, ma ci sono dei semi-lavorati che possono essere usati come base di partenza (librerie, web-service,...).

Quando faccio l'analisi dei requisiti non guardo solo le richieste dell'utente ma guardo quanto le caratteristiche richieste sono d'accordo con moduli base già pre-esistenti, o magari che sono d'accordo quasi totalmente con delle piccole modifiche.

Le fasi del processo devono prevedere anche le seguenti **attività**:

- Analisi dei componenti (devo conoscere se esistono software che fanno più o meno quello a cui sono interessato)
- Modifica dei requisiti (ho più vincoli – derivati dai semi-lavorati - poiché non sto partendo da zero a realizzare il software)
- Progettazione del software(sistema) con riuso
- Sviluppo del sistema e l'integrazione con eventuali parti già esistenti



Il riuso è uno degli approcci più utilizzati al giorno d'oggi. Un esempio di riuso può essere: se ho delle librerie di classi dedicate alle analisi statistiche,... non inizio di nuovo da zero. Quali sono i **passi principali** quando vado verso il riuso?

- Specifica dei requisiti
- ↓
- Analisi dei componenti
- ↓
- Modifica dei requisiti
- ↓
- Progettazione del software con riuso
- ↓
- Sviluppo e integrazione
- ↓
- Validazione
  - Validazione componente per componente
  - Validazione del tutto

Se questo tipo di approccio lo mischio con uno sviluppo incrementale, c'è il rischio che gli utenti che hanno partecipato alla prima validazione non sono neutri nelle successive validazioni.

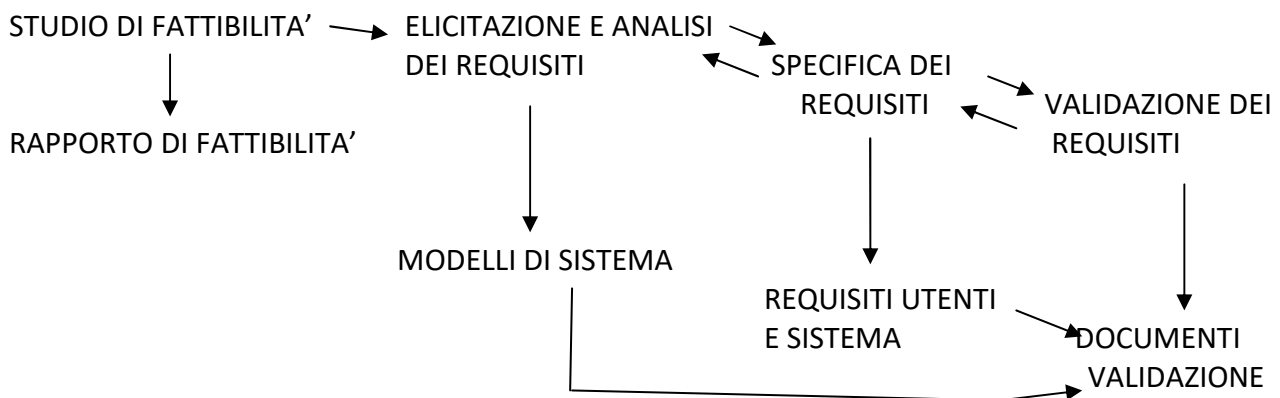
Esempi: web-service, librerie, sistemi stand-alone,...

## **ATTIVITA' DEL SOFTWARE (in dettaglio)**

### **1) SPECIFICA DEL SOFTWARE**

processo complesso che richiede varie parti:

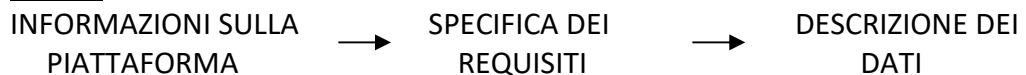
- Studio di fattibilità → quello che mi viene richiesto è affrontabile in termini di costi e di risorse? Quali risorse, tempi e costi devo prevedere per arrivare alla specifica del software?
- "Elicitazione" (sollecitazione) e analisi dei requisiti → i requisiti non sono già "pronti" ma devono essere capiti e interpretati: lo faccio sulla base di sollecitazioni da parte dell'utente (lui mi dice cosa non gli va bene usando una prima versione), utilizzo documenti (dati) che sfrutta l'utente e che fondano le varie parti del software e poi devo intervistare gli utenti (ognuno ha un suo modo di raccontare le specifiche e un modo diverso di raccontare le cose a persone diverse) e saper gestire quello che mi dicono. È difficile gestire più utenti che mi danno più specifiche tutti insieme.
- 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.



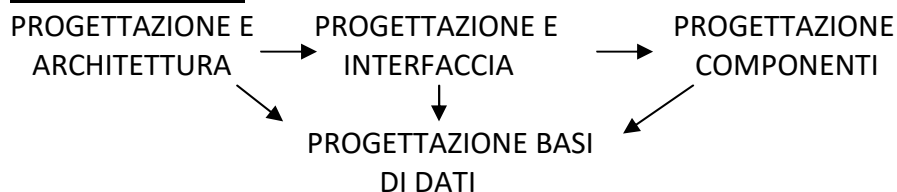
## PROGETTAZIONE E VALIDAZIONE DEL SOFTWARE

- **Progettazione del software (design):** devo progettare software per l'utente, la struttura dei dati da gestire,... Nessuno mi dice quali sono le componenti del software. Ci sono vari prodotti tra cui documenti cartacei o su supporto informatico.

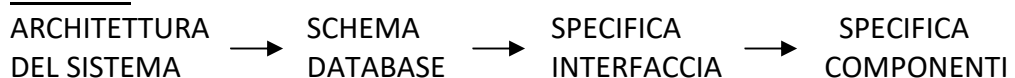
### INPUT:



### PROGETTAZIONE:



### OUTPUT:



Specifica interfaccia: specifico come i vari moduli individuati nell'architettura si devono parlare

Specifica componenti: design pattern,... servono per sviluppare sia l'architettura sia i singoli componenti. Uso di conoscenza acquisita per non sviluppare da zero cose già note.

Tra i componenti una parte fondamentale è lo sviluppo dell'**interfaccia grafica**. Interfacce non grafiche sono sempre meno diffuse.

- **Implementazione:** riguarda l'implementazione delle varie parti e poi l'integrazione del tutto.

- **Verifica (e validazione) del software:**

devo prevenire gli errori. Da una parte ci sono dei sistemi che impediscono di inserire codici senza senso, dall'altra c'è la parte di testo: vado a verificare con soluzioni reali che il software non dia problemi.

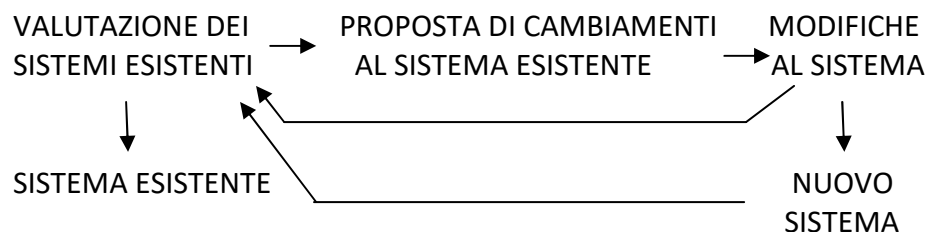


- Test dei singoli componenti: eseguito da parte dello stesso sviluppatore: faccio la verifica di ogni singolo componente. Inseriscono dati reali e anche dati interni che l'utente non vede.  
(Esempio: anno bisestile = multipli di 4, non divisibili per 100 ma per 400).
- Test complessivo del sistema: 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 reali. Non sono ancora arrivato all'utente finale.
- Test di accettazione: viene fatto su uno o più utenti finali a cui viene permesso di accedere al sistema e che effettuano una validazione complessiva non solo sui dati reali ma anche su dati che verrebbero usati in una situazione reale.

Quando vado a distribuire il software ho una situazione in cui ho una fase di BETA-TEST : questi utenti devono fare una validazione effettiva finale per trovare dei bug a loro rischio e pericolo di utilizzo ma col vantaggio di avere il software nella sua ultima versione.

- **Evoluzione del software:**

tra evoluzione e sviluppo del software non c'è quasi più alcuna differenza (velocità nello sviluppo,...). Ad esempio, il sistema universitario a livello nazionale prevedeva lauree di 4 e 5 anni fino agli anni 90 e tutto era bloccato: nessuna università poteva cambiare il nome di un insegnamento. Poi c'è stata una evoluzione immensa.



**Gestione del cambiamento:**

esistono delle query sviluppate che aiutano nella gestione dei cambiamenti. Se avvengono dei cambiamenti devo gestire due diverse voci:

- **Costo legato al fatto che devo re-implementare tutto** (devo ripartire con l'analisi dei requisiti, capire cosa vuole l'utente,... anche l'utente stesso fa parte del cambiamento).  
Si devono ridurre questi costi: in che modo?

## **TOLLERANZA**

**1\_Cerco di evitare il cambiamento:** 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.

## **PROTOTIPAZIONE**

Un modo per evitare o tollerare il cambiamento è quello di utilizzare i **prototipi**, che rispondono ai principali requisiti richiesti dall'utente. Verifico sul prototipo il software e i cambiamenti prima di passare alla versione ufficiale. Quando lo uso?

**1\_ Nella fase di elicitazione dei requisiti:** riesco a far capire all'utente cosa può fare il sistema e l'utente può dirmi cosa vuole.

PREVENZIONE  $\leftrightarrow$  PROTOTIPAZIONE

TOLLERANZA  $\leftrightarrow$  SVILUPPO INCREMENTALE

**Prototipazione** (gestione dei cambiamenti usando prototipi)

**Tolleranza** (non cerco di prevedere tutti i cambiamenti ma cerco di fare in modo che il mio software sia cambiato il più semplicemente possibile)

**PROTOTIPO:** versione iniziale del sistema che voglio sviluppare, usata per confermare le idee principali che stanno sotto il sistema da sviluppare tralasciandone altre. Nel prototipo ho una versione iniziale del sistema che fa vedere le cose più importanti, più delicate, quelle su cui si vuol improntare nuove cose. Serve per estrarre, raffinare i requisiti ed evitare ambiguità.

**Quando può essere utilizzato?**

- ✓ Nella **fase di analisi e gestione dei requisiti**
- ✓ Nella **fase di progettazione** (lo uso per vedere differenti opzioni: 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. Tenere presente che quasi mai il prototipo è completo.)
- ✓ (Nello **studio di fattibilità**: come faccio a vedere gli strumenti da usare, il tempo da impiegare,...?)

**Vantaggi:**

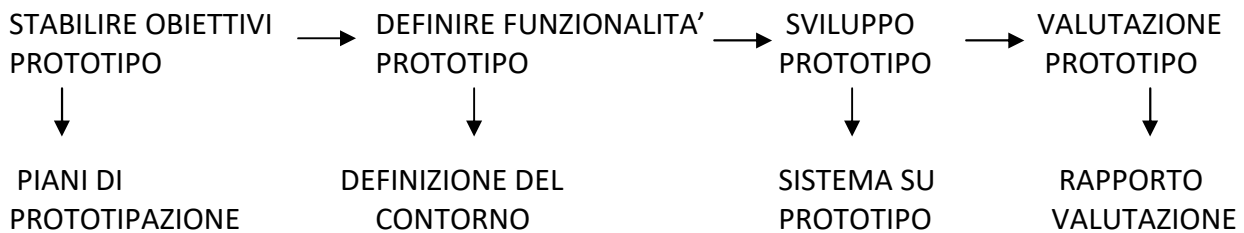
- ✓ aumenta l'usabilità del sistema
- ✓ tengo l'utente vicino allo sviluppo del progetto
- ✓ aumenta la qualità della progettazione e la manutenibilità del software
- ✓ può ridurre i costi di sviluppo

**Fasi per lo sviluppo di un prototipo:**

1\_ stabilire gli obiettivi del prototipo

2\_ definire le funzionalità del prototipo

3\_sviluppo il prototipo  
4\_valutazione prototipo



Non c'è una vera e propria fase di validazione, perdere tempo altrimenti. Deve far vedere gli obiettivi principali di ciò che voglio sviluppare e valutare questi obiettivi.

Non supporta tutte le funzionalità, deve risolvere le parti più critiche, non contiene tutte le fasi di gestione degli errori e si focalizza su **requisiti funzionali**.

### THROW-AWAY PROTOTYPES

I prototipi non vanno pensati come il primo prodotto software che poi pian piano metto a posto.

I prototipi **dovrebbero** essere abbandonati una volta che verifico che i miei test e validazioni sono andati a buon fine oppure no. Non bisogna pensare che nel prototipo ci sia la soluzione finale che poi posso scalare per produrre il software finale: si parte dai prototipi come un primo approccio al problema. Non posso partire da lì come prodotto.

#### **Perché ?**

- ❖ I requisiti non funzionali potrebbero diventare pesanti da immettere nel prototipo
- ❖ Di solito i prototipi sono minimamente o non commentati, quindi dovrei tornare indietro e commentare tutto
- ❖ In generale, i prototipi vengono sviluppati in un modo non coerente con uno standard di qualità coerente con i requisiti necessari ad un software
- ❖ Il fatto di estendere i prototipi con modifiche, aggiunte, aggiornamenti,... porta ad una degradazione delle prestazioni

### TOLLERANZA AL CAMBIAMENTO

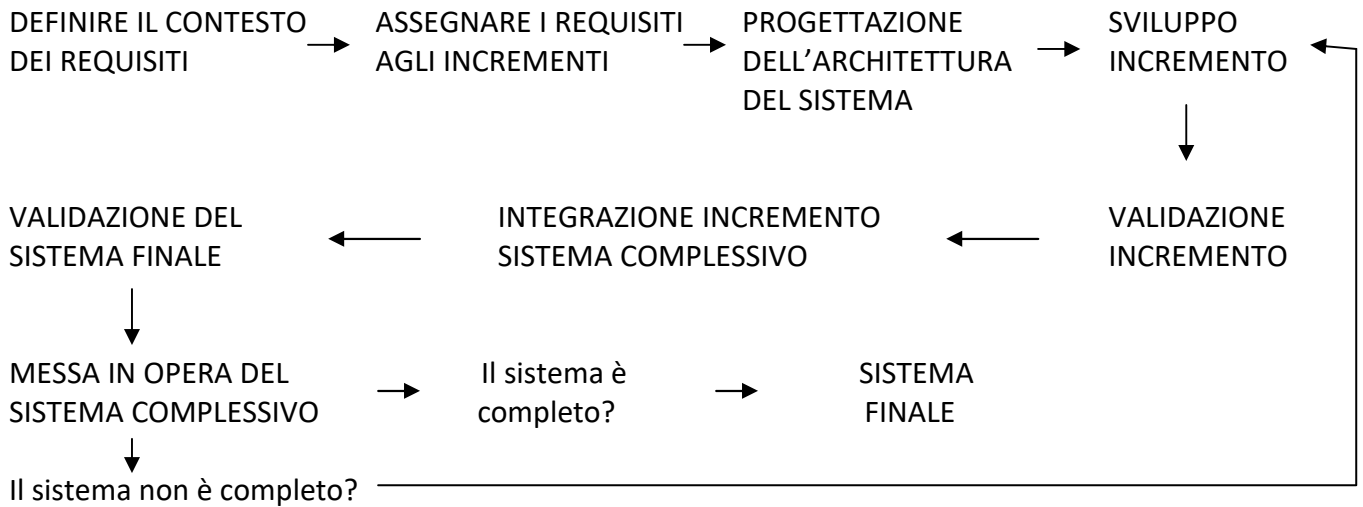
#### Sviluppo incrementale

Lo sviluppo incrementale e la conseguente consegna incrementale al cliente prevedono che io non dia lo strumento finale all'utente una volta per tutte, ma **sviluppo e consegno al cliente vari incrementi/parti del software** e, ogni parte, risponde ad un certo numero di funzionalità che il software deve supportare. **Prima consegno le parti più importanti**, poi consegno quelle meno richieste. In accordo con questo approccio, metto in ordine i vari pezzi/incrementi fin dall'analisi dei requisiti. Una volta che lo sviluppo di un incremento parte, blocco i requisiti che fanno riferimento a quell'aspetto; nel mentre, le analisi degli altri requisiti vanno avanti.

Prima di passare all'incremento successivo, faccio la **valutazione dell'incremento** appena sviluppato e consegnato. La valutazione può essere fatta dall'utente finale.

Un altro vantaggio della consegna incrementale delle parti è che i requisiti vengono **verificati in situazioni reali**. Il problema è che se devo cambiare un sistema già esistente, può essere che l'utente non sia contento di ciò che si fa.

### FASI DELLO SVILUPPO INCREMENTALE



**Vantaggio dell'utente:** le parti le verifico presto e in tempo

**Vantaggio dello sviluppatore:** grazie allo sviluppo di un incremento, vedo e mi aiuto per lo sviluppo dell'incremento successivo.

Lo svantaggio è che le parti più critiche vengono considerate e riviste tante volte.

**Svantaggi:**

- 1) Se ho sviluppato per incrementi lo sviluppo del software, e non sono in grado di capire che le parti del software che sto sviluppando fanno riferimento ad uno sviluppo base già implementato perché comune a tutti i software.
- 2) Non essere in grado di gestire la contrattazione con il cliente, ad ogni sviluppo di un incremento: questo può essere dovuto a problemi tecnici/informatici.

### OSSERVAZIONI SUL MIGLIORAMENTO DEI PROCESSI SOFTWARE

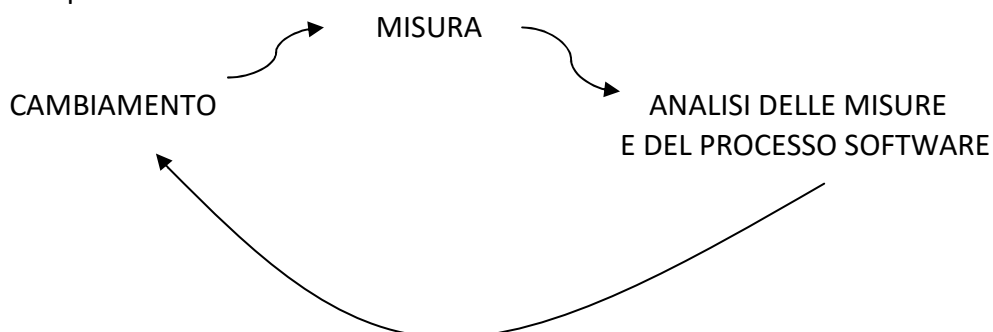
**Processo software:** insieme di attività opportunamente coordinate che soddisfano le funzionalità e l'insieme di tutte le risorse necessarie. Prevede tante persone, costi elevati.

Il miglioramento dei processi software ha lo scopo di ridurre costi e risorse. Devo essere in grado di formalizzare questi processi software.

Il miglioramento viene fatto dentro delle attività che si chiamano **PROJECT MANAGEMENT**.

Gli approcci al miglioramento dei processi software sono diversi:

- **Approcci basati sulla maturità del processo:** formalizzo il processo e poi vado ad analizzare colli di bottiglia, se ho assegnato poche risorse al processo, vado a vedere se lo posso modificare, posso spostare di ruolo degli sviluppatori che lavorano male assieme, ...
- **Approcci agili:** non c'è una pianificazione iniziale del piano, ma questo viene costruito in maniera incrementale e può essere modificato passo passo osservando i risultati ottenuti in tempo reale.



**Come cambiare?** Non ci sono regole fisse per risolvere tutte le cose. Possiamo proporre tutti i cambiamenti organizzativi più belli, ma se le persone che lavorano non sono in gamba va tutto a fondo. Ma anche se le persone sono le più brave e l'organizzazione è pessima il sistema va a fondo.

È stata fatta una certa gerarchia di livelli in base al quale i software sono supportati. Da una conoscenza implicita (livello più basso) si va su fino ad avere una conoscenza esplicita dello sviluppo; dopodiché, a seconda della capacità di un'azienda, si ha un controllo costante dei processi per cui si possono anche cambiare i processi di sviluppo del software (consapevolezza sempre maggiore).

## **SVILUPPO AGILE DI SOFTWARE**

### **Rapid Development Delivery**

Consiste in una serie di approcci legati al fatto che, chi sviluppava software, si è accorto che c'erano numerosi processi software che fallivano per eccessiva lentezza. I processi plan-driven erano sempre più in difficoltà.

È nato un nuovo approccio chiamato **sviluppo agile** per ridurre al minimo la richiesta da parte del cliente e la consegna del software.

#### **Aspetti su cui si fondano gli approcci agili:**

- ✓ Specifica, progettazione e sviluppo (implementazione) non sono più fasi distinte
- ✓ Il sistema viene sviluppato attraverso una serie di versioni
- ✓ I committenti sono coinvolti in tutte queste fasi (il rischio è che il committente diventi quasi tecnico nell'ambito del software richiesto)
- ✓ Consegna frequente di nuove versioni per la valutazione da parte dell'utente
- ✓ Strumenti ad hoc per il Rapid Development Delivery (ad esempio strumenti che supportano la validazione)
- ✓ Documentazione ridotta al minimo (il software dovrebbe "parlare da solo" grazie alla sua chiarezza). Nel mondo reale questo solleva un problema vero (sia eccedere nella documentazione, sia ridurre la documentazione può portare svantaggi). Il software deve essere sviluppato bene e con la documentazione essenziale.

Lo sviluppo plan-based può essere rappresentato in questo modo:

INGEGNERIA DEI  
REQUISITI

SPECIFICA DEI  
REQUISITI

PROGETTAZIONE E  
SVILUPPO

RICHIESTA DI  
MODIFICA DEI REQUISITI

Nell'approccio agile ho due fasi:

INGEGNERIA DEI  
REQUISITI

PROGETTAZIONE E  
SVILUPPO

### Terminologia in contrapposizione:

- Individui e interazioni  $\leftrightarrow$  Processi e strumenti
- Software che funziona  $\leftrightarrow$  Documentazione completa
- Collaborazione del cliente  $\leftrightarrow$  Negoziazione del contratto
- Rispondere al cambiamento  $\leftrightarrow$  Seguire un piano

Per gli “**approcci agile**” sono più importanti le fasi a sinistra, pur essendo importanti anche le fasi a destra.

**1\_** Se i processi e gli strumenti utilizzati sono corretti ma ho individui incapaci che sviluppano il software fallisce. Le interazioni con le persone (sviluppatori) sono importantissime ai fini di un buon sviluppo del software. In certe metodologie agili le coppie di lavoro continuano a cambiare durante lo sviluppo del progetto, per far in modo che tutti conoscano tutto il progetto e siano responsabili di ogni parte del progetto.

**2\_** E’ più importante un software che funzioni correttamente piuttosto di avere una documentazione completa. Un software che funziona e basta però non basta! Serve la documentazione.

**3\_** Se penso ad un contratto, penso a come cautelarmi rispetto al prodotto che mi viene fornito in cambio di un pagamento. Se il prodotto fornitomi non rispetta le clausole contrattuali, mi cautelo sulla base della responsabilità che il programmatore deve assumersi. Allo stesso modo anche lo sviluppatore si cautela. Questo tipo di approccio è quindi basato sulla **sfiducia reciproca**.

**4\_** Se cambia qualcosa, so adeguarmi ai cambiamenti. Seguire un piano è importante, ma è più importante sapersi adattare alle modifiche.

### **Caratteristiche principali:**

- Coinvolgimento del cliente
- Consegna incrementale
- Attenzione maggiore ai clienti
- Gestire il cambiamento ma mantenendo la semplicità

Quando va bene questo approccio? Va bene per lo sviluppo di prodotti piccoli o di media dimensione, che quindi abbia un numero di persone collaboratori ridotto.

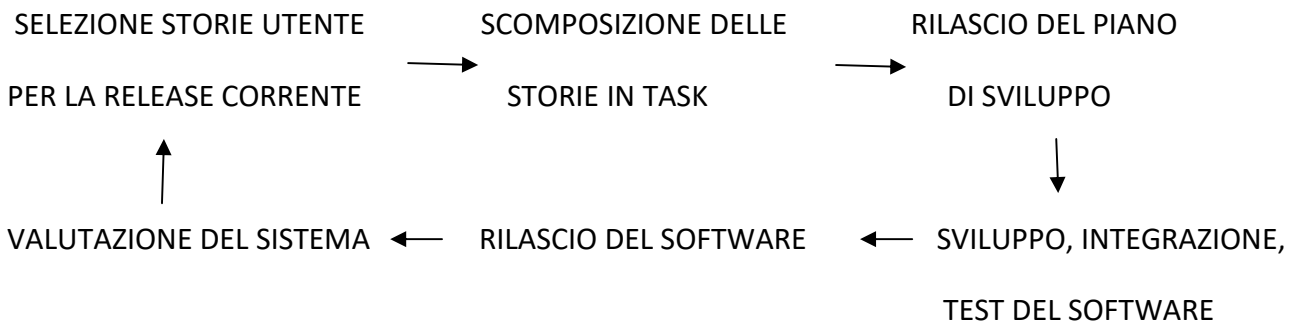
Altra cosa è che devo essere garantito e applicare questo approccio a progetti in cui il cliente è d’accordo.

### **XP (Extreme Programming)**

Tecnica sviluppata negli anni ’90 che produce un approccio estremo al processo iterativo. E’ una produzione di software molto rapida e devo far vedere gli incrementi del progetto al mio 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.



### Fasi principali:



Queste fasi hanno delle **caratteristiche**:

- **Pianificazione incrementale:** ogni volta che approccio una singola “storia” in cui ho distinto il mio sviluppo agile faccio un piccolo “piano” per lo sviluppo di quella parte. I requisiti sono sviluppati appositamente per ogni storia. Se l’utente sbaglia, gli sviluppatori non prendono la strada migliore (svantaggio).
- **Rilasci minimi:** le “storie utente” sono dei piccoli passaggi da sviluppare (release piccole).
- **Progettazione abbastanza semplice:** questo è dovuto al fatto che ho le release minime.

### TEST DRIVEN-DEVELOPMENT E REFACTORY

**Test driven-development:** una delle leggi dell’XP è l’uso dei framework per verificare i test sulle singole unità. Prima però bisogna progettare i test e poi sviluppare il modulo. Specifico tutto prima quello che deve essere testato. **A cosa serve?** Serve a semplificare tantissimo la comunicazione tra moduli diversi: 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.

**Refactory:** quando sviluppo, non è che se il codice che ho fatto funziona e va bene all’utente smetto di migliorarlo. Devo continuare a migliorarlo affinché abbia una performance migliore. Il fatto di continuare ad aggiungere pezzi, se non li integro bene migliorando le parti già implementate, perdo la funzionalità e la prestazione. Questa cosa va osservata e fatta attentamente. Il refactory è un miglioramento dell’implementazione.

- **STORIE DELL'UTENTE:** parte di specifica dei requisiti. Rispetto ad altri approcci nell'ingegneria dei requisiti, l'idea è che, per la progettazione, si parte da requisiti rappresentati in **schede** in cui l'**utente** dice cosa vuole sia fatto nello sviluppo del software. Queste storie sono definite dall'utente finale o il cliente coinvolto nella progettazione. Il **rappresentante dei committenti è partecipe nel team di sviluppo**: è lui a scrivere le storie ed è lui che sceglie l'ordine di priorità di ciò che deve essere sviluppato. Poi le storie sono suddivise in singoli **task**, e lo sviluppatore li realizza uno alla volta.
- **REFACTORING:** re-implementazione e semplificazione delle soluzioni trovate del software (prima si creano i test che le varie versioni del software dovranno passare e poi si sviluppa il software con un lavoro a coppie). A differenza di altri approcci, l'idea è che non c'è mai un fermarsi per considerare solo cose nuove. **Il team di sviluppo continua a semplificare e ottimizzare il software già esistente.** Questa cosa è fatta perché, se vado avanti con soluzioni incrementali che aggiungono dei pezzi, il rischio è che se queste aggiunte non sono armonizzate rischio di avere dei problemi. È fatta anche perché posso utilizzare in modo analogo pezzi di codice già implementati in altre parti dello stesso software.
- **SVILUPPO TEST-FIRST:** i programmatori non apprezzano **creare i test** e soprattutto crearli **prima di aver sviluppato il software** (anche perché creare i test non è facile: si deve creare dei test che possano gestire tutte le possibili casistiche). Nei test non sono **coinvolti** solo gli **sviluppatori, ma anche gli utenti finali** (utente che partecipa con gli sviluppatori). L'utente, man mano che viene coinvolto in questo processo agile, diventa un utente speciale perché ha delle conoscenze più approfondite del software. Quindi alla fine non è più un utente comune. Questo dipende ovviamente anche dal numero di utenti che richiedono il software. I test di solito sono automatizzati.
- **PROGRAMMAZIONE IN COPPIE:** 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.** Lo sviluppo di codice in coppia può essere **più veloce** e può essere un approccio **più critico e consolidato.** I requisiti per cui la programmazione in coppia funziona sono: avere lo stesso livello di competenza, apprendimento di come discutere, responsabilità comune e non singola,... Le coppie, nello sviluppo XP, non sono fisse ma cambiano continuamente: serve a completare l'idea che il software non è di una sola persona e la responsabilità è condivisa; serve inoltre per incrementare e condividere le conoscenze di uno o dell'altro. Rispetto alle competenze medie e al ruolo di coinvolgimento medio richiesto al programmatore, in questo contesto è richiesta una competenza più elevata che in altri contesti.

## METODOLOGIA AGILE SCRUM

Telefilm “Silicon Valley”: Scrum si basa sul manifesto “agile”. L’idea è organizzare il lavoro senza alienare il cliente durante la progettazione di un software.

- **TEAM:** gruppo di persone piccolo (max di 7 persone) che andrà a sviluppare il software o parte di esso.
- **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.
- **BACK LOG:** elenco degli incrementi di un progetto più o meno spezzettato. Deve essere mantenuto dagli sviluppatori e dal proprietario del progetto
- **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, tradurre le funzionalità in **user stories** dallo **scrum master**. Traduce le funzionalità espresse dal proprietario in un linguaggio più capibile dal team.
- **SPRINT:** tempo dalle 2-4 settimane in cui si vanno a prendere degli obiettivi dal back log che vanno sviluppati in quell’arco di tempo.
- **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.
- **USER STORY:** funzionalità tradotte dallo scrum master per gli sviluppatori.

Il **BACK LOG** è una serie di blocchi che definisce cosa c’è da fare nel progetto. All’inizio di ogni sprint si fa una riunione (30 min) dove lo scrum master e il team decidono quali sono le funzionalità che andranno implementate nelle successive 2-4 settimane (**INIZIO SPRINT**). In questa riunione si elencano i task da eseguire presi dal back log (**SPRINT BACK LOG**): è un sottoinsieme del back log totale. Sempre in questa fase il team assegna una stima di risorse (ore o minuti di lavoro): la stima deve andare da un minimo a un massimo.

Fatto questo c’è la fase di lavoro detta **SPRINT** (2-4 settimane). La cosa interessante è che ci sarà una riunione di 10 minuti ogni giorno a inizio giornata, possibilmente in piedi, chiamata **SCRUM MEETING**, 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**).

Alla fine 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.

Al termine di queste operazioni ci sarà un ulteriore meeting, chiamato **MEETING DI FINE SPRINT**, 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,...

## FARMACOLOGIA

Sezione dell'Università che si occupa di progettare software per il Ministero della Salute per sorvegliare il mercato dei farmaci in commercio.

Si fanno dei test su animali e persone sane. Si sorvegliano i farmaci sul mercato utilizzando una tecnica usata **segnalazione spontanea**: medici, farmacisti,... andranno a segnalare spontaneamente i vari casi anomali che gli si sono presentati davanti. Poi, con l'azienda produttrice, si discute del problema creato dal farmaco e si cercano soluzioni. Il gruppo di farmaco - sorveglianza ha creato un progetto, per lo sviluppo di software per il Ministero, per la segnalazione di queste possibili reazioni avverse.

Hanno creato tre software principali:

- **VigiSegn**: raccoglie l'analisi dei dati della rete nazionale di Farmacosorveglianza
- **VigiFarmaco**: raccoglie tutte le segnalazioni spontanee. Si è introdotto scrum in questo software osservando dei miglioramenti notevoli.

### **Sistema di versionamento del codice basato su GIT:**

ognuno ha sul proprio PC pezzi di codice personali. Ogni pezzo può essere **committato (salvato)** su questo sistema (punto di ripristino: si può tornare a quel punto quando si vuole). Ognuno ha il proprio **repository in locale** e si può comunicare con tutti gli altri sviluppatori con una **repository in remoto** (ORIGIN). All'interno di GIT può essere implementato un framework che permette di sviluppare in armonia del codice attraverso tante persone.

Concetto di **branch**: il codice può essere committato dentro un ramo. Se si vuole implementare una nuova cosa, si può creare un nuovo ramo per non disturbare gli altri o rovinare il codice di altri.

Esempi di software per implementare SCRUM:

- [www.sv-scrum-board.herokuapp.com](http://www.sv-scrum-board.herokuapp.com) → le colonne hanno un nome diverso dalla terminologia introdotta.
- [www.trello.com/b/d34x3hp3/scrum](http://www.trello.com/b/d34x3hp3/scrum) → gratuito, con accesso di credenziali. Ci sono le schede delle terminologie spiegate prima: back-log, sprint,...  
Esempio di back-log: web server, dbms, utenti, ruoli,...  
Esempio di sprint: web server (primo task da implementare), utenti  
Esempio di In Progress: web server
- [www.vivifyscrum.com/#/boards/13189/product-backlog](http://www.vivifyscrum.com/#/boards/13189/product-backlog) → gratuito fino a un max di 3 utenti e un progetto per utente. È un sistema molto più ricco perché si può aggiungere un task, assegnare un utente, decidere quanto tempo serve per l'implementazione, ... Lo sprint può essere schedato: funzione di inizio sprint che permetterà di generare lo sprint back log (dove sono raccolti i task selezionati). Esiste un grafico di una curva ideale che rappresenta quanto tempo serve per completare il task. Permette di tenere traccia del tempo lavorato. I task possono essere raggruppati.
- [www.cpg.myjetbrains.com](http://www.cpg.myjetbrains.com) → a pagamento e completo. Integrazione con GitHub.

## “SCALARE” APPROCCI AGILI

I progetti “agili” sono progetti medio - piccoli dove gli sviluppatori risiedono nello stesso luogo. Questo è un problema quando dobbiamo “scalare” poiché una delle caratteristiche degli approcci agili è che c’è pochissima documentazione pur essendo più veloci e tutti gli sviluppatori comunicano tra loro.

Quando faccio questi tipi di “**scaling**” voglio mantenere le caratteristiche dei progetti agili: sviluppo con test, planning flessibile, sviluppo incrementale con release frequenti, re factoring continuo.

## SCALING UP

**Cosa succede quando tento di “scalare” ,con un approccio agile, per progetti più ampi, più lunghi (durata) e con più team di sviluppo distribuiti?**

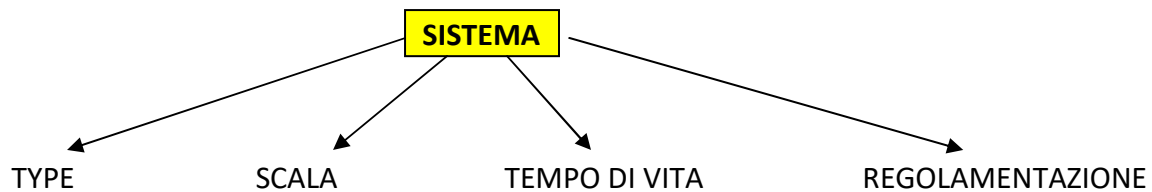
## SCALING OUT

**Cosa succede quando, in questi progetti, voglio introdurre un approccio agile dentro organizzazioni che non hanno familiarità con questo tipo di approccio?**

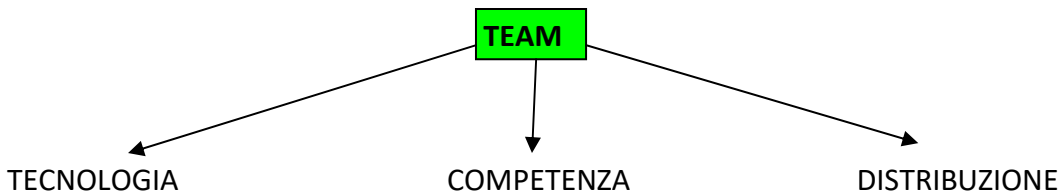
### **Problemi pratici:**

- Quando sviluppo con tecniche agili, utilizzo approcci informali. Questo è un problema, ad esempio, quando devo introdurre un cliente terzo. Ogni volta che formalizzo, pago un dazio ma ci devo guadagnare dall’altra parte.
- Se vado verso software più complessi, devo capire come armonizzare i software più complessi più che per lo sviluppo per il mantenimento, la correzione di errori,...
- I team di sviluppo sono distribuiti, quindi dobbiamo capire come, grazie a delle tecnologie, si debba gestire il tutto (es. Skype).
- Aspetti contrattuali: è difficile che in un contratto si dica che si avanzi in modo incrementale finché non c’è qualcosa che soddisfa l’utente finale. Nel contratto si scrivono le specifiche del risultato finale dettagliate (questo non va d’accordo con lo sviluppo agile perché non rispetta eventuali dinamiche lungo il processo: eventuali modifiche,...). Converrebbe quindi pagare a ore di sviluppo.
- Manutenzione rispetto allo sviluppo: l’approccio agile, nella manutenzione, è vantaggioso per l’attenzione al re-factoring ma è svantaggioso perché la **documentazione è pochissima**. Se il codice non “parla” al posto della documentazione in modo corretto si incorre in problemi nella manutenzione, che non è più scontata. Nei processi agili assumo che gli sviluppatori siano sempre gli stessi dall’inizio alla fine, ma non è detto che sia così (**cambio del team di sviluppo**). Se un nuovo team prende in mano il software, deve capire tutte le parti del codice.
- Quando si scala nei progetti più grossi, non c’è un’unica categoria di utenti (**tante categorie di utenti**): questo implica punti di vista differenti. Non posso utilizzare un rappresentante per ogni categoria di utenti!! Sarebbe troppo difficile armonizzare il tutto.

Una soluzione che vada bene universalmente in questi aspetti non esiste.



Più il tempo di vita è lungo, più è necessaria documentazione.



Negli approcci agili servono una tecnologia e una competenza maggiori. Il ruolo di un team di un processo agile è un po' più creativo degli sviluppatori in un ambito più organizzato, con più persone,... Se i team sono distribuiti geograficamente diventa più difficile farli interagire con approcci agili.



## **MODELLAZIONE DEI SISTEMI SOFTWARE**

La modellazione di sistema ha a che fare con una **rappresentazione astratta** delle caratteristiche software che sto progettando o che ho già progettato. Mi danno una prima struttura scheletrica del software che sto andando a sviluppare oppure mi danno una sorta di documentazione del software che ho già sviluppato o che è stato sviluppato da altri.

Posso rappresentare diverse caratteristiche astratte a seconda dei punti di vista da cui mi metto.

- **Prospettiva esterna:** riguarda la rappresentazione del contesto in cui il sistema software andrà a lavorare (ad esempio, come si rapporta con sistemi esterni o con sistemi già esistenti). Esplico il tipo di interazione del mio sistema con altri sistemi.
- **Prospettiva sull'interazione:** riguarda la rappresentazione di due tipi di interazione: come il sistema interagisce con l'esterno e come il sistema, nelle sue componenti, interagisce una componente con l'altra.
- **Prospettiva strutturale:** riguarda la rappresentazione della struttura e delle componenti di un sistema e come sono connesse l'una con l'altra. Riguarda principalmente i dati.
- **Prospettiva comportamentale:** riguarda la rappresentazione della dinamica del mio sistema (come evolve e come si comporta nel suo funzionamento dinamico). Riguarda principalmente i processi e l'esecuzione che utilizza i dati per fare qualcosa (dinamica del software).

Queste prospettive vengono utilizzate in un...

## **APPROCCIO COMPLESSIVO ORIENTATO AGLI OGGETTI (OO)**

**UML** = Unified Modeling Language → insieme di strumenti informali per supportare la modellazione delle parti appena elencate.

## **MODELLAZIONE DI SISTEMI SOFTWARE**

29/03/17

- Descrizione
- Progettazione

Di solito questi approcci alla modellazione sono formalismi basati su diagrammi. Questi formalismi vengono usati dall'analisi dei requisiti fino alla fine della progettazione (la progettazione di software non è fatta con un unico formalismo o un unico livello di astrazione: si va per raffinamenti successivi. Ad alto livello la progettazione sarà qualcosa di simile all'analisi dei requisiti un po' più ad alto livello, e poi si approfondiscono i dettagli scendendo di livello).

**Esempio di formalismo:** diagramma delle classi UML (poche cose obbligatorie ma tante che posso aggiungere fino al dettaglio estremo).

Qualunque sia l'approccio, quando parto dall'analisi dei requisiti ho dei formalismi più espressivi (servono di più per comunicare con gli stakeholder) più scendo di sensibilità ma divento più vicino allo strumento software.

Questi strumenti formali si preoccupano di descrivere e fornire un'astrazione di quello che devo progettare o di quello che c'è secondo diversi **punti di vista**:

- **Prospettiva esterna:** faccio capire, con determinati formalismi, il confine del sistema che sto progettando (chi sono le entità fuori dal sistema e come interagiscono)
- **Prospettiva legata all'interazione:** l'interazione può essere tra l'esterno e il mio sistema oppure tra le diverse componenti del sistema
- **Prospettiva strutturale:** non dico come interagiscono tra loro le diverse parti, ma descrivo la struttura di queste parti e la struttura del software che implementa le varie parti (per la parte di struttura si parla anche di *schema di dati*)
- **Prospettiva comportamentale (o dinamica):** devo specificare come il mio sistema software evolve nel tempo a fronte di interazioni con l'utente o a fronte di cambiamenti che comunicano col modulo che sto considerando

Queste sono le parti che devo avere quando descrivo come è fatto un software e come è strutturato.

A livello di **modellazione** ci sono due potenziali significati:

- 1) Modello  $\leftrightarrow$  insieme di concetti del formalismo utilizzato (decido di utilizzare un approccio completamente orientato agli oggetti: suppongo di usare il formalismo di diagramma delle classi. Il modello è la spiegazione dei concetti per descrivere il sistema)
- 2) Modello  $\leftrightarrow$  risultato dell'applicazione di questi strumenti (dei concetti del formalismo)

Rispetto a questa distinzione dei vari formalismi, bisogna introdurre ...

## UML (UNIFIED MODELLING LANGUAGE)

Insieme di formalismi proposto come risultato di sintesi di vari strumenti metodologici per la creazione di software. Sono tutti strumenti formali basati su diagrammi e sono per software **orientati agli oggetti**.

UML propone diversi tipi di **diagrammi (DA FARE PER L'ESAME!!!!)**:

- **Diagrammi di attività:** diagrammi che permettono di raccontare quelle attività coinvolte nell'uso del software che stiamo sviluppando oppure cosa mi permette di fare il software.
- **Use case diagrams:** i casi d'uso possono essere usati nella fase di ingegneria dei requisiti e caratterizza sia l'esterno sia le parti fondamentali del mio software ad altissimo livello (con diagrammi che mostro agli utenti). Non racconto la parte legata al tempo (**CASI D'USO PIU' IMPORTANTI DA FARE**).
- **Sequence diagrams:** sono usati a due livelli diversi e possono supportare il racconto della sequenza di operazioni nel tempo tra l'esterno e il sistema nel suo complesso oppure le interazioni delle componenti principali del sistema stesso.
- **Class diagrams:** dice com'è strutturato il mio sistema software: quali sono le componenti software nella sua **struttura** e come sono fatte le varie componenti nella loro struttura. È usato anche per raccontare il significato e il senso di tutti gli altri costrutti di tutti gli altri formalismi. È il diagramma più legato all'orientamento agli oggetti.
- **Object diagrams:** rappresentano com'è lo stato, o una parte di essa, del mio sistema software in un certo momento (rappresenta la **singola istanza**).

Possono essere usati in **due modi**:

- Facilitare la **discussione con tutti i sistemi già esistenti** (grazie ad una rappresentazione astratta delle parti più importanti e su cui voglio concentrarmi) e facilitare l'interazione con i clienti
- Facilitare la **documentazione di software** già esistenti



Questi modelli, usati nei vari contesti, possono essere incompleti (una delle caratteristiche di tutti i sistemi UML: tante cose facoltative e poche obbligatorie).



## FORMALISMI

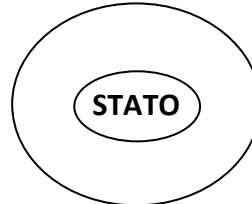
### 1) CLASS DIAGRAMS:

#### ✓ Concetti di orientamento agli oggetti

**Oggetto** = descrizione di un qualunque concetto, oggetto fisico, pezzo di informazione che voglio rappresentare nel mio sistema software, ed è un particolare concetto.

È caratterizzato da:

- **Object id:** *identificatore dell'oggetto* che ha varie caratteristiche: non è in mano a chi ha definito o istanziato l'oggetto ma è *gestito dal sistema software* che capisce questi oggetti. Un object id già utilizzato non viene re-utilizzato. Non è in mano all'applicazione o al programmatore ma è gestito dal sistema OO. Il valore, o **stato dell'oggetto**, è dato dal valore delle sue entità o relazioni con altri oggetti. Si può identificare l'oggetto anche senza ricorrere allo stato dell'oggetto stesso.
- **Attributi + relazioni con altri oggetti:** descritti dallo stato dell'oggetto (es. data di nascita,...). Gli attributi sono una parte privata dell'oggetto. Questo approccio si chiama **ENCAPSULATION**: posso vedere come è fatto il mio oggetto da uno stato interno e, dei metodi da utilizzare, posso vedere solo la loro "**signature**" ma non so niente della sua implementazione. È utile come interfaccia dell'oggetto.



- **Metodi (o interfaccia dell'oggetto):** raccontano la parte pubblica o comportamentale. Spiegano come posso fare *esternamente* per interagire con l'oggetto (operazioni che posso svolgere sull'oggetto/a che cosa risponde l'oggetto).

Gli oggetti sono descritti da un **TIPO** (concetto **INTENSIONALE**) e sono raccolti in **CLASSI** (concetto **ESTENSIONALE**).

Il tipo è la rappresentazione della struttura che accomuna gli oggetti con caratteristiche analoghe. Gli oggetti sono **istanze** di un tipo. Intensionale vuol dire che in una sola volta racconto caratteristiche comuni a tanti oggetti. La parte estensionale è che, quando ho oggetti dello stesso tipo, faccio riferimento al termine di classe (= insieme di oggetti dello stesso tipo).

Tipo → struttura (es. CREATE TABLE Studente → struttura della tabella)

Classe → insieme degli oggetti con quella struttura (es. SELECT \* FROM Studente → cercami tutte le righe, oggetti dello stesso tipo, che stanno nella tabella studente)

All'interno del concetto di classe si definisce anche **l'implementazione**: potrei avere classi diverse con implementazioni diverse dello stesso tipo (cambia il codice dei metodi, ma il tipo è sempre lo stesso con tante implementazioni attaccate).

L'**insieme dei tipi**, nei sistemi orientati agli oggetti, non è predefinito. Posso definire meccanismi che baso su due componenti diversi:

**1\_Tipi atomici:** stringhe, interi, numerici, double, data, booleani, object id (è atomico perché gestisco la struttura della relazione tra oggetti grazie all'object id).

**2\_Costruttori di tipo:** tipi che mi permettono di costruire altri tipi più complessi: record\_Of\_Tuple, set\_Of, list\_Of, bag\_Of (o multi\_Set → insiemi degeneri dove un elemento dell'insieme può appartenere all'insieme più di una volta),...

#### Metodi:

**1\_Costruttori:** servono a istanziare nuovi oggetti

**2\_Distruttori:** servono per cancellare degli oggetti

**3\_Accessori:** servono per accedere in vario modo e per leggere parte dello stato

**4\_Trasformatori:** servono per cambiare valori dello stato in un oggetto

#### Oggetti:

**1\_Identità:** un oggetto è identico solo a sé stesso. Nuovi oggetti sono identici se hanno lo stesso object id.

**2\_Uguaglianza:** due oggetti sono uguali ...

**A\_ Superficiale:** ... a livello superficiale se hanno gli stessi valori degli attributi che compongono il loro stato

< OID1 [a:10, b:25, c:OID2]>

<OID2 [f:20, h:15]>

<OID3 [a:10, b:25, c:OID2]>

L'uguaglianza superficiale indica l'uguaglianza profonda ma non viceversa.

**B\_ Profonda:** ... a livello profondo, se la rappresentazione dello stato la ottengo andando a cercare tutti gli attributi atomici che in un qualche modo fanno parte dello stato.

< OID4 [a:10, b:25, c:OID5]>

< OID5 [f:20, h:15]

OID5 e OID2 hanno lo stesso stato. OID1 e OID 4 non hanno lo stato uguale, ma se cambio l'OID dell'oggetto c diventano uguali.

Il fatto di avere oggetti esattamente con lo stesso stato non è assolutamente un problema: lo stato non è identificativo di niente. Posso avere migliaia di oggetti diversi che condividono lo stesso stato.

#### RIASSUNTO:

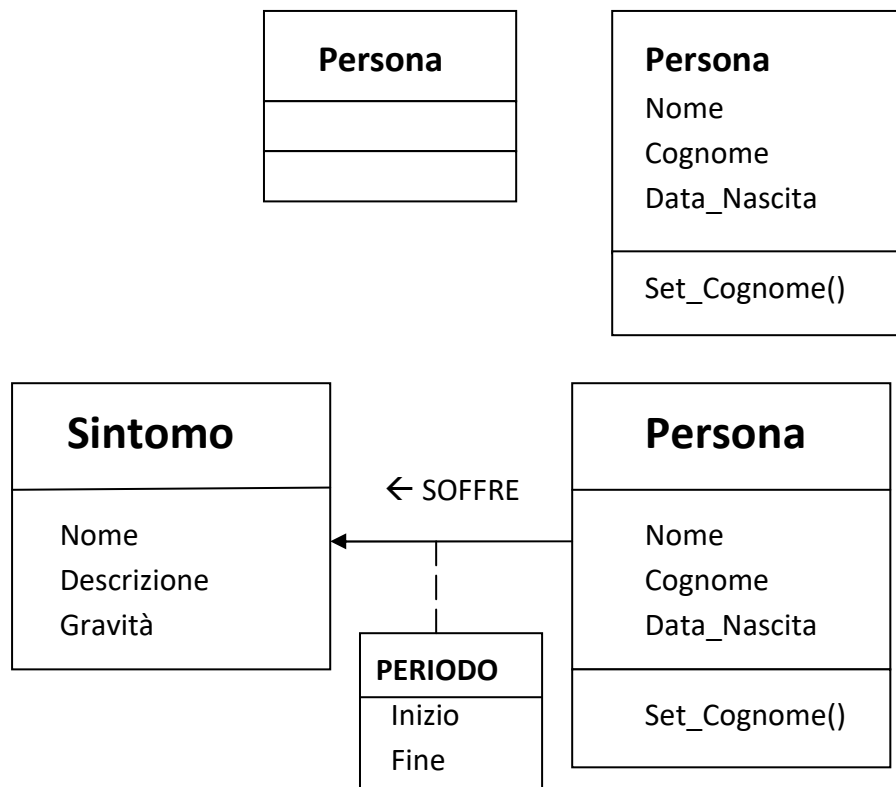


## RAPPRESENTAZIONE IN UML

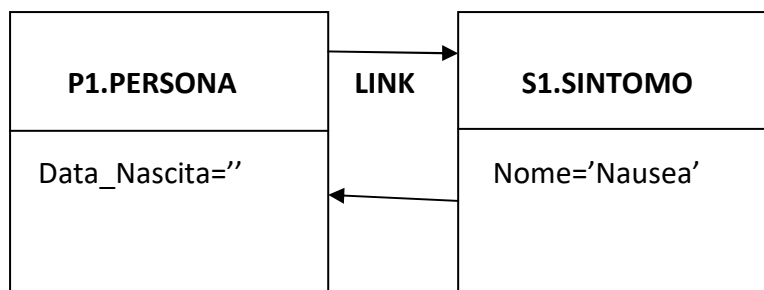
- ✓ **Dettagli sul Class Diagram:** riserva il termine **classe** per specificare un tipo particolare. Sono una rappresentazione per la struttura/schema degli oggetti di cui rappresento la parte comune di struttura.

**CLASS** → specifica di un tipo (usare lo stile **UPPER CAMEL CASE**: uso le maiuscole per descrivere le varie parole che descrivono il nome di un tipo. Ad esempio: *StudenteTriennale*).

Per i nomi di attributi si usa il **LOWER CAMEL CASE**.



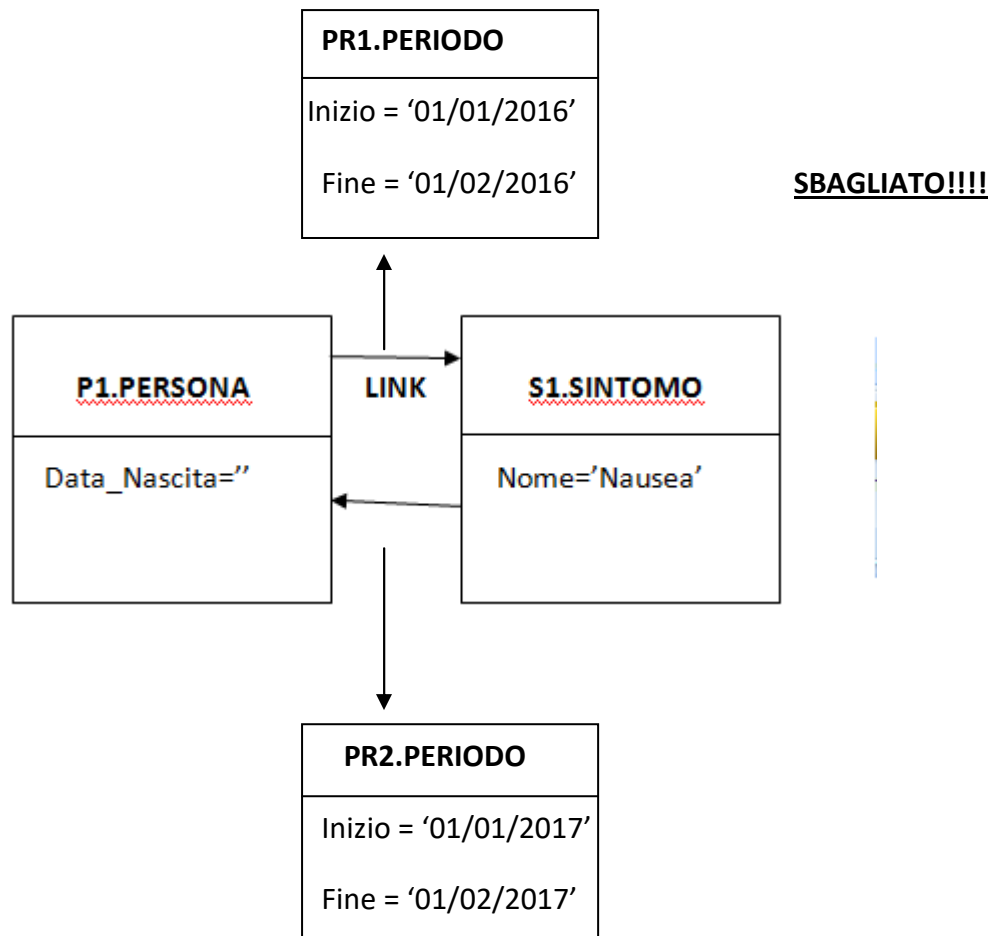
Le istanze della relazione Soffre sono coppie di Object ID.



L'istanza di un'associazione si chiama **LINK**. Non posso avere due link per la stessa associazione.

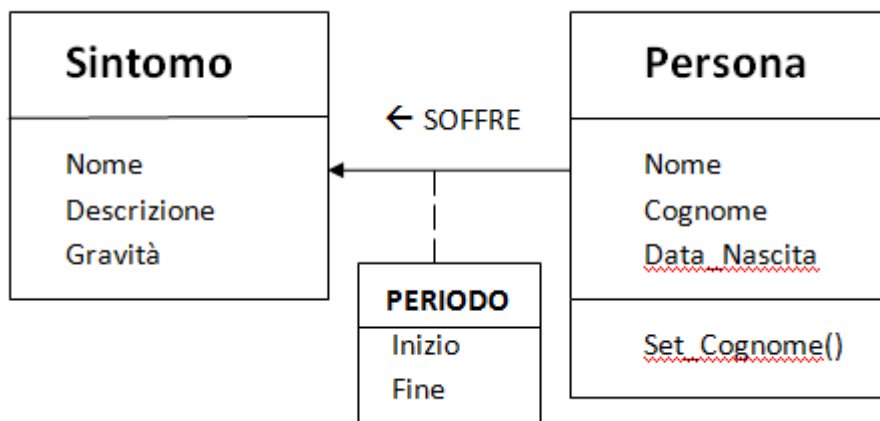
Per dare il nome all'associazione o do un **VERBO** (e allora devo specificare il verso/**navigabilità**) oppure utilizzo un **SOSTANTIVO** (es. presenza, sofferenza,...) dove non devo specificare il verso. Con la freccia non dico da dove parto a dove arrivo. Un'associazione può avere anche **ATTRIBUTI**.

Supponiamo che una persona abbia sofferto di nausea due volte: una volta quest'anno e una volta l'anno scorso. Come faccio a rappresentarlo nel diagramma????



Lo posso gestire in tre modi:

- Attacco al link una classe PERIODO contenente traccia di tutti i periodi (qui ogni coppia SOFF, PERS è UNICA)
- Inserisco una tabella SOFFERENZA (Inizio, Fine) in mezzo tra le tabelle PERSONA E SINTOMO: qui però rischio di avere la stessa coppia (SOFF1,... legata con tanti sintomi e tante persone. Non c'è unicità!!). Lo stesso oggetto può essere legato a tanti oggetti. Potrei mettere come vincolo esterno l'unicità.
- Faccio una relazione ternaria che chiamo PERIODO (Inizio, Fine). Come sono fatti gli elementi delle istanze di questa ternaria? Sono triple: (Pers1, Periodo1, Sint1). Posso legare lo stesso paziente allo stesso sintomo in periodi diversi.



- Collego alla PERSONA P1 una tabella PERIODO PER2 con un'altra tabella SINTOMO S2 (meglio evitare oggetti con lo stesso stato ma non è vietato farlo). Questa soluzione funziona solo in un contesto orientato agli oggetti.

### CARDINALITA'

Posso specificare quanti sono gli oggetti che possono essere messi in relazione con un oggetto su cui mi concentro. La notazione è al contrario di quella che usavamo in BDD.

PERSONA ----- (1,N) SINTOMO

- Un oggetto di tipo PERSONA può essere collegato con tanti SINTOMI (1,N)
- Un oggetto di tipo SINTOMO può essere collegato solo con una sola PERSONA (1,1) o (0,1)

*PRESENZA*

ANAGRAFICA (1,1) ----- (1,1) PERSONA ----- SINTOMO

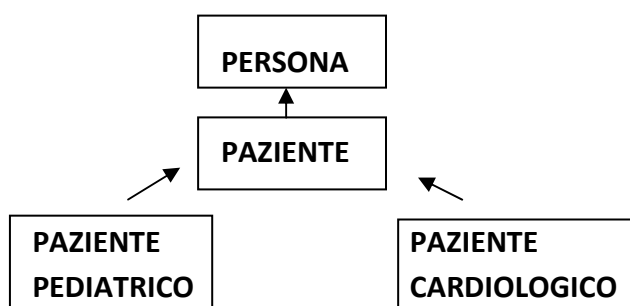
Indirizzo

Professione

### GERARCHIE DI GENERALIZZAZIONE

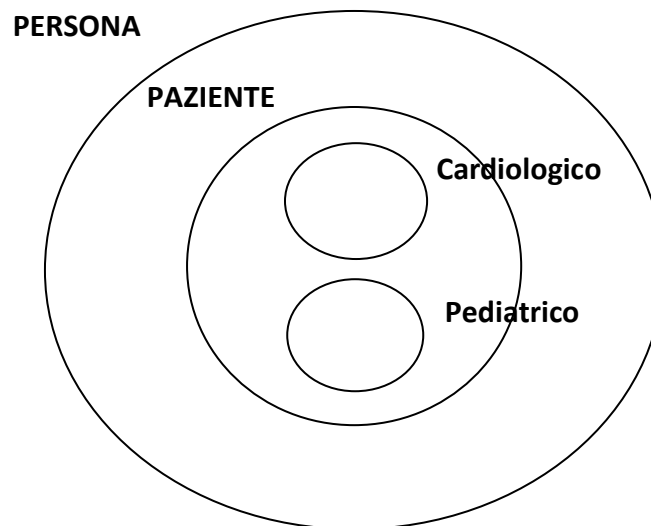
Servono a raccontare due concetti diversi:

- 1) **INHERITANCE:** ha a che fare con i tipi nel senso di descrizione della struttura



Dico una sola volta le caratteristiche comuni, invece di raccontarle più volte (ereditarietà).

- 2) **SUB-TYPING:** ha a che fare con la classe nel senso di usare istanze di un tipo al posto di istanze di un altro tipo. Quando ho un oggetto di tipo Persona da usare, posso usare altri oggetti di altro tipo? Questo ha a che fare con le occorrenze.

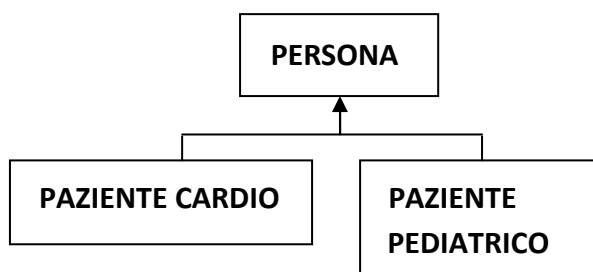


## GERARCHIE DI GENERALIZZAZIONE

### OGGETTO

**TIPO:** descrive la struttura dell'oggetto e rappresenta la signature dei metodi e dice com'è la struttura interna dell'oggetto e attributi rispetto ad altri oggetti. Legato al concetto di tipo c'è il concetto di **class diagram**, in cui rappresento tipi e connessioni tra tipi (posso rappresentare la cardinalità, un verso di lettura,...).

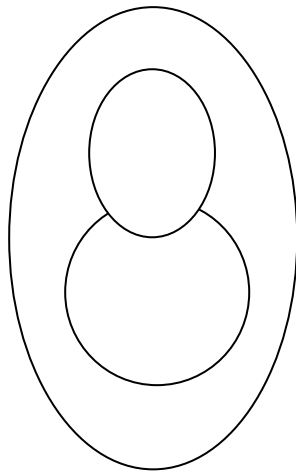
**CLASSE:** insieme di oggetti di un certo tipo e rappresentazione di dati rispetto ad un altro tipo



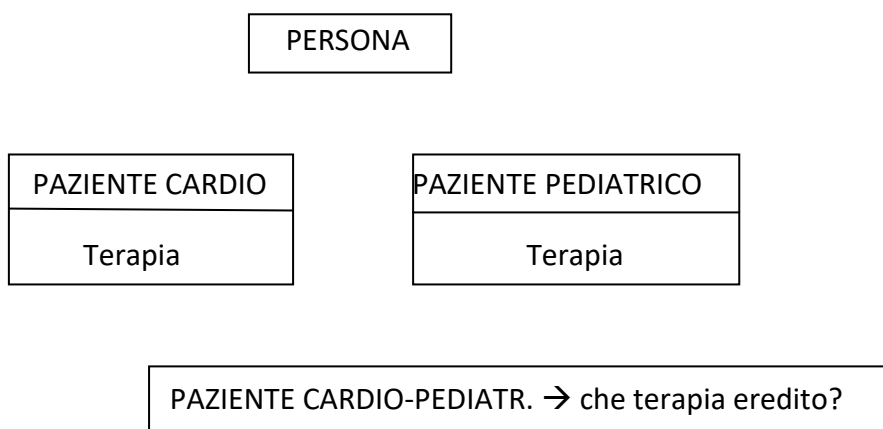
Per quanto riguarda le gerarchie di generalizzazione si deve fare distinzione tra due concetti:

- **Riuso del codice:** le classi generalizzate ereditano tutti gli attributi della classe padre
- **Sub-typing:** due classi sono in relazione di sub-typing, ogni volta che ho un oggetto di una classe non mi fa alcun problema avere un oggetto di un'altra classe (Es. dentro la classe persona non ho problemi ad avere un oggetto di tipo paziente cardio). La relazione di sub-typing ha un verso di relazione.

Ci possono essere anche situazioni, supportate da alcuni sistemi, in cui un oggetto è istanza di più di una classe più specializzata. Ad esempio, potrei avere dei sistemi in cui un singolo oggetto può essere istanza sia di paziente cardio sia di paziente pediatrico (avrà tutte le caratteristiche delle due classi). Non esiste nessuna classe o tipo che rappresenta questa intersezione.



Posso avere inoltre una gerarchia di generalizzazione multipla (ad es. paziente pediatrico-cardiologico).

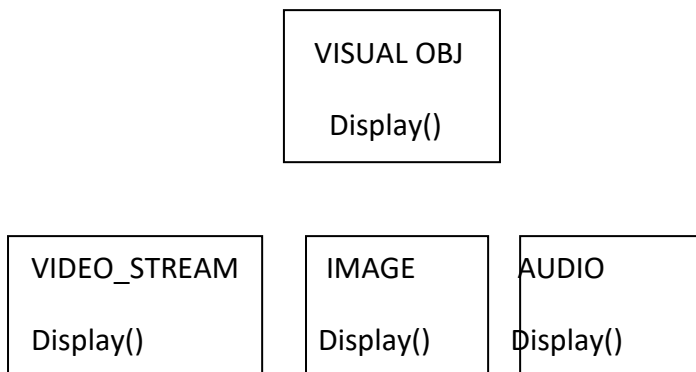


I problemi che possono sorgere in questo caso sono legati al fatto che nessuno ha detto che gli attributi ereditati dalle due classi siano, per esempio, nomi identici appartenenti a insiemi disgiunti. Posso utilizzare dei default o opzioni a livello di specifica che mi permettono quale delle due strade scegliere o usare (se sono indeciso, prendo la terapia della classe elencata prima).

Oppure sfrutto una caratteristica per cui ridefinisco, a livello della classe più specifica, cosa intendo per terapia. Posso ridefinire attributi e metodi localmente.

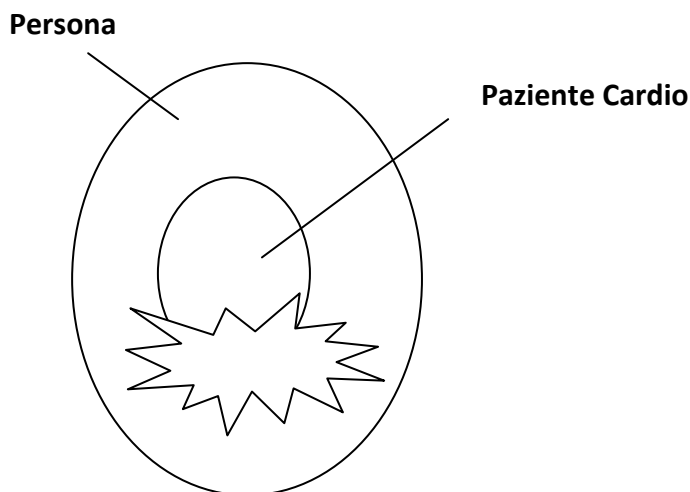
Supponendo di avere dentro Persona il metodo “dati anagrafici”: dentro paziente cardio e paziente pediatrico posso ridefinire lo stesso metodo e, nella definizione, quando poi eseguirò il codice, il sistema software eseguirà sempre il metodo più specifico nella gerarchia (**uso sempre il metodo più specifico**) →

**ESEMPIO DI OVERRIDING** = sovrapposizione di metodi in una gerarchia di generalizzazione.



**OVERLOADING (di metodi)** = un metodo con lo stesso nome ma diverse signature (diverso numero di parametri, diverso tipo di parametri,...). Allo stesso nome di metodo corrispondono tanti pezzi di codice diverso e il sistema è in grado di chiamare l'opportuno codice corrispondente.

**LATE BINDING** = caratteristica per cui il collegamento tra il metodo da invocare e il codice del metodo lo faccio solo a tempo di esecuzione (non riesco a farlo a tempo di compilazione).



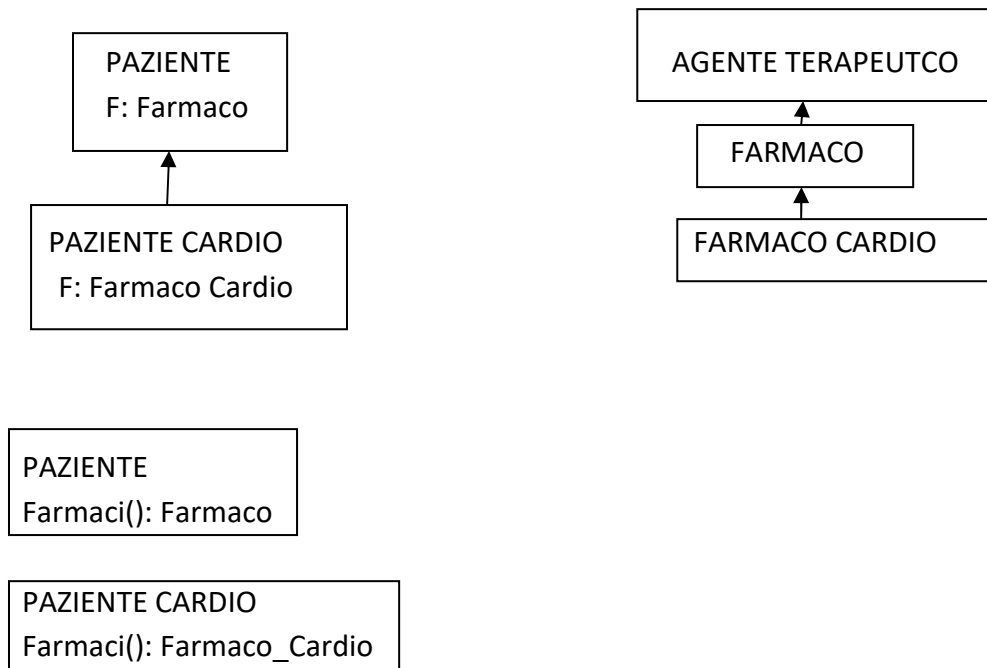
**POLIMORFISMO** = si hanno dei metodi o del codice in generale polimorfico. Allo stesso nome di metodo corrispondono implementazioni diverse.

Una delle cose che posso fare nella specializzazione di una classe è anche di ridefinire sia i metodi sia gli attributi di una classe (cambiando i tipi dei parametri in ingresso del metodo oppure cambio il tipo del parametro in uscita oppure cambio il tipo di un attributo mantenendo lo stesso nome) : Di solito permetto di ridefinire metodi di una classe o attributi grazie a due metodi:

- **Covarianza:** quando sto specificando un metodo, permetto che anche il tipo degli attributi o il tipo dei parametri in ingresso/uscita sia più specifico.
- **Controvarianza:** è il contrario. Quando sto specificando un metodo nella classe più specifica, permetto che il tipo dell'attributo o il metodo che sto specializzando appartenga alla super classe della classe che sto specializzando (permetto di salire nella gerarchia di specializzazione corrispondente). Questa soluzione funziona solo in alcuni casi.



### Esempio covarianza:



Rispetto alla covarianza, funziona perfettamente il secondo caso. Ogni volta che ho un invocazione del metodo Farmaci() mi può arrivare sia un oggetto di tipo Farmaco, sia un oggetto di tipo Farmaco\_Cardio. Questa cosa non funziona nel primo caso: può dare errori se specifico gli attributi o i parametri in ingresso. Qui funziona meglio la covarianza.

PAZIENTE  
F: Farmaco

### Covarianza

PAZIENTE CARDIO  
F: Agente Terapeutico

### Covarianza delle proprietà (= degli attributi)

CLASS Medico

TYPE tuple (dati\_anagrafici: tuple (cognome\_nome: stringa))

METHOD set\_dati\_anagrafici

END;

METHOD Body SET dati\_anagrafici IN CLASS Medico {

SELF → dati\_anagrafici = tuple(cognome\_nome: "aaa");

}

CLASS Cardiologo INHERITS Medico

TYPE tuple (dati\_anagrafici: tuple (cognome\_nome:stringa, indirizzo:stringa))

// questa tupla è in covarianza con la tupla dati\_anagrafici della classe Medico

METHOD read\_indirizzo: string

END;

```

METHOD Body read_indirizzo IN CLASS Cardiologo {
    RETURN (SELF → dati_anagrafici).indirizzo
}

```

### **Esecuzione:**

```

M = new Cardiologo // oggetto di tipo Cardiologo
M → set_dati_anagrafici // metodo set_dati_anagrafici : istanzia la tupla con un solo
attributo
M → read_indirizzo

```

La covarianza mi da problemi perché non so come verranno usati questi oggetti in run time.

### **Covarianza dei parametri di ingresso del metodo**

```

CLASS Paziente
    TYPE tuple (cognome:string, terapia:farmaco)
    METHOD stessa_terapia(p:paziente):boolean
    // lo uso per confrontare pazienti diversi
END;

```

```

CLASS Paziente_Cardio INHERITS Paziente
    METHOD stessa_terapia(pc:paziente_cardio):boolean
    // gli passo un paziente cardio, non un paziente "normale"
END

```

### **Esecuzione malvagia:**

```

METHOD pazienti_simili (p:paziente) IN CLASS Paziente

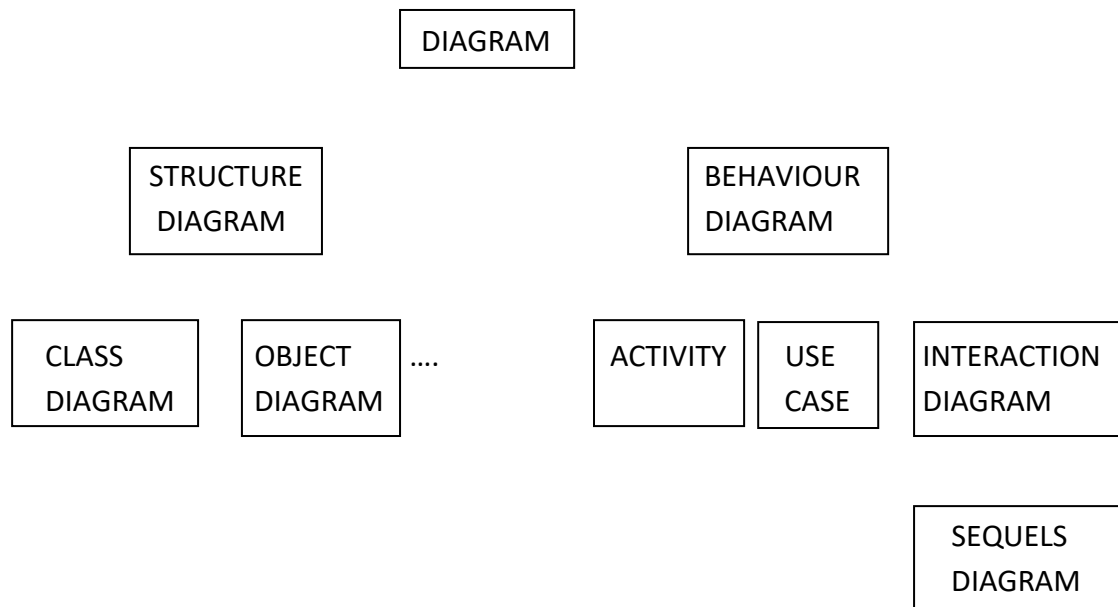
METHOD body pazienti_simili (p:paziente) IN CLASS Paziente {
    RETURN ( P → stessa_terapia (SELF) AND ... qualunque cosa) }

```

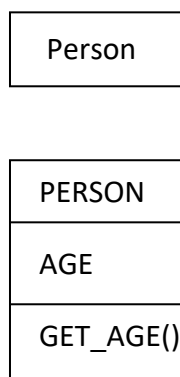
```

P1: paziente
PC1: paziente_cardio
P1 → pazienti_simili (PC1)
// su PC1 chiamo stessa_terapia e gli passo P1
PC1.stessa_terapia(P1)

```



## CLASSE



L'unica cosa obbligatoria quando specifico una classe è il nome, di cui è obbligatorio scrivere in *upper camel case* (es: PazienteCardiologicoOspedaliero). Eventualmente si potrebbe avere nella casella della classe tra parentesi angolari doppie uno *stereotipo* (lo uso come caratteristica del mio modello). Ad esempio: <<temporal>>. Questi stereotipi possono essere rappresentati anche con dei simboli: ad esempio, il simbolo di un piccolo orologio.

Posso avere al più lo slot della classe diviso in tre slot:

- Nome della classe
- Attributi (non obbligatori)
- Metodi (non obbligatori)

Gli attributi o i metodi devono essere rappresentati in *lower case camel* (ad es.: getAge() ).

Gli attributi possono essere caratterizzati da *ornamenti*:

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

Per le operazioni (metodi):

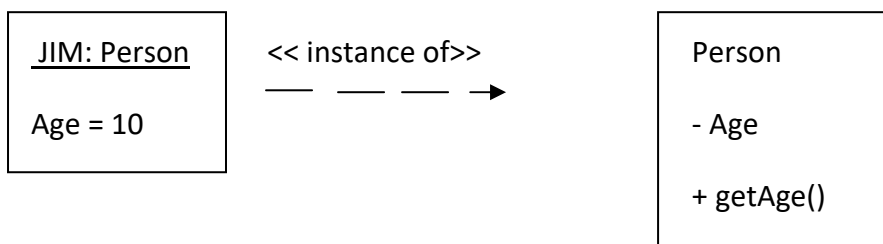
- <visibilità> <nome Metodo> ({<nome Parametro: tipo parametro>{nomePar2: tipoPar2,...>})  
<tipoParametroRestituito>

Solo il nome degli attributi e dei metodi sono obbligatori.

L'idea è che, quando parto a livello di progettazione più astratta subito dopo l'analisi dei requisiti, inizio a specificare dei diagrammi dei requisiti molto sintetici.

- + → public : classe pubblica
- - → private
- # → protected
- ~ → package



### Come visualizzare le istanze di una classe dentro il diagramma degli oggetti



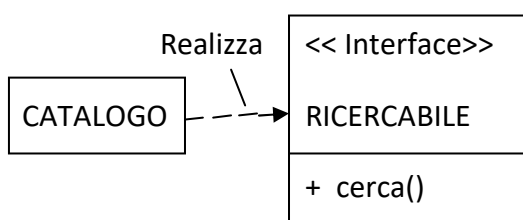
// generico oggetto di tipo persona

// degli oggetti non rappresento i metodi (non è obbligatorio mettere attributi ma se lo faccio devo metterne i valori)

### RELAZIONI FRA CLASSI

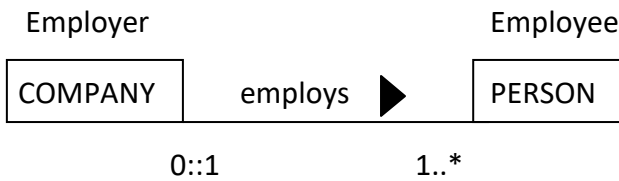
- Associazioni (semplice linea)
- Generalizzazioni (freccia vuota)
- Realizzazione (freccia 'piena' con linea tratteggiata): relazione tra le interfacce e una classe
- Dipendenza (freccia 'normale' con linea tratteggiata →)
- Aggregazione 
- Composizione 

### Realizzazione



Le interfacce contengono solo metodi, da implementare.

**Associazione** = collegamento tra due o più classi (posso coinvolgere anche due volte la stessa classe)



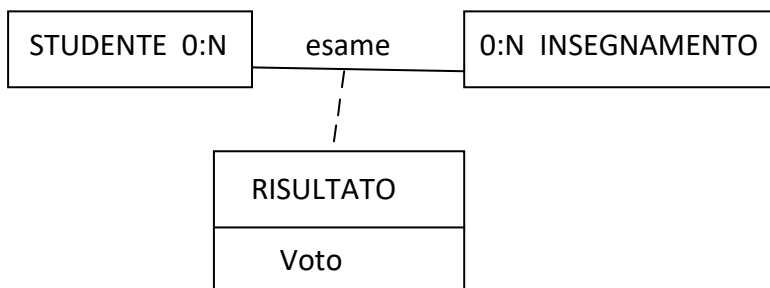
### NAVIGABILITA'



Se vogliamo dire che la direzione inversa non funziona, metto una croce. La mancanza sia della freccia sia della croce indica “navigabilità non specificata”. Specificare sempre la navigabilità appesantisce il programma.

Di solito un approccio seguito è questo:

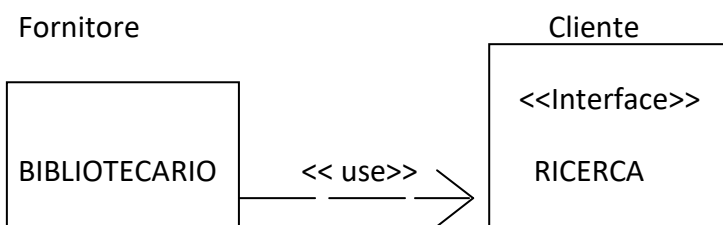
- Le croci non si usano
- Se non ci sono frecce la navigabilità è in entrambe le direzioni
- Se specifico la navigabilità, pur senza la croce, la navigabilità è solo quella del verso della freccia



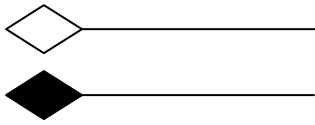
$\{(S1,I1), (S2,I1), \dots\} \rightarrow$  NO COPPIE UGUALI

Se dico che la navigabilità è in entrambe le direzioni, a livello di implementazione potrei pensare di avere il caso in cui ogni volta che faccio un’aggiunta di qualche coppia di questi oggetti devo andare a modificare tutti e due gli oggetti delle classi.

### Dipendenza

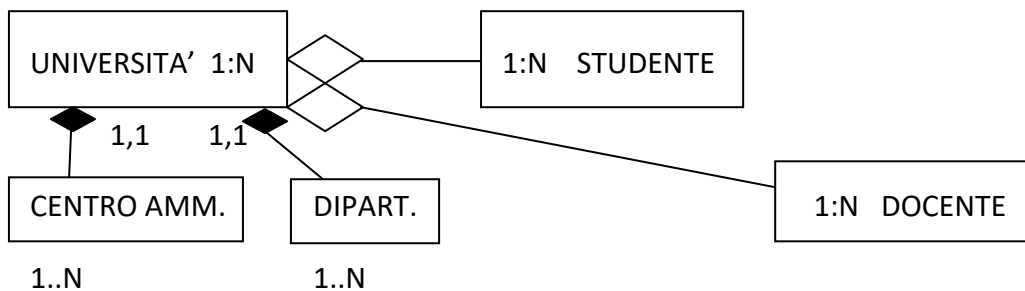


## Aggregazione e composizione



Relazione di contenimento parte-tutto:

- Aggregazione : tutte e due le classi coinvolte hanno spessa dignità
- Composizione: le parti dipendono da tutto e non possono esistere senza il tutto



Un docente o studente potrebbe non essere connesso con l'università.

## USE CASE DIAGRAM

Sono usati a livello di analisi e rappresentazione dei requisiti o alla fine di tutto per rappresentare i casi ad alto livello del software (nasconde ogni tipo di applicazione → no sequenza di attività nel tempo)

**ATTORI:** rappresentati da degli omini e ci possono essere anche gerarchie di generalizzazione negli attori. Non sono necessariamente attori umani: potrebbero esserci attori esterni che sono altri sistemi software. Gli attori sono coloro che fanno partire i casi d'uso o che vengono coinvolti nell'esecuzione dei casi d'uso. Non identificano singoli specifici utenti, oggetti,... ma fanno riferimento ad un ruolo, a una funzione che possono appartenere a più persone fisiche o software. Per identificarli devo capire chi interagisce col sistema, quali sono i ruoli, chi interagisce con i ruoli,...

DOCENTE



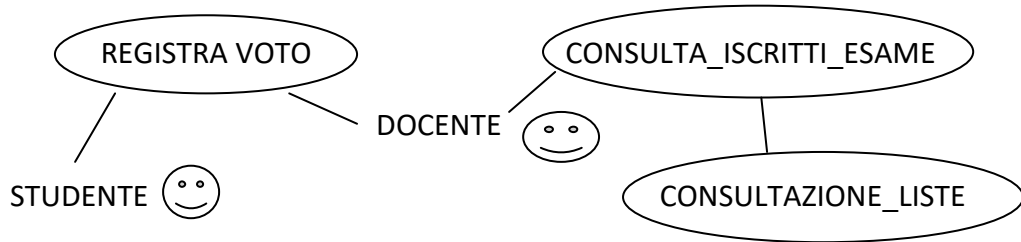
RESPONSABILE ☺

STUDENTE ☺

PERSONALE AMMINISTRATIVO ☺

CORSO DI LAUREA

**CASI D'USO:** si rappresentano all'interno di un ovale con un'etichetta



**NON SI POSSONO COLLEGARE ATTORI AD ATTORI!!!!**

### **DIPENDENZE TRA I CASI D'USO**

**<<Extends>>** : comportamento opzionale. Uno use case è una parte eccezionale/opzionale all'interno di un altro use case.

**<<Include>>** : stereotipo. Etichette con significato predefinito all'interno di un diagramma UML ("prevede sempre"). La freccia va verso l'incluso.

**Extension Point:** ad esempio, lo studente non è nella lista degli iscritti all'esame

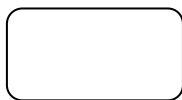
Esiste un altro pezzo di documentazione alfanumerica detta **specifica degli use case** che racconta quelle parti temporali che non si riesce a rappresentare graficamente negli use case.

### **ACTIVITY DIAGRAM**

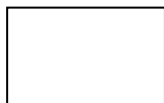
Sia a supporto degli use case sia a supporto di altre attività di funzione, a livello di comportamento, ci sono i **diagrammi di attività** in cui si vanno a rappresentare quegli oggetti di nostro interesse (attività legate al tempo). Può essere legato a diversi livelli di dettaglio di rappresentazione. Racconto il 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 che sto implementando,...

**Componenti:**

- 1) **Box con angoli arrotondati:** rappresenta una singola attività o azione. Rappresenta l'unità di comportamento al più fine livello di astrazione.

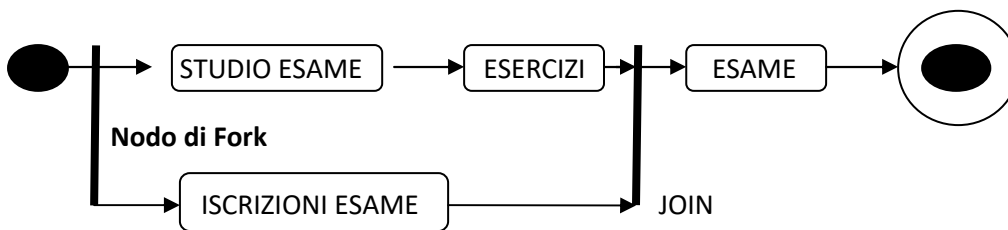


- 2) **Box con spigoli vivi:** rappresentano gli oggetti software usati come input/output delle attività.



### 3) Nodi di controllo:

- diamante (rappresenta una scelta)
- barra (rappresenta l'esecuzione contemporanea di più attività)
- pallino pieno (rappresenta l'inizio di un diagramma, di un'esecuzione di un insieme di attività)
- pallino pieno circondato da un'altra circonferenza (rappresenta la fine di tutta l'esecuzione)
- pallino con una croce dentro (rappresenta l'inizio di una traccia di esecuzione che finisce)

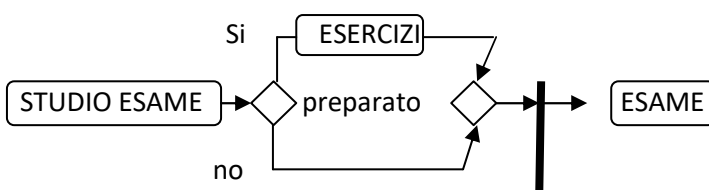


Ogni attività non può cominciare prima che finisca l'azione precedente.

Per interpretare questi diagrammi si usa un'interpretazione basata sulle reti di Petri: si usa un token (segnaposto). Ogni volta che genero un inizio di attività creo un token che prosegue lungo il senso della freccia e termina quando termino l'esecuzione del diagramma di attività.

**FORK:** divide in due strade separate il percorso (ad es., iscrizione esame e studio esame non sono correlate temporalmente tra loro).

**JOIN:** prima di lasciar continuare un token sulla sua uscita, aspetta che arrivino tutti i token dalle sue entrate. Continua solo quando tutte le sue linee di ingresso sono terminate. Consuma tutti i token e ne fa partire uno solo.



**NODI DI DECISIONE E FUSIONE:** ricevono i token in ingresso, e, in accordo con una particolare condizione, fanno seguire al token una strada piuttosto che un'altra. Il nodo fusione sta ad aspettare in ingresso un token che gli permetta di andare avanti.