Ingegneria del software semplice (per davvero)

Rovesti Gabriel

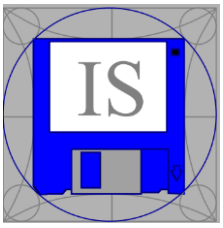
Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione: metodo e concetti chiave di sviluppo del software (Vardanega)



L'ingegneria del software è un campo di studi che si concentra sulla progettazione, lo sviluppo, la manutenzione, il collaudo e la valutazione dei sistemi software. Comporta l'applicazione di principi e pratiche scientifiche alla creazione e al funzionamento dei sistemi software.

Gli ingegneri del software lavorano per sviluppare e mantenere il software che soddisfa le esigenze degli utenti e aiuta le organizzazioni a raggiungere i loro obiettivi. In particolare, anche da parte nostra, apprendiamo metodi e pratiche di lavoro alla base della professione informatica (integrando la teoria con la pratica):

- Gestire il tempo
  - Disponibilità, scadenze, conflitti, priorità
- Collaborare
  - Fissare obiettivi, dividersi compiti, verificare progressi, riportare difficoltà
- Assumersi responsabilità
  - Fare quanto pattuito, agire al meglio delle proprie capacità, auto-valutarsi prima di valutare
- Auto-apprendere

Vogliamo infatti avvicinarci ad un modo di lavorare (way of working) professionale, cioè: «operante allo stato dell'arte», in termini di conoscenze tecnologiche e metodologiche. La tecnologia avanza continuamente ed è necessario continuare *con l'auto-apprendimento* cercando di colmare i buchi ed apprendere nuove nozioni. La conoscenza passa dalla comprensione profonda, sperimentata, dei significati (non ricordare a memoria, ma riconoscere determinate applicazioni di nozioni).

Vogliamo fissare tali conoscenze in un glossario, al fine di legare la teoria con la pratica e raffinare la comprensione di quanto fatto:

- Raccolta di termini/concetti centrali al dominio SWE
- Registrati in modo da facilitarne la localizzazione
- Corredati dalla nostra personale specifica del loro significato e ogni altra informazione utile a riconoscerli

- Glossario delle parole chiave (definizioni *intuitive* in questa lezione):

  - Protocollo: in informatica, un insieme di regole o procedure per la trasmissione di dati tra dispositivi elettronici, come i computer. Affinché i computer possano scambiarsi informazioni, deve esistere un accordo preesistente su come le informazioni saranno strutturate e su come ciascuna parte le invierà e le riceverà.

  - Progetto (software) → Visto (esaminato) da chi lo crea e da chi lo usa. Idee chiave:
    - Sviluppo continuativo
    - Professionale
    - Valutazione qualitativa
    - Collaborativo
    - Monitoraggio

    Esso deve essere un mezzo e non un "obiettivo" (goal/aim).
    Quest'ultimo, infatti, va compreso; infatti, è visibile a pochi da subito ma l'obiettivo deve essere chiaro a tutti. Non si intende programmazione, quanto piuttosto la *realizzazione di obiettivi* per mezzo dello stesso progetto.

- Sviluppo: Percorso di realizzazione tra obiettivi *detti* ed obiettivi *raggiunti* per il tramite del software. Questi ultimi rispondono principalmente a dei *bisogni (needs)*, essendo essi stessi il *mezzo* per realizzarli. Si intende tutto ciò attraverso il percorso di progettazione (*design*), di cui il progetto è un prodotto. L'insieme delle regole che lo realizzano è il *paradigma*, seguito dagli informatici che realizzano software e applicano queste regole.
  - Per realizzare in maniera coesa (interconnessa, espressa come unica entità) le azioni di tanti programmi distinti.
  - Il ruolo dell'informatico è distinguere i bisogni tecnici e i bisogni utente creando un prodotto di *qualità* (quindi seguendo una chiara idea di realizzazione (protocollo), comodo da usare, accessibile, ecc.), risultando dunque professionale. Si cerca di raggiungere lo stato dell'arte (inteso come la qualità in un senso personale) attraverso un costante *autoapprendimento*. Colui che ha gli strumenti e crea è il *practitioner*, continuando ad imparare in autonomia.

Introduzione (slide omonima): obiettivi, metodo, concetti chiave

(Eventuali approfondimenti listati dal prof:
http://worrydream.com/refs/Brooks-NoSilverBullet.pdf
https://www.acm.org/binaries/content/assets/membership/images2/fac-stu-poster-code.pdf)

Questo nel nostro corso si realizza tramite un progetto didattico collaborativo:
- Promosso da un proponente esterno (aziende terze che portano un'idea)
- Con esigenze e obiettivi funzionali innovativi
- Complesso, impegnativo, visionario
- Tecnologicamente avanzato

Segue una definizione di Harold Kerzner, autore di "Project Management") del termine:

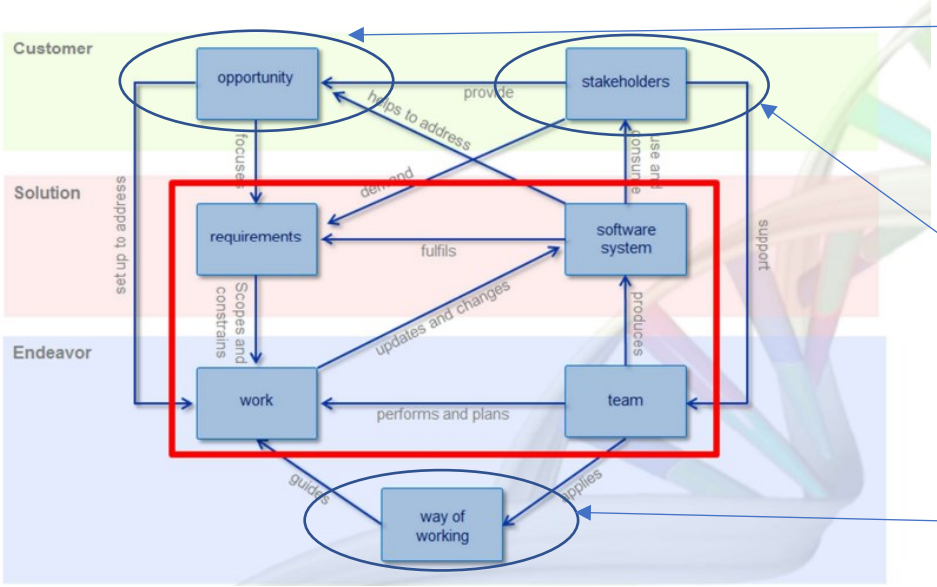
Progetto
- Insieme di attività che:
  - Devono raggiungere determinati obiettivi a partire da specifiche fissate (rispondendo ai needs-bisogni)
  - Hanno una data d'inizio e una data di fine fissate
  - Dispongono di risorse limitate (persone, tempo, denaro, strumenti)
  - Consumano tali risorse nel loro svolgersi
- L'uscita di un progetto è un prodotto composito
  - SW (software) sorgente/eseguibile, librerie, documenti, manuali

I costituenti di un progetto sono:
- Pianificazione → Studiare la natura del problema, capire come impostare le attività sulla base della domanda e pianificare l'organizzazione delle risorse
  - Gestire risorse (persone, tempo, denaro, strumenti) in modo responsabile, in funzione degli obiettivi

- Analisi dei requisiti → I needs sono tradotti in prodotti informatici (non sapendo come fare, si chiede e dialoga con una serie di interlocutori)
  - Definire cosa bisogna fare, normalmente compreso tramite interazione continuativa tra le parti in gioco (quelli che in project management sono chiamati *stakeholders*, quindi "portatori di interesse")

- Progettazione (design) → Trasformare l'idea di comprensione (profonda) del problema in modo attuabile e con una soluzione sensata ed accettabile. Qui non si ha una sola soluzione, ma ce ne possono essere tante, definendo come farlo

- Realizzazione (implementation) → Creare un *utensìle* (cioè, avere un prodotto *usabile*; quindi, una volta creato compie correttamente la sua funzione e sia utile allo scopo voluto)
    - Farlo, perseguendo qualità (cioè, il grado di bontà oggettiva delle azioni eseguite)
    - Accertando l'assenza di errori od omissioni
    - Accertando che i risultati soddisfino le attese



Vedere la possibilità di creare
un prodotto utile allo scopo possibile

Chi ha cosa da dire significative sulle opportunità, sui bisogni e sulla soluzione/implementazione della stessa.
Possono essere clienti, committenti, venture capitalists (investitori)

Stato dell'arte, adattato al proprio livello sulla base delle regole presenti

Figura 1 - Proveniente da https://semat.org/

Collassare design ed implementazione, esempio scrivendo codice, è tipico di chi progetta male.
Prima si pensa bene a cosa fare e l'implementazione risulta essere conseguenza (non premessa) del design.
In tal modo, mi assicuro di soddisfare le aspettative di chi lo richiede, chiedendo ed interagendo spesso.

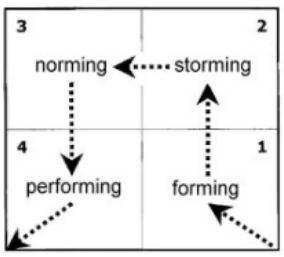
Cosa <u>non è</u> un progetto:

- Si è accecati dalla fondamentale inutilità di fondo dei loro prodotti, dal senso di successo che si prova nel farli funzionare (il fatto che compili/funzioni ci toglie dall'esaminare ulteriormente un prodotto)
- In altre parole, i loro difetti fondamentali di progettazione sono completamente nascosti dai loro difetti di design o difetti superficiali (errori di sintassi, segmentation fault, ecc.)

Esso dovrebbe essere la manifestazione concreta sulla base di regole di un prodotto preciso e *professionale*: il bisogno esiste in quanto individuato all'inizio, ma deve rispondere precisamente ad alcuni requisiti prestabiliti.

Altri termini del glossario:

- *Teamwork*
    - Lavoro collaborativo che punta a raggiungere un obiettivo comune in modo efficace ed efficiente
    - I membri del team sono inter-dipendenti
    - La gestione di questa inter-dipendenza richiede il rispetto di regole e di buone pratiche:
        - Condivisione e collaborazione
        - Comunicazioni aperte e trasparenti: risoluzione dei conflitti
        - Assunzione di responsabilità: coordinamento
        - Condivisione dei rischi
    - La sua base è un solido *way of working*



- *Stakeholder* (portatore di interesse)
    - Tutti coloro che a vario titolo hanno influenza sul prodotto e sul progetto
        - La comunità degli utenti (che usa il prodotto)
        - Il committente (che compra il prodotto)
        - Il fornitore (che sostiene i costi di realizzazione)
        - Eventuali regolatori (che verificano la qualità del lavoro)

- *Way of working*
    - Come organizzare al meglio le attività di progetto (quindi, in modo professionale)

Per svolgere un progetto potendo confidare nel suo successo serve *ingegneria*, quindi usando principi scientifici e matematici noti ed autorevoli. Nel caso della software engineering (SWE), si tratta di usare le cosiddette *best practices*, per garantire i migliori risultati in circostanze note e specifiche (*practical ends*).

Diamo a glossario alcune caratteristiche utili sulla SWE:
- Disciplina per la realizzazione di *prodotti SW* impegnativi da richiedere attività collaborative
- Capacità di produrre "in grande" e "in piccolo"
- Garantendo qualità: *efficacia*
    - Misura della capacità di raggiungere l'obiettivo prefissato
- Contenendo il consumo di risorse: *efficienza*
    - Misura dell'abilità di raggiungere l'obiettivo impiegando le risorse minime indispensabili
- Lungo l'intero periodo di sviluppo e di uso del prodotto: *ciclo di vita*
    - Gli stati che il prodotto SW richiesto assume dal suo concepimento (bisogni → needs) all'uso e poi eventualmente al ritiro
- Lavorare allo stato dell'arte seguendo le *best practices*
    - Modo di fare (*way of working*) noto, che abbia mostrato di garantire i migliori risultati in circostanze note e specifiche

Un sistema SW è tanto più utile quanto più è usato.
- *Metrica*: integrale della sua intensità d'uso nel tempo

Più lunga la vita d'uso di un prodotto, maggiore il suo costo di *manutenzione*
- Manutenzione: insieme di attività necessarie a garantire l'uso continuativo del prodotto
    - Reattivamente (per correzione dopo malfunzionamento) o preventivamente

Il costo di manutenzione ha varie componenti:
- Mancato guadagno, perdita di reputazione, reclutamento esperti, sottrazione risorse ad altre attività

I principi SWE puntano ad abbassare tali costi
- Sviluppando SW più facilmente manutenibile

L'obiettivo dell'ingegneria del software è raccogliere, organizzare, consolidare la conoscenza (*body of knowledge*) necessaria a realizzare progetti SW con efficacia ed efficienza (collezione e manutenzione migliorativa di best practice) e quindi applicare principi ingegneristici calati nella produzione del SW.

Secondo il glossario IEEE l'approccio ha varie caratteristiche:
- *Sistematico*
    - Modo di lavorare metodico e rigoroso
    - Che conosce, usa ed evolve le best practices di dominio
- *Disciplinato*
    - Che segue le regole che si è dato
- *Quantificabile*
    - Che permette di misurare l'efficienza e l'efficacia del suo agire

Distinguiamo infatti la figura del programmatore da quella del software engineer:

- il programmatore scrive programmi da solo, in modo tecnico e personalizzato (creativo);
- il software engineer realizza parte di un sistema complesso con la consapevolezza che potrà essere usato, completato e modificato da altri. Egli comprende il contesto in cui si colloca il sistema cui contribuisce (che non si limita al SW) e cerca di attuare dei compromessi intelligenti e lungimiranti tra costi – qualità, risorse – disponibilità, esperienza utente – facilità di realizzazione.

Il nostro scopo è quindi applicare tutti questi principi sulle attività di progetto.

Il metodo di acquisizione delle competenze avanza in corrispettivo alle sfide proposte, ciascuna con un punto di partenza e di arrivo, creandosi la condizione giusta per apprendere ed entrare nel flow.
L'obiettivo è creare un *MVP (Minimum Viable Product)*, cioè un prodotto con abbastanza caratteristiche da attrarre clienti e validare un'idea di prodotto nelle prime fasi di sviluppo, con il minimo spreco di tempo e di denaro ed entrando subito nel mercato (prodotto reale, funzionalità minime ma sufficienti, costo contenuto, tempo di realizzazione breve).

L'impegno necessario per raggiungere gli obiettivi di progetto ha *limite superiore stretto*:

- Per essere compatibile con gli altri propri obblighi personali
- Ma richiede impegno «solido», che sconsiglia la partecipazione con "arretrati"

Gli obiettivi di progetto vanno fissati in modo elastico:

- Per essere fattibili entro i limiti di impegno dati
- Tra MVP e un massimo ambizioso, concordati dinamicamente con i due interlocutori del progetto
    - Docente-committente (Tullio)
    - Proponente-cliente-mentore (Azienda)

Lo studio di SWE si basa costruendo incrementalmente il proprio glossario (mentale, cartaceo, digitale), prima con la teoria, consolidandolo con la pratica e confrontandolo con i colleghi (unendo conoscenze parziali, correggendosi reciprocamente). Il glossario serve a cogliere, fissare, ritrovare concetti chiave della materia, evitando di essere superficiali.

Processi di ciclo di vita (Vardanega)

(Eventuali approfondimenti:
https://www.math.unipd.it/~tullio/IS-1/2010/Approfondimenti/A03.pdf
https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf
http://www-scf.usc.edu/~csci201/lectures/Lecture11/boehm1988.pdf
http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf
https://webstore.iec.ch/preview/info_isoiec15271%7Bed1.0%7Den.pdf
http://www.scrumalliance.org/learn_about_scrum)

Un sistema SW, tanto più è usato, tanto più dovrebbe essere posto a manutenzione, la quale permette di risalire ad uno storico di vita di un SW. L'uso permette di scoprire necessità, difetti, adattandosi alle nuove esigenze e creandosi nuovi obiettivi. In particolare, *garantisce di continuare a soddisfare i requisiti utente*, creando sicurezza (anche in merito a regole e standard) ed affidabilità (migliorando l'esperienza utente ed estendendo la vita del software, risparmiando tempo e risorse).

L'attività di manutenzione ha uno storico, cioè una memoria di ciò che funziona (ora) o ha funzionato. Questa idea è il controllo di versione/versionamento. Esso è il processo di assegnazione di identificatori unici alle diverse versioni di un'applicazione software, chiamati numeri/stringhe di versione, che rappresentano le versioni principali, minori e patch del software e definiscono quanti cambiamenti sono avvenuti nel software, testimoniando la sua evoluzione.
Questo ha tre caratteristiche chiave: reversibilità (posso sempre tornare indietro), concorrenza (più persone apportano modifiche e si facilita l'integrazione) ed annotazione (spiego cosa ho fatto nelle modifiche).

Scritto da Gabriel

Un software non è un oggetto statico (*monolite*); ciò lo renderebbe fragile e complesso da gestire. Occorre pensarlo come costituito di parti, le quali sono disposte secondo un preciso ordine. Questo semplifica la comprensione, lo sviluppo e la manutenzione stessa. Normalmente, in questo caso si parla di *moduli*, quindi parti di prodotto SW che eseguono uno specifico compito/specifiche funzioni.

Questo approccio è definito *modularità*, quindi creazione di componenti indipendenti che possono essere sviluppate e mantenute separatamente (migliora manutenibilità, scalabilità, collaborazione e riutilizzo).
La decisione di riconoscere quali parti usare e come tenerle insieme è detto configurazione.
Da integrare al controllo di versione vi è il controllo di configurazione, al fine di tracciare e gestire cambiamenti al software e ai file ad esso correlati (ogni parte ha una specifica "storia").

- Le parti che cambiano molto sono quelle molto esposte agli utenti
- Le parti che cambiano poco sono dei protocolli software che rimangono fissi

La storia di un prodotto è definita da maturazioni progressive, definite come stati.

- E.g.: concezione (analisi dei requisiti) → sviluppo (design, realizzazione) → utilizzo → ritiro

L'insieme di attività svolte sul prodotto SW per cambiare il suo stato di origine tali da permetterne una transizioni di stato sono gli archi.
Quali/quanti siano gli stati e quali regole attivino o abilitino gli archi (pre- e post-condizioni) dipende da:

- Vincoli/obblighi contrattuali, impegni (way of working), opportunità

Il cambiamento di stato viene causato dal *design*, sia per motivazioni interne che esterne allo stato di sviluppo, continuando a soddisfare i *needs* (realizzandolo progressivamente con l'implementation, mantenendo i principi cardine ed evolvendolo).

Ogni progetto ha il compito di cercare di spingere un prodotto verso una nuova fase del suo ciclo di vita, con il solito obiettivo di creare qualcosa di usabile (implementation sulla base del design), spingendolo tra un segmento e l'altro.
Più nello specifico, il ciclo di vita del SW intende gli stati che il prodotto SW assume tra concepimento e ritiro in conseguenza delle attività svolte su di esso (sulla base di attività ripetute comunemente).

Definiamo in generale:

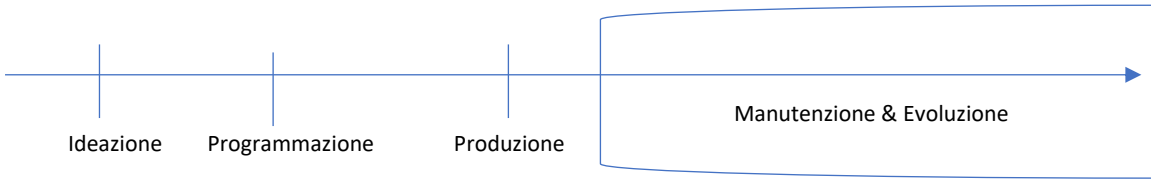
- i processi di ciclo di vita, che raggruppano e codificano stati/transizioni nel ciclo di vita di un SW effettuando quelle giuste (cioè, trasformando ingressi (needs) in uscite (requisito software);
- i modelli del ciclo di vita, capendo quali stati/transizioni privilegiare, quali processi attivare ed attuare aderendo ad un modello di ciclo di vita e pianificare le attività, eseguendole e controllandone lo svolgimento

Le regole dello stato dell'arte hanno stabilito quali attività privilegiare seguendo la logica del ciclo di vita, in cui ogni attività va opportunamente configurata.

Object Oriented Programming Principles – Le dipendenze fra componenti (Cardin)

(Eventuali approfondimenti:
https://www.math.unipd.it/~rcardin/swea/2022/Dependency%20Management%20in%20Object-Oriented%20Programming.pdf
http://blog.rcard.in/design/programming/oop/fp/2018/06/13/the-secret-life-of-objects.html
http://blog.rcard.in/design/programming/oop/fp/2018/07/27/the-secret-life-of-objects-part-2.html)

Il metodo di produzione del software segue dei principi di massima, cosiddetti ingegneristici, affinché la produzione di qualcosa di astratto come un software sia un processo ripetibile. Il ciclo è come segue; nel corso vediamo soprattutto la parte di Manutenzione (può essere molto dannoso chiunque ci metta mano, portando ad una potenziale regressione se fatta male, oppure non è possibile aggiungere nuove funzionalità, portando alla morte immediata del software):



Il processo di scrittura del software è un viaggio lungo e ricco di errori; ovviamente, questi sono la principale forma di apprendimento. La progettazione di un software deve idealmente comprendere sempre direttamente lo stakeholder, tale che si abbia un confronto diretto con chi conosce il dominio applicativo di riferimento. Caso per caso, non si ha una soluzione universale: dipende sempre dal contesto.

Parliamo quindi della programmazione ad oggetti (object oriented programming/OOP), che è un paradigma di programmazione basato sul concetto di "oggetti", unità di codice autonome che rappresentano entità del mondo reale e incapsulano dati e comportamenti.
Si basa su tre principi cardine della programmazione ad oggetti:

- Encapsulation (incapsulamento)
    - L'incapsulamento è il principio del raggruppamento di dati e comportamenti in un'unità autonoma chiamata "oggetto". Esso permette di “nascondere” le informazioni all’interno di moduli, così strutturando il software (information hiding)

- Inheritance
    - Principio della creazione di un nuovo oggetto basato su un oggetto esistente, detto "genitore". Il nuovo oggetto, detto "figlio", eredita gli attributi e il comportamento dell'oggetto padre e può anche avere attributi e comportamenti propri. L'ereditarietà consente agli sviluppatori di creare una gerarchia di oggetti e di riutilizzare il codice, risparmiando tempo e migliorando l'organizzazione del software.

- Polymorphism (polimorfismo)
    - Principio che consente agli oggetti di assumere più forme o "comportamenti". Ciò può essere ottenuto attraverso l'ereditarietà o l'uso di interfacce, che sono contratti che specificano il comportamento che un oggetto deve implementare. Il polimorfismo consente di utilizzare gli oggetti in modo flessibile e generico, migliorando l'estensibilità e il riutilizzo del software.

Essi non sono perfetti; nessuno dei tre principi può esistere implicitamente senza gli altri. Infatti, ognuno di questa si basa sulla separazione ideologica di dati che dipendono gli uni dagli altri per arrivare effettivamente alla definizione di oggetto (l’incapsulamento dipende dall’essere in una gerarchia, il polimorfismo richiede di poter incapsulare dati e far parte di una gerarchia, l’ereditarietà richiede polimorfismo per i vari sottotipi, etc.).

Scritto da Gabriel

Si distingue in questo il comportamento *estrinseco/extrinsic:* più le variabili, con questo paradigma, hanno scope ampio, più problemi abbiamo; se cambia valore (side effects), il programma alla lunga non è più adatto per determinati compiti; tanto più se si ha a che fare con delle gerarchie, dove una modifica si ripercuote su tutti gli altri oggetti facenti parte.

Nella *programmazione procedurale (procedural programming)*, si ha la funzione/*procedure* come blocco di costruzione, in cui le variabili cambiano e anche i loro valori (side effects). I dati sono primitivi e strutturati in record, senza connessione tra dati e funzioni. Metterli insieme significa creare strutture/*struct,* solitamente verbose (molto lunghe da scrivere), difficili da mantenere e con tanti parametri.
Mancando l'information hiding, possiamo facilmente accedere ad altri dati che non ci interessano, complicando il testing (dati dipendenti tra di loro implicitamente e con interferenze gli uni con gli altri).

Nella OOP, i dati sono implementati con dei *comportamenti/behaviours*, la cui azione non è quella di organizzare i comportamenti e i dati in strutture, riducendo al minimo le dipendenze tra loro.
L'obiettivo è di nascondere le informazioni all'esterno; per costruire un tipo sulla base dell'information hiding, dovremmo:

- Trovare procedure che condividono gli stessi input
- Prendendo l'insieme minimo di input comuni → Evitando accoppiamento stretto/tight coupling
- Creare una struttura usando questi input → I dati sono accessibili dappertutto
- Associare la struttura alle funzioni, formando un tipo

I client devono dipendere *solo dal comportamento*, nascondendo i dati in uno scope (visibilità) *privato.*

Si tende ad andare verso le *interfacce* per accedere a determinati comportamenti *nascondendo* le implementazioni e *minimizzando* le dipendenze.
L'ereditarietà è il problema più grande usando le classi, con dei tipi (richieste a cui rispondere), con l'obiettivo di risparmiare codice (code reuse) ereditando il comportamento (subtyping) e nascondendo le implementazioni.

Esempio di codice che cerca di adattarsi ad una serie di funzionalità multiple (leggere formati di file e framework e relativi percorsi/permessi) a lato.

Qui a lato, il fatto di avere metodi omonimi e con nomi simili e implicitamente legati fra di loro può portare a comportamenti inattesi. Il codice è anche parecchio verboso e difficilmente manutenibile.

```
class AlgorithmThatReadFromCsvAndWriteOnMongo(filePath: String,
                                              mongoUri: String) {
  def read(): List[String] = { /* ... */ }
  def write(lines: List[String]): Unit = { /* ... */ }
}
class AlgorithmThatReadFromKafkaAndWriteOnMongo(broker: String,
                                                topic: String,
                                                mongoUri: String)
  extends AlgorithmThatReadFromCsvAndWriteOnMongo(null, mongoUri) {
  def read(): List[String] = { /* ... */ }
}
class AlgorithmThatReadFromKafkaAndWriteOnMongoAndLogs(brk: String,
                                                       topic: String,
                                                       mongoUri: String,
                                                       logFile: String)
  extends AlgorithmThatReadFromKafkaAndWriteOnMongo(broker, topic,
                                                    mongoUri) {
  def write(lines: List[String]): Unit = { /* ... */ }
}
```

Questo tipo di programmazione ad oggetti è totalmente ridondante, dovendo sempre trasportarsi dietro un contesto, dato che le classi figlie hanno una dipendenza forte dalle classi parent/genitori.

Noi non riutilizziamo comportamento, ma cerchiamo solo di riutilizzare il codice (in questo caso, questo riutilizzo aumenta la *dipendenza* delle classi). Qui abbiamo una *forte dipendenza* dalle classi genitore.
Infatti, si dovrebbe mantenere il principio di ereditarietà dai soli tipi astratti (realizzando uno dei principi SOLID, SRP – Single Responsibility Principle, esplorato a fine corso di Cardin).

L'ereditarietà delle classi può potenzialmente rompere l'incapsulamento in alcuni modi (per questo viene considerata *da evitare*).

- L'ereditarietà può esporre i dettagli di implementazione della classe genitore alla classe figlio, il che può rendere più difficile modificare l'implementazione della classe genitore senza influenzare la classe figlio
    - Le classi espongono due interfacce diverse, pubbliche e protette; le sottoclassi possono accedere allo stato interno delle classi base
- L'ereditarietà può anche creare uno stretto accoppiamento tra le classi genitore e figlio, rendendo più difficile modificare o estendere il software in futuro.
- L'ereditarietà può portare alla creazione di gerarchie di classi ampie e complesse, che possono rendere il software più difficile da comprendere e da mantenere.
    - Sempre più client per una classe

L'ereditarietà definisce l'implementazione di un oggetto in termini di un altro oggetto. In breve, è un meccanismo per la condivisione di codice e di rappresentazione. Al contrario, l'ereditarietà delle interfacce (o *subtyping/sottotipizzazione*) descrive quando un oggetto può essere usato al posto di un altro (questo dato da una *relazione* tra i due tipi, tale da avere un *sottotipo* ed un *supertipo*).

- Ereditare solo dalle interfacce e dalle classi astratte
    - Non sovrascrivere i metodi
    - Non nascondere le operazioni di una classe genitore
- Accoppiamento stretto (loose coupling)
    - I client rimangono ignari del tipo specifico
    - Il polimorfismo dipende dal subtyping

È possibile ottenere accoppiamento lasco (loose coupling) nel subtyping attraverso l'uso di interfacce/classi astratte che definiscono un contratto che il sottotipo deve rispettare.
In particolare, è bene privilegiare la composizione/composition rispetto all'ereditarietà. Letteralmente quest'ultima significa che "non per forza abbiamo bisogno della classe base, dipendendo da essa e dovendo fare override di metodi o dipendere da parametri/metodi che non usiamo; invece, usiamo delle interfacce, chiamando i tipi e le classi quando mi servono senza essere accoppiato a queste ultime". Di fatto, *le classi figlie non possono esistere senza la classe padre*.

È un modo per creare relazioni tra gli oggetti, componendoli insieme per formare un nuovo oggetto. Invece di ereditare da una classe madre, un oggetto è composto da uno o più altri oggetti, chiamati "componenti". L'oggetto può utilizzare gli attributi e il comportamento dei componenti attraverso la delega o l'aggregazione. La composizione consente agli sviluppatori di creare oggetti flessibili e modulari che possono essere facilmente composti e riutilizzati in contesti diversi.
Questo permette di assemblare funzionalità in nuove caratteristiche senza conoscere i dettagli interni (*black box reuse*).

```
trait Reader {
  def read(): List[String]
}
trait Writer {
  def write(lines: List[String]): Unit
}
class CsvReader(filePath: String) extends Reader { /* ... */ }
class MongoWriter(mongoUri: String) extends Writer { /* ... */ }

class Migrator(reader: Reader, writers: List[Writer]) {
  val lines = reader.read()
  writers.foreach(_.write(lines))
}
```

Un *trait* è un tipo che definisce un insieme di metodi e campi astratti. Può anche fornire implementazioni concrete per alcuni di questi metodi. Simile alle interfacce, spesso usati per definire un insieme di comportamenti che possono essere mescolati alle classi

Questo è un esempio di composizione, poiché la classe Migrator è composta da un Reader e da un List[Writer]. La composizione è un modo per creare oggetti complessi combinando oggetti più semplici. In questo caso, la classe Migrator combina un Reader e un List[Writer] per eseguire la migrazione dei dati.

Nel complesso, questo codice sembra essere un esempio di composizione ben progettato. I tratti Reader e Writer definiscono astrazioni per la lettura e la scrittura dei dati e la classe Migrator combina queste astrazioni per eseguire la migrazione dei dati.
Ciò consente flessibilità e riutilizzo, poiché diverse implementazioni concrete dei tratti Reader e Writer possono essere inserite nella classe Migrator a seconda delle necessità.

L'ereditarietà va usata con cautela; in particolare va rispettato il Liskov Substitution Principle (altro principio SOLID, si guardi la lezione apposita) per cui:
"Le funzioni che utilizzano puntatori o riferimenti a classi base devono essere in grado di utilizzare oggetti di classi derivate senza saperlo". Per fare ciò:

- Non sovrascrivere le pre- e post-condizioni della classe base
    - Le precondizioni devono essere più deboli e le postcondizioni devono più forti rispetto a quelle della classe base.
- Progettare per contratto (design by contract) → Meglio spiegata nella lezione dei principi SOLID
    - Evitare la ridefinizione di comportamenti pubblici estrinseci, ma attenersi alle specifiche date dal codice (pre e post condizioni in modo preciso/atteso)

Alcune conclusioni:

- Definire le classi in termini di messaggi
- Non dipendere mai dallo stato interno
- Non usare l'ereditarietà delle classi
- Privilegiare la composizione rispetto all'ereditarietà
- Progettare per contratto
- Utilizzando l'ereditarietà e l'information hiding abbiamo costruire una procedura per definire i tipi in OOP

Dando una veloce sintesi dell'approfondimento slide di Cardin (citato dato che un poco di tempo ce lo ha passato sopra alla slide, per motivi che si capiranno in seguito, legati agli UML):

- Le *modifiche* di un componente possono influenzare le sue dipendenze
    - Modifiche interne: implementazione
    - Modifiche esterne: interfaccia o comportamento estrinseco.
- La dipendenza è una misura della *probabilità* di cambiamenti tra componenti dipendenti
    - Più forte è la dipendenza, più alta è la probabilità

Una misura del grado di dipendenza:

- Accoppiato in modo stretto/Tightly-coupled: maggiore probabilità di cambiamenti
- Accoppiato in modo lasco/Loosely-coupled: minore probabilità di modifiche

La dipendenze vanno minimizzate, rendendo i componenti accoppiati in modo lasco; in questo modo, si ha libertà di modificare un componente, senza introdurre bug (con modifiche interne/esterne).
Le architetture sono dinamiche e si evolvono nel tempo; esistono infatti varie dipendenze tra i tipi.

In particolare, abbiamo quelle tra:

- Classi concrete/astratte/interfacce:
    - Dipendenza (relazione)
        - Tipo *più debole di dipendenza*
        - Quando gli oggetti di una classe lavorano brevemente con oggetti di un'altra classe
            - Limitata nel tempo: esecuzione di un solo metodo
            - Limitata nel codice condiviso: si ha solo un'interfaccia
    - Associazione
        - Quando gli oggetti di una classe lavorano con gli oggetti di un'altra classe per un periodo di tempo prolungato (hanno riferimento/reference ad un oggetto)
            - Si ha per tutta la durata di vita/lifetime di un oggetto → Permanente
    - Aggregazione

        - Quando una classe possiede ma condivide un riferimento (reference) ad oggetti di un'altra classe
            - La creazione e la cancellazione di responsabilità non è cosa semplice
    - Composizione
        - Quando una classe contiene oggetti di un'altra classe
            - Evitare la condivisione di componenti
    - Ereditarietà
        - Quando una classe è un tipo di un'altra classe
- Linee di codice e il tempo di esecuzione (scope)
    - Tipo *più forte* di dipendenza
        - Ereditarietà e riutilizzo del codice non privato
        - Qualsiasi modifica al genitore può disturbare i suoi figli

Maggiore è il codice condiviso, più forte è la dipendenza. L'accoppiamento è proporzionale alla probabilità di cambiamento reciproco tra i componenti.
La misura dell'accoppiamento totale per un componente è la media di tutte le misure di accoppiamento.

Processi di ciclo di vita (Vardanega): continuazione e conclusione slide

Tornando alla definizione di *processi*, abbiamo che sono un insieme di attività collegate e coese che trasformano bisogni in prodotti. Qualifichiamo però l'insieme delle attività al loro interno.

Un insieme di attività è:
- *efficiente* quando usa il minimo numero di risorse indispensabili;
    - metrica: produttività (i.e., efficienza produttiva): rapporto tra quantità di prodotto realizzato e risorse utilizzate;
- *efficace* quando è capace di raggiungere l'obiettivo prefissato;
    - metrica: grado di raggiungimento obiettivi interni (del fornitore) o esterni (gradimento del cliente/degli utenti).

La combinazione di efficienza ed efficacia si chiama «*economicità*».

In questo modo, applichiamo dei limiti al nostro campo d'azione ("strade collaudate" o degli strumenti che ci dicano *per certo* che stiamo facendo bene); questi limiti sono gli standard, cataloghi di processi già formati, accettati e seguiti dall'industria, che aiutano a raggiungere l'economicità (in modo consistente).
Distinguiamo due tipi:
- Standard generali (base per tutti coloro che creano software)
    - ISO/IEC 12207:1995 e sue evoluzioni (Software life cycle processes)
        - https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf
- Standard settoriali: per specifici domini applicativi (di seguito alcuni esempi)
    - IEC 880 : settore nucleare
    - RTCA DO-178 : settore aeronautico
    - ECSS E40: settore spaziale

Ogni standard ha due funzioni complementari:
1) Modello di azione
    a. Fissa quali attività svolgere e come farlo: visione *prescrittiva*
    b. Specifica cosa serva fare, lasciando libertà sul come: visione *descrittiva*
2) Modello di valutazione
    a. Identifica "*best practices*" (pratiche buone, da seguire) di riferimento: la regola d'arte
    b. Permette di misurare quanto un particolare *way of working* disti da essa (soprattutto, chi cerca di avvicinarsi alle regole allo stato dell'arte, sperando di saperlo prima)

Gli standard:

- da un lato danno una guida su *come fare* le attività (modello)
- dall'altro, servono come strumenti di valutazione sulla *way of working* rispetto allo stato dell'arte

La misura del way of working si chiama qualità, quindi la "vicinanza alle best practices", misurando principalmente la distanza tra quanto fatto e l'ideale stato dell'arte che gli standard forniscono.
ISO/IEC 12207:1995 è il modello più noto e riferito nell'ambito del ciclo di vita dei processi software, specificando i processi che devono essere seguiti durante lo sviluppo, mantenimento e ritiro di sistemi software, compresi ruoli/responsabilità degli stakeholder (migliorando la collaborazione/comunicazione con essi). Mantiene una struttura modulare, specializzandosi caso per caso.

Esso è "ad alto livello", composto da quattro parti principali:

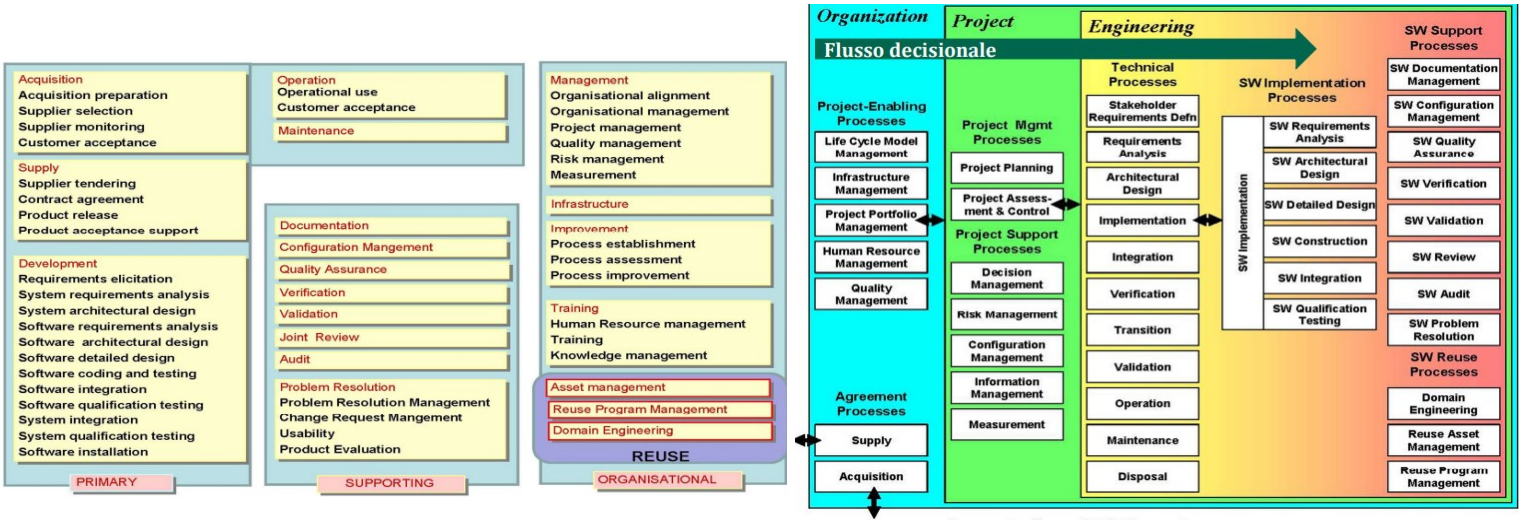
- Introduzione: Fornisce una panoramica dello standard e del suo scopo.
- Processi del ciclo di vita: Descrive i processi da seguire durante le diverse fasi del ciclo di vita del software, tra cui la raccolta dei requisiti, la progettazione, l'implementazione, il collaudo, la distribuzione e la manutenzione.
- Processi di supporto: Definisce i processi che supportano i processi principali del ciclo di vita, come la gestione della configurazione, l'assicurazione della qualità e la gestione del rischio.
- Tecniche di supporto: Descrivono le tecniche e gli strumenti che possono essere utilizzati a supporto dei processi del ciclo di vita, come la prototipazione, i metodi di test e gli standard di documentazione.

Graficamente, qui di seguito:

- I processi *primari* sono i processi principali che vengono seguiti durante il ciclo di vita di un sistema software. Questi sono responsabili della creazione del sistema software e comprendono attività quali la raccolta dei requisiti, la progettazione, l'implementazione, il collaudo e la distribuzione.
    - Processi primari: Acquisizione, Fornitura, Sviluppo

- I processi *di supporto* sono processi che sostengono i processi primari, fornendo attività o funzioni aggiuntive necessarie per lo sviluppo e la manutenzione del sistema software.
    - Processi di supporto: Manutenzione, Documentazione, Verifica, Validazione, Configurazione

- I processi *organizzativi* sono processi specifici di un'organizzazione e non sono direttamente correlati allo sviluppo o alla manutenzione del sistema software. Questi processi possono includere attività come la gestione dei progetti, la gestione finanziaria e la gestione delle risorse umane.
    - Processi organizzativi: Formazione, Infrastruttura

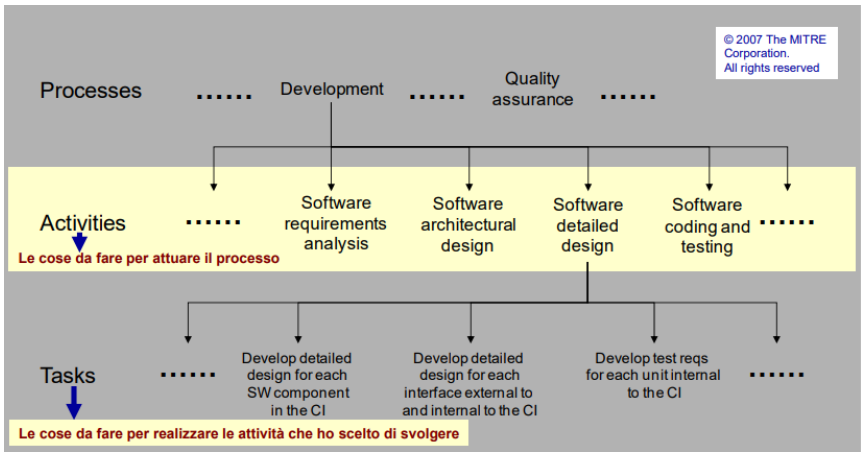


I processi di sviluppo software comprendono, in sé, più compiti, dato che il progetto si dice collaborativo per definizione. Anche i processi sono svolti collaborativamente (connessione sia con altre attività sia con processi insiti alle attività stesse). Se più persone collaborano per lo svolgimento delle attività, esse non si devono sovrapporre.

Quando questo succede, è possibile che l'efficienza diminuisca grandemente (si spreca tempo, in quanto qualche componente non ha "nulla da fare"). Al fine di mantenerla, si può pensare di scomporre le attività in pezzi più piccoli, rendendoli elementari (atomici, non divisibili). Più lunga è l'attesa, peggio è.

Come si vede anche sotto, distinguiamo:

- Le attività come compiti "a grana grossa", utili per attuare il processo, che comprendono passi multipli/sottocompiti per ottenere l'obiettivo disposto
- I task come compiti "a grana fine", utili per realizzare le attività che ho scelto di svolgere. Sono concentrati su un singolo risultato e possono essere assegnati a singoli membri del team/gruppi



Esempio di attività di processo:

§5.3 Sviluppo SW
- .1 Istanziazione del processo
- .2 Analisi dei requisiti del sistema
- .3 Progettazione architetturale del sistema
- .4 Analisi dei requisiti del SW
- .5 Progettazione architetturale del SW
- .6 Progettazione di dettaglio del SW
- .7 Codifica e prova dei componenti SW
- .8 Integrazione dei componenti SW
- .9 Collaudo del SW
- .10 Integrazione di sistema
- .11 Collaudo del sistema

Esempio di task:

Codifica e prova dei componenti SW §5.3.7
- Definire procedure e dati di prova .1
- Eseguire e documentare le prove .2
- Aggiornare documentazione e pianificare prove d'integrazione .4
- Valutare l'esito delle prove .5

Integrazione dei componenti (sistema) §5.3.8
- Definire il piano di integrazione .1
- Eseguire e documentare le prove .2
- Aggiornare documentazione e pianificare prove di collaudo .4
- Valutare l'esito delle prove .5

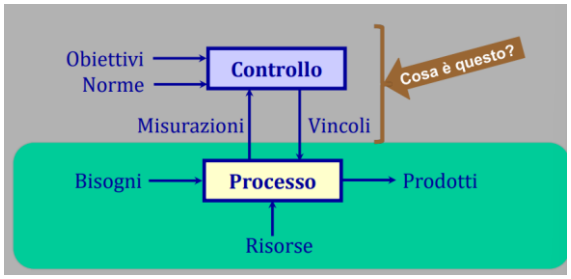
Collaudo del SW §5.3.9
- Eseguire e documentare il collaudo .1
- Valutare l'esito del collaudo .3

Integrazione del sistema §5.3.10
- Eseguire e documentare le prove .1
- Aggiornare documentazione e pianificare prove di collaudo .2
- Valutare l'esito delle prove .3

Collaudo del sistema §5.3.11
- Eseguire e documentare il collaudo .1
- Valutare l'esito del collaudo .2

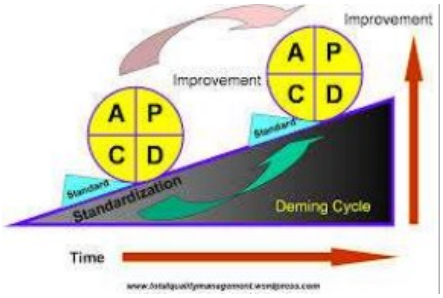
I processi possono essere specializzati secondo alcuni fattori:
- Dimensione e complessità del progetto
- Tipo e intensità dei rischi di progetto
    - Nel dominio applicativo, nel rapporto con clienti e utenti, nella maturità/complessità delle tecnologie in uso
- Competenza ed esperienza delle risorse umane

Associare ai processi un *sistema di qualità* aiuta a migliorarli e a garantire *conformità*:
- *Maturità* = qualità misurata delle prestazioni
- *Conformità* = adesione alle aspettative e agli obblighi



Il <u>controllo</u> di processo contente di attuare manutenzione migliorativa al proprio way of working (fare le cose al meglio possibile).
Si ha il *principio del miglioramento continuo* (a salita continua), in cui ogni attività (poiché misurabile) è confrontabile con le attese.

Per realizzare questo principio, si ha un ciclo a 4 stadi nato per apportare migliorie, noto come *Shewhart-Deming's Learning-and-Quality Cycle/PDCA Cycle*, così articolato (in quanto costituito da pezzi ripetibili):
- Pianificare (*Plan*)
    - Definire attività, scadenze, responsabilità, risorse per raggiungere specifici obiettivi
    - Essa non è pianificazione, ma variazione del way of working per migliorare la qualità
- Eseguire (*Do*)
    - Eseguire le attività secondo la pianificazione data
- Valutare (*Check*)
    - Verificare l'esito delle azioni di miglioramento rispetto alle attese
    - Questo si fa in modo continuativo, non si fa tramite testing sul SW prodotto
- Agire (*Act*)
    - Consolidare il buono e cercare modi per migliorare il resto



Eseguendo e facendo, si eseguono piccoli e costanti passi di miglioramento, utili a livello produttivo, sempre spostandosi in avanti. Si individuano in particolare le debolezze, pianificando le attività di miglioramento e identificando dove migliorare.
Il ciclo PDCA non è la normale sequenza di attività di progetto, in quanto non *opera* sul prodotto, ma *sul way of working*, per migliorarlo.

Il ciclo di vita del SW (inizio slide) (Vardanega)

(Eventuali approfondimenti:
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9121630
https://www.agilealliance.org/agile101/
https://www.ivarjacobson.com/files/field_iji_file/article/scrum_teams_improving_with_essence_0.pdf)

Il SWE/Software Engineering spiega come svolgere le attività in un modo che abbia sempre 3 caratteristiche (qui classificate non in ordine di importanza, riprese dalla detta slide 19 riguardo alla telemetria):

1) Sistematico
2) Disciplinato
3) Quantificabile

Idealmente, si vede come "cruscotto", cioè esaminiamo pezzo per pezzo tutte queste caratteristiche, verificando quando siamo distanti dall'obiettivo (telemetria).
Questo permette di controllare le transizioni tra stati di processo, che avviene per azione di processi di ciclo di vita, facendo progredire lo stato di avanzamento di un prodotto SW.
Per farlo, il progetto mobilita specifiche attività di processo, ordinate secondo le dipendenze tra ingressi/uscite, fissando le pre/post condizioni necessarie.

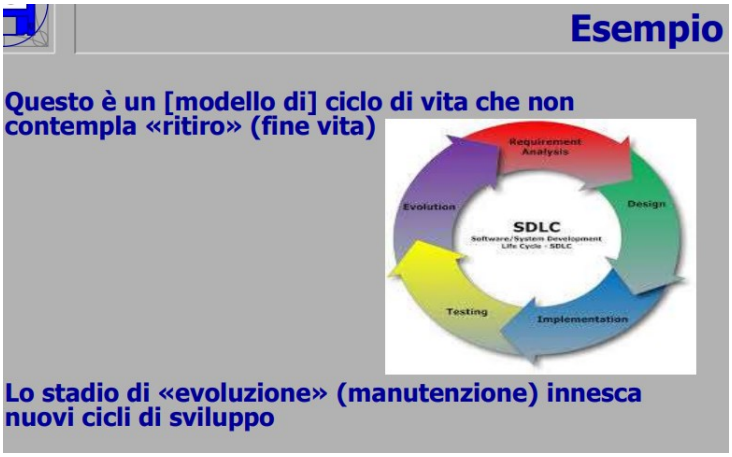
Il ciclo di vita di un progetto è articolato in varie fasi (a noi interessa il segmento [concezione → sviluppo]):

- Concezione (Elaborazione a priori di tutto ciò che il progetto deve fare)
- Sviluppo (Trasformazione dei needs in un utensile tramite l'implementation)
- Utilizzo (Stadio di vita del prodotto, quindi il fornitore garantisce che il prodotto continui ad essere usato tramite la manutenzione)
- Ritiro (Usare qualcosa che tecnicamente funziona ma non è più adatto)

L'obiettivo di un progetto è progredire come stato di avanzamento di prodotto SW.

Il termine fase corrisponde alla permanenza del prodotto software in uno stato o in una transizione, in particolare un tempo continuo entro il quale si eseguono attività in modo univoco e specifico.

Esistono *molteplici* modi per passare e spostarsi tra i cicli di vita, che differiscono per stati/regole; essi sono definiti come modelli di sviluppo. Questi definiscono specifici vincoli sulla pianificazione/gestione del progetto, influenzando il way of working e gli strumenti utilizzati per normarlo.



Un modello viene definito come insieme di specifiche che descrivono un fenomeno di interesse (astratto / concreto), in modo che non dipenda dall'osservatore ed è dimostrato corretto, in modo empirico o formale. Essi sono rappresentazioni astratte di un sistema/di un concetto e semplificano sistemi complessi, aiutando a capire relazioni ed interazioni tra le componenti.
Ne esistono di diversi tipi:

- Modelli di requisiti: Descrivono i requisiti funzionali e come vengono usati nel sistema (*perché*)
- Modelli di progettazione: Descrivono la struttura e l'organizzazione adottate nel design (*cosa*)
- Modelli comportamentali: Descrivono il comportamento/le interazioni tra le parti (*come*)

Il *cowboy coding* è un termine usato per descrivere uno stile di sviluppo del software caratterizzato dalla mancanza di pianificazione, organizzazione e disciplina. Nel cowboy coding, gli sviluppatori lavorano in

modo non strutturato, ignorando le best practices e gli standard a favore di soluzioni rapide e approssimative (meccanismo *code-'n-fix*).

Questo approccio allo sviluppo del software può portare a una serie di problemi, tra cui la scarsa qualità del codice, la difficoltà di mantenere e aggiornare il software e la difficoltà di collaborare con gli altri membri del team. Quello stile causò la crisi del SW, in quanto non si aveva un vero e proprio standard; questo portò alla nascita della disciplina SWE.

Esistono una serie di modelli organizzati come segue:

Modello	Tratti caratteristici
Cascata	Rigide fasi sequenziali
Incrementale	realizzazione in più passi
A componenti	Orientato al riuso
Agile	Altamente dinamico, fatto di brevi cicli iterativi e incrementali

Glossario utile in questo contesto:

- Iterazione
  - Ripetizione di una sequenza di attività al fine di raggiungere un obiettivo desiderato
- Incremento
  - Aggiungere nuove funzionalità o caratteristiche a un sistema software in modo sistematico
- Prototipo
  - Modello preliminare di un sistema software che viene utilizzato per testare e dimostrare la funzionalità di base del sistema stesso
- Riuso
  - La pratica di utilizzare codice, componenti o altre risorse esistenti nello sviluppo di un nuovo sistema software. Il riutilizzo può contribuire a ridurre i tempi e i costi di sviluppo e a migliorare la qualità e l'affidabilità del software.

Definiamo quindi i singoli tipi di modelli:

1) Modello sequenziale (a cascata/*waterfall*), è un processo lineare in cui lo sviluppo scorre in modo discendente attraverso le fasi di raccolta dei requisiti, progettazione, implementazione, test, distribuzione e manutenzione.
   - Le fasi sono *rigidamente sequenziali*, in cui il design è guida all'implementation, dicendoci esattamente cosa deve essere fatto. Non ammette ritorno a fasi precedenti, in quanto le iterazioni costano troppo.
   - Ogni fase deve essere completata prima di poter iniziare la fase successiva (precondizioni/*gates*) e la sovrapposizione tra le fasi è minima o nulla.
   - Si centra sull'idea di processi ripetibili, basandosi sulla successione di fasi distinte (quindi adatto allo sviluppo di sistemi complessi)
   - I prodotti sono principalmente *documenti*, non tanto codice, fino poi ad includere il SW.
   - Ogni fase ha attività previste e prodotti attesi, struttura di documenti che descrivono lo stato raggiunto, ruoli e scadenze di attività e prodotti
     - Entrare, uscire, stazionare in una fase comporta lo svolgimento di determinate azioni

Quanto tutti i needs sono ben definiti (*once and for all*), allora inizia *l'analisi dei requisiti*:

Il fatto che i needs siano definiti tutti all'inizio è utopistica; quando comincia ad emerge un concetto ed un eventuale prototipo, non si ha qualcosa di adattabile all'uso da parte degli utenti.
In particolare, tutte le funzioni realizzative sarebbero già state pensate (deve essere provato, altrimenti è idea solo sulla carta).

Da questo punto di vista, il modello risulta troppo fisso (*critica del modello sequenziale*).

- Difetto principale: eccessiva rigidità
  - Stretta sequenzialità: nessun parallelismo e nessun ritorno (ci si impiega molto tempo)
  - Non ammette modifiche nei requisiti in corso d'opera; questo comporta un prodotto che non rispetta a pieno i requisiti utente (quindi, limitata visibilità)
  - Cattiva gestione degli errori (non ammettendo iterazione)

Per correggere tale rigidità sono stati introdotti dei modelli ibridi di due tipi:

- Correttivo 1: con prototipazione (avanzamento concettuale ma non di risultati)
  - Prototipi di tipo "usa e getta" (throw away)
    - Per capire meglio i requisiti o le soluzioni (ma non come risolvere i problemi)
    - Strettamente all'interno di singole fasi
- Correttivo 2: con ritorni (si accorge che si sbaglia e, progressivamente, corregge)
  - Come «allenamenti» prima dell'atto definitivo, per imparare a fare sempre meglio ciò che serve a realizzare il prodotto
  - Ciò comporta ripensare fasi e relative scadenze, per contro

I ritorni dovrebbero condurre ad un miglioramento e possono essere di tipo iterativo o incrementale. Tutto dipende dalla specifica capacità di adattamento, considerando che le iterazioni sono distruttive e il modo migliore è decomporre la realizzazione del sistema in parti critiche e integrare pezzi piccoli, sapendo che funzionano (e riducendo la possibilità di fallimento). Questa pratica è l'incremento, prevedendo rilasci multipli e ripetuti, adattandosi progressivamente.

Sconsigliato integrare il prodotto tutto-in-una-volta, mettendo tutti i moduli insieme (big-bang-integration), spendendo molto più tempo. Assai meglio adottare l'integrazione continua/continuous integration/CI, con iterazioni di avanzamento che portano a migliorare progressivamente, assicurandosi che ogni cambiamento sia utile e non introduca regressione. Queste coincidono con l'incremento quando la sostituzione raffina ma non ha impatto sul resto.

Questo viene garantito da due tipi di modelli: incrementali e iterativi. Questi sono approcci allo sviluppo del software che prevedono la consegna di piccoli incrementi di un progetto, chiamati "sprint", su base regolare, con l'idea di apportare frequenti e piccoli miglioramenti a un progetto.
Nei modelli incrementali (vantaggi):

- Possono produrre valore a ogni incremento
  - Un insieme crescente di funzionalità utili diventa presto e progressivamente disponibile
  - Questo permette di ottenere subito feedback dagli stakeholder

- Procedere per incrementi riduce il rischio di fallimento
    - Si passa per ogni fase producendo continuativamente valori, riducendo il rischio di errore e correggendo dopo aver prodotto un risultato
- Le funzionalità fondamentali (più necessarie) vanno sviluppate prima
    - Il loro uso frequente aiuta a verificare che siano solide

Tuttavia, abbiamo anche svantaggi da riportare per i metodi incrementali:
- Possono essere più complessi da gestire, richiedendo frequente comunicazione
- Sono più difficili da pianificare/stimare e richiedono più risorse, di test e non solo
- Maggiore complessità nella gestione, richiedendo cambiamenti/aggiornamenti di frequente

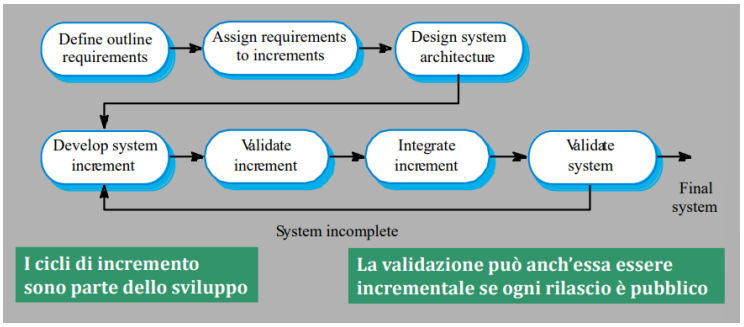
Nei modelli iterativi (vantaggi):
- Applicabili a qualunque modello di sviluppo
    - Ma comportando forte potenziale distruttivo
- Consentono maggior capacità di adattamento
    - Insorgere di problemi, cambio di requisiti, collasso tecnologico

Tuttavia: (svantaggi)
- Comportano il rischio di non convergenza
    - Come un ciclo *while*, da cui non sappiamo per certo se e quando usciremo
- Si devono implementare tecniche di mitigazione
    - Decomporre il sistema in parti, lavorando prima su quelle più critiche, perché più complesse o con requisiti più incerti
    - Fissando un limite superiore al numero di iterazioni
- Ogni iterazione comporta un ritorno all'indietro nella direzione *opposta* all'avanzamento del tempo

Iterativamente, cerchiamo di eseguire il cosiddetto *refactoring*, "passando e migliorando iterativamente all'interno senza modificare l'esterno"; il costo dell'utilizzo di scorciatoie o di decisioni progettuali non ottimali durante lo sviluppo di un sistema software può essere dannoso. Queste possono facilitare lo sviluppo del software nel breve termine, ma può essere molto costoso, in quanto alla lunga rende difficile la manutenzione del software (potrebbe richiedere molto lavoro, a mano a mano che si "accumula", ricorreggere e scalare correttamente, per brevi/lunghi periodi/obiettivi) → technical debt.

Parliamo ora nel dettaglio dei modelli incrementali, i quali prevedono rilasci multipli e successivi, in cui ciascuno realizza un incremento di funzionalità. I *requisiti* sono classificati e trattati in base alla loro importanza strategica (si scoprono tutti avanzando nel design e nell'implementation):

- i primi incrementi puntano a soddisfare i requisiti più importanti/più evidenti sul piano strategico (nascondo idee interne al prodotto);
- i requisiti importanti diventano presto chiari e stabili e più facilmente soddisfacibili;
- i requisiti meno importanti hanno più tempo per stabilizzarsi e armonizzarsi con lo stato del sistema;



In questo modello, analisi dei requisiti e progettazione architetturale vengono svolte *una sola volta*: ciò è un'idea ambiziosa, perché richiede molta esperienza per poterlo fare bene. Ciò viene fatto:
- Per stabilizzare presto i requisiti principali e *l'architettura* complessiva del sistema
- Per decidere preventivamente il numero di incrementi e i loro specifici obiettivi

La realizzazione è incrementale → Istanziazione del processo e numero di iterazioni prefissato
- Raffinando l'analisi dei requisiti e la progettazione di dettaglio, entro l'architettura adottata
- Il completamento dei primi incrementi serve a rendere disponibili le principali funzionalità

Un modello arrivato dopo e oggi utilizzato combatte l'insensatezza dello sviluppare partendo da zero (*from scratch*) cercando di usare cose già esistenti (*componenti*).
Esso è il modello a componenti, cioè un modo di organizzare la struttura di un sistema software dividendolo in componenti più piccoli e riutilizzabili. Questi componenti sono unità modulari che possono essere sviluppate, testate e mantenute in modo indipendente e possono essere facilmente combinate con altri componenti per creare sistemi più grandi e complessi.

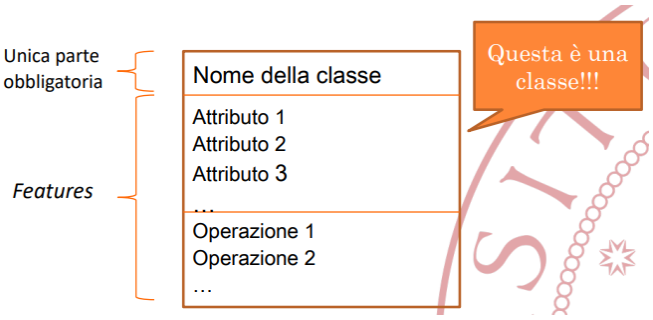
L'uso di un modello a componenti può contribuire a migliorare la manutenibilità e l'estensibilità di un sistema software, in quanto consente agli sviluppatori di apportare modifiche ai singoli componenti senza influenzare il resto del sistema, riutilizzando cose già esistenti.

Diagrammi delle classi (Cardin)

Il linguaggio naturale è poco espressivo nella creazione di un software; questo avviene soprattutto quando si ha a che fare con altre persone, è fondamentale farsi comprendere e comprendersi. Per poter parlare di software, usiamo UML (Unified Modeling Language) è un linguaggio di modellazione visuale standardizzato utilizzato per rappresentare e documentare la progettazione di un sistema software. Fornisce un insieme di elementi di notazione grafica che possono essere utilizzati per creare modelli visivi della struttura, del comportamento e delle relazioni di un sistema.

Vediamo i diagrammi delle classi, che sono un tipo di diagramma di struttura statica che rappresenta la struttura di un sistema mostrando le classi del sistema, i loro attributi e le relazioni tra di esse. I diagrammi delle classi sono utilizzati per modellare gli aspetti statici di un sistema, compresa la gerarchia delle classi e le relazioni tra di esse.

Descrizione del tipo degli oggetti che fa parte di un sistema:
- Relazioni statiche fra i tipi degli oggetti

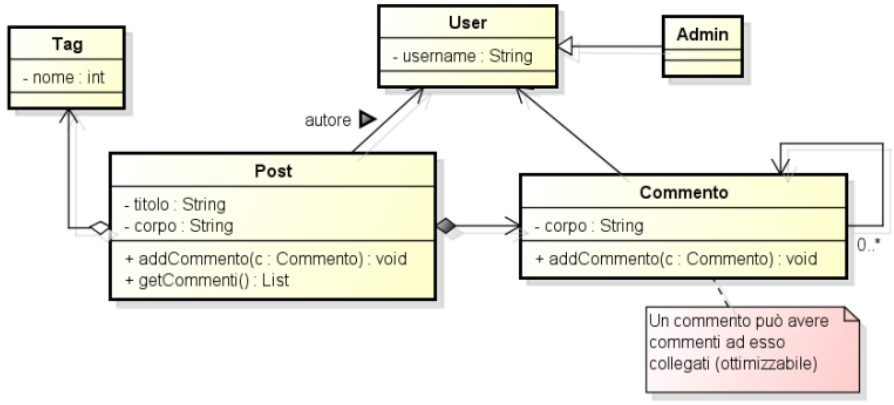


Prendiamo uno strumento che ci permetta di realizzare diagrammi: https://staruml.io/download
Altri strumenti possibili:
- https://app.diagrams.net/
- LucidChart
- Draw.io

Per esempio, proviamo a modellare la classe Movie (nel modo che si vede sotto e a lato, come idea):

Esempio

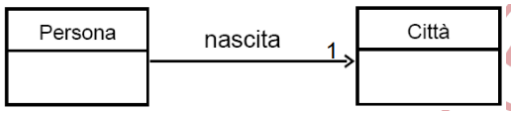
È richiesto lo sviluppo di un'applicazione che permetta la gestione di un semplice blog. In particolare devono essere disponibili almeno tutte le funzionalità base di un blog: deve essere possibile per un utente inserire un nuovo post e successivamente per gli altri utenti deve essere possibile commentarlo. Queste due operazioni devono essere disponibili unicamente agli utenti registrati all'interno del sistema. La registrazione avviene scegliendo una username e una password. La username deve essere univoca all'interno del sistema.



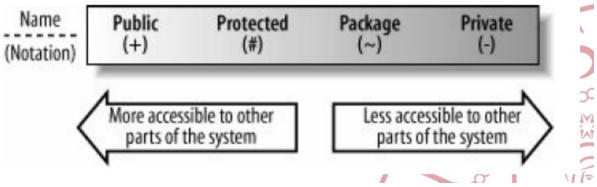
Gli attributi sono caratteristiche *strutturali*, ciascuno con una visibilità (+ pubblica, - privata, # protetta).

```
Visibilità nome : tipo [molteplicità] = default {proprietà aggiuntive}
```

L'associazione è rappresentata da una *linea continua* e orientata fra due classi e viene rappresentata la molteplicità; quindi, quanti oggetti possono far parte dell'associazione, indicata con 1, 0..1, 0..*, *,… spesso *interscambiabile* con un attributo. Normalmente è bidirezionale. Tuttavia (mai detto da Cardin), può esistere la *directed association* (associazione diretta) che è unidirezionale, come si vede a lato.
Ciò intende che "da A si va verso B"; in alcuni contesti, "A usa B".



Essi hanno una certa visibilità:



Per esempio, vogliamo inserire le molteplicità degli attributi, utile per le validazioni, tale da poter tracciare visualmente le dipendenze-

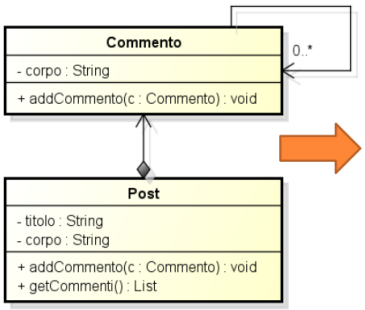
Gli attributi/metodi vengono chiamati *properties* (perché definiscono proprietà aggiuntive), in quanto *membri di classe nel contesto dei linguaggi di programmazione* (solitamente privati); in aggiunta, per convenzione, possiamo avere dei metodi *getter* e *setter* per ogni attributo.

Le associazioni vengono rese tramite un *nome* (meglio fare così piuttosto che etichettare con un verbo) ed è meglio evitare le associazioni *bidirezionali*.

Vengono definite *operazioni* le azioni che la classe sa eseguire (*comportamento*) oppure un *servizio* che ad istanza viene chiesto (query, modificatori); un'operazione *non è* un metodo (a meno che non si parli di polimorfismo).

- Direzione: *in, out, inout* (default *in*).
- Visibilità: + pubblica, - privata, # protetta

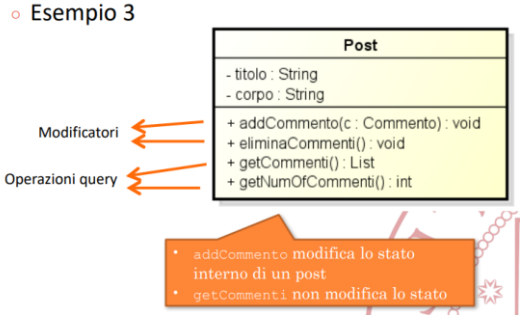
◦ Esempio 2



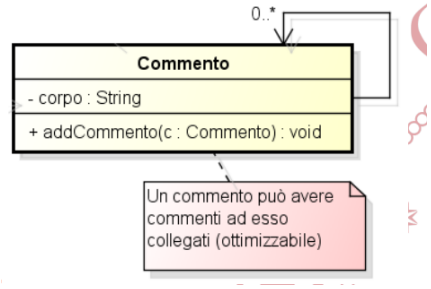

```java
public class Commento {
  private String corpo = null;
  private List<Commento> commenti =
    new ArrayList<Commento>();

  public void addCommento(Commento c) {
    commenti.add(c);
  }
}

public class Post {
  private List<Commento> commenti =
    new ArrayList<Commento>();

  // ...

  public List<Commento> getCommenti()
    {...}
}
```

Un esempio di tipo di operazioni:

◦ Esempio 3

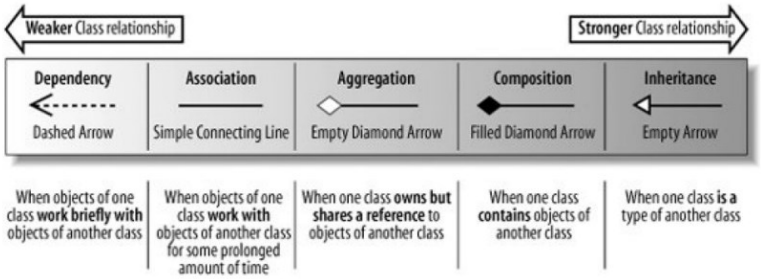


Per esprimere *informazioni aggiuntive* (es. *note/commenti*), esse vengono aggiunte come singole/solitarie, con una linea tratteggiata, generalmente legate ad un elemento grafico (classe) (su StarUML vengono aggiunte nel menù a sx nella penultima categoria tra quelle a tendina (menù *Annotations*, selezionando *Note*). Piccolo esempio a lato.



Si ha una relazione di dipendenza quando "tra due elementi di un diagramma se la modifica alla definizione del primo (fornitore) può cambiare la definizione del secondo (client)". UML permette di modellare ogni sorta di dipendenza e le dipendenze vanno *minimizzate* (*loose coupling*, da inserire solo quando danno valore aggiunto).
La dipendenza vera e propria è tratteggiata e significa che la classe B dipende da A quando ha bisogno di una sua operazione (es. qui, la classe CustomerView ha bisogno di Customer per visualizzare le proprie informazioni):

Ci sono vari tipi di relazione di dipendenza, come segue, ordinati come visto qui:



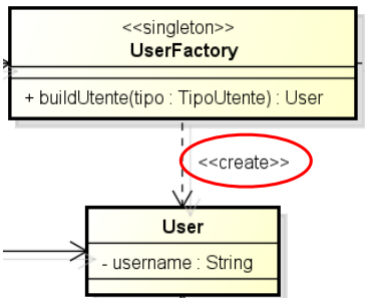
Scritto da Gabriel

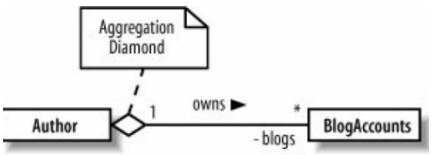
Sempre in merito alle relazioni di dipendenza, si ha che la sua definizione è: "maggiore è la quantità di codice condiviso tra due tipi, maggiore la dipendenza tra essi". Inoltre:

- La dipendenza tra due tipi è direttamente proporzionale alla probabilità di modificare entrambi
- La dipendenza è quindi una funzione di numero SLOC condivise e di ampiezza dello scope del codice condiviso

Listiamo anche le dipendenze UML, in base al loro significato:

Parola chiave	Significato
«call»	La sorgente invoca un'operazione della classe destinazione.
«create»	La sorgente crea istanze della classe destinazione.
«derive»	La sorgente è derivata dalla classe destinazione
«instantiate»	La sorgente è una istanza della classe destinazione (meta-classe)
«permit»	La classe destinazione permette alla sorgente di accedere ai suoi campi privati.
«realize»	La sorgente è un'implementazione di una specifica o di una interfaccia definita dalla sorgente
«refine»	Raffinamento tra differenti livelli semantici.
«substitute»	La sorgente è sostituibile alla destinazione.
«trace»	Tiene traccia dei requisiti o di come i cambiamenti di una parte di modello si colleghino ad altre
«use»	La sorgente richiede la destinazione per la sua implementazione.

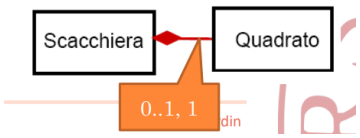




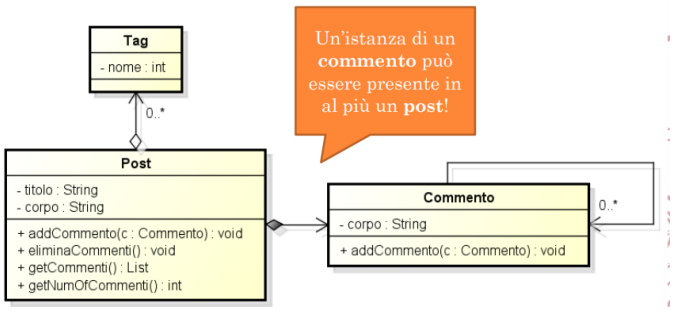
Ci possono essere vari tipi di relazione (le seguenti sono passive; cioè il diamante pieno o vuoto sta dalla parte di chi aggrega/compone, mentre la sottoclasse *è aggregato/è composto* della superclasse).

Consideriamo l'<u>aggregazione</u> (part of/parte di, tale che gli aggregati possano essere condivisi); la classe aggregata è responsabile della costruzione degli aggreganti (diamante vuoto).
Esempio stupido ma utile: Auto e Motore; Motore è aggregato di Auto, mentre Auto aggrega motore.

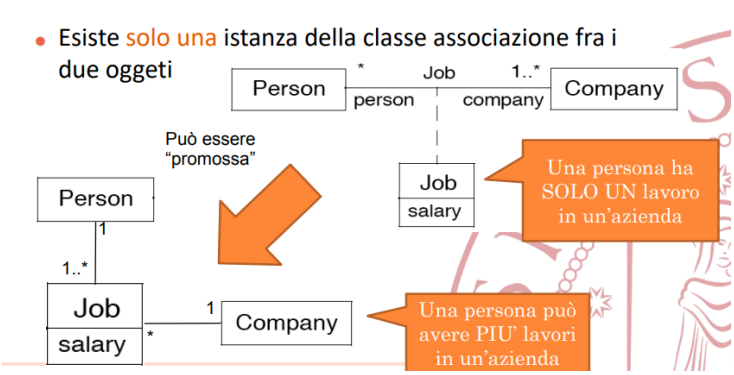
La <u>composizione</u> è simile all'aggregazione (ma ha il diamante pieno), ma gli aggregati appartengono ad un *solo aggregato*. Solo l'oggetto intero può *creare* e distruggere le sue parti.
Le sottoclassi non possono esistere senza la classe che le compone e, quando si utilizza, si intende a livello di codice che usiamo operazioni *solo quando ci servono* nelle sottoclassi, ma dipendono da qualcosa che sta sopra.

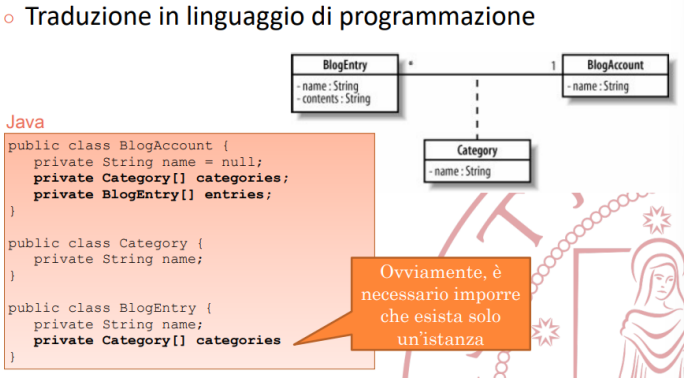


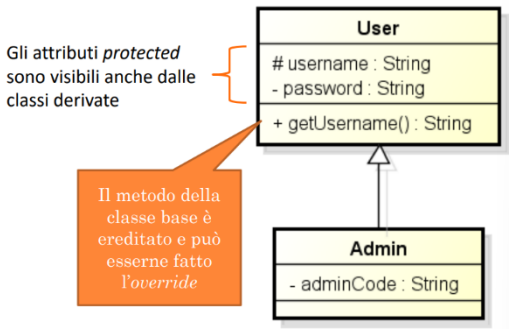
Un esempio secondo le slide di aggregazione e composizione è come segue:



Un altro esempio sono le classi di associazione, che aggiungono attributi e operazioni alle associazioni.

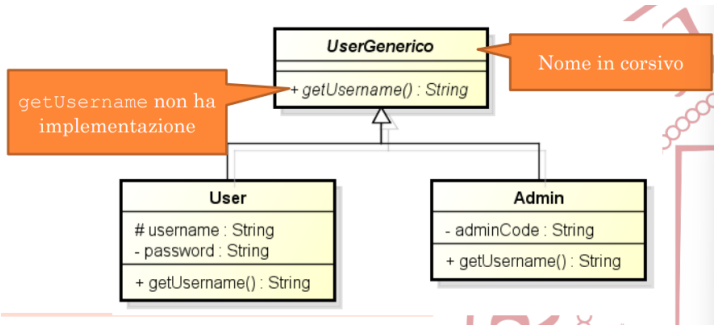


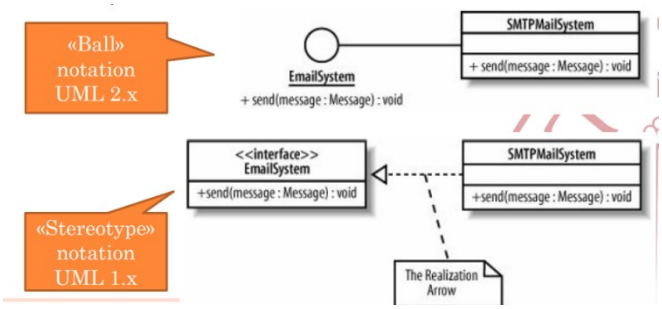




Similmente, si hanno generalizzazioni ad esempio quando con due classi A e B, ogni oggetto di B è anche un oggetto di A e ciò equivale all'ereditarietà dei linguaggi di programmazione. Le proprietà della superclasse non si riportano nel diagramma della sottoclasse (a meno di override); si ha il principio di sostituibilità (sottotipo != sottoclasse), che permette di differenziare le interfacce in base alla loro implementazione.

Prendiamo poi le classi astratte, che ricordiamo essere classi che *non possono essere istanziate*; le operazioni astratte non hanno implementazione e queste vengono definite con nome in corsivo. Esse non devono essere molte, anzi, tendenzialmente sono poche (per non confondere l'UML).





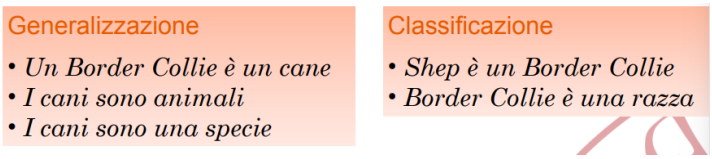
Similmente, abbiamo le interfacce, classi prive di implementazioni; una classe realizza un'interfaccia se ne implementa le operazioni. Vengono definite come <<interface>> (le parentesi angolate in UML sono definite stereotipo/keyword).
La palla è UML2, mentre <<interface>> è UML1.

Di fatto, purtroppo mai citata da Cardin, esiste anche la *Interface realization*, che letteralmente è una generalizzazione con i trattini (tratteggiata) e che indica che le sottoclassi implementano una interfaccia con delle operazioni. In UML2 viene indicata con una linea come l'associazione, collegando sempre la freccia dalla sottoclasse verso l'interfaccia.
Letteralmente, la interface realization, si può vedere sopra nell'esempio, sia in forma tratteggiata (UML1) che forma non tratteggiata (UML2). Si può interpretare come una generalizzazione per le sottoclassi di un'interfaccia.

Differenza tra *extends* ed *implements* (qui usiamo Java, ma l'idea è questa in generale):

- *implements* significa che si utilizzano gli elementi di un'interfaccia Java nella propria classe
- *extends* significa che si crea una sottoclasse della classe base che si sta estendendo
- È possibile estendere solo una classe nella propria classe figlio, ma è possibile implementare tutte le interfacce che si desidera.

Un sottotipo è diverso da un *is-a*, dato che a livello generale hanno le stesse caratteristiche, ma un sottotipo normalmente viene solo istanziato, un *is-a* possiede sempre le caratteristiche che servono; ciò viene visto come dipendenza *<<istantiate>>*.
Distinguiamo in particolare generalizzazione da classificazione:

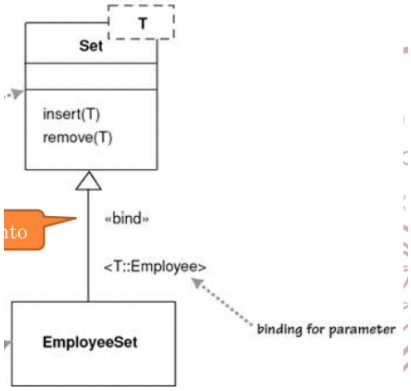


- la generalizzazione è transitiva (la classificazione invece no)
- la classificazione si esprime con la dipendenza <<istantiate>>

Cose varie:

- Gli attributi statici, applicati alla classe, non all'oggetto, sono sottolineati sul diagramma
  - Vengono definiti con un'estensione semantica UML (costrutto simile + parola chiave
    - Es. <<interface>>, {abstract}
  - Offrono funzionalità e vengono aggiunte alla classe come commento
- Le proprietà derivate, che possono essere calcolate a partire da altri valori, definiscono un vincolo fra valori e sono indicate con "/" che precede il nome della proprietà
- Le proprietà *readonly* non vengono fornite nei servizi di scrittura e si indicano con {readonly}
- Le proprietà *frozen* rimangono immutabili (non variano durante il ciclo di vita) e si indicano con {frozen}
- Le enumerazioni non hanno altre proprietà oltre il valore simbolico e si indicano con <<enumeration>>.

Le *classi Parametriche* hanno *T* come "segnaposto"; con parametrico si intende il concetto di *template* in C++ o di *generics* in Java, tale che per ereditarietà si possa per binding realizzare un'operazione collegata (es. setting dell'impiegato/employee, specificando dove sto realizzando l'operazione).



Su StarUML si fa tasto destro su una classe e si mette "Add → Template Parameter" per fare "T" come si vede a lato.

Regolamento del Progetto Didattico (Vardanega)

In merito al progetto, ci sono 7 proponenti esterni e ciascuno presenta un proprio *capitolato* che illustra bisogni, vincoli, e suggerimenti (documento narrativo, motivando l'interesse nel dominio d'uso).
Ogni gruppo si candida come *fornitore* di un singolo capitolato di proprio interesse

- Il proponente è cliente rispetto alle esigenze di prodotto.
- Il proponente è mentore rispetto alle scelte di sviluppo

Il docente è *committente* di tutte le forniture. L'interazione fornitore-committente concerne vincoli contrattuali (scadenze, costi) e questioni regolamentari (sia lato cliente, tramite accordi con il proponente, sia lato mentore, figura di riferimento usate il più frequente possibile).

Il progresso nel progetto viene misurato tramite *revisioni di avanzamento*, in cui si deve raccontare al committente cosa è stato fatto, per quali motivazioni, difficoltà incontrate, con quali giudizi, ecc.
Similmente, ci si incontra spesso con l'azienda, in base alle rispettive necessità.
La valutazione segue un modello *Product, Process, Progression in learning*, idealmente crescendo nell'asse delle skills, misurato da vari punti di vista:

- del cliente (percezione e giudizio del way of working)
- misura dell'automiglioramento dopo i confronti
- media rendicontata di ogni componente del gruppo (in base alle ore)

Ogni gruppo fornitore si dota di un way of working, fatta di strumenti e piattaforme collaborative, normando ogni compito in maniera personalizzata e collaborativa (dettagliando l'uso di strumenti nelle *norme di progetto*), tale da comportarsi in modo sistematico e disciplinato.
Esso deve contenere ciò che sarà fatto ad intervalli brevi (*JIT, Just In Time*), ripetendo più volte le stesse attività, motivate adeguatamente.

Entro le ore 17:00 di venerdì 28 ottobre 2022, occorrerà scegliere un capitolato (tra i 7 rispetto ai 13 gruppi), sapendo quale scegliere fino a capienza discutendo con il proponente. Vanno presentate queste candidature, altrimenti si slitta al secondo lotto. Il gruppo diventa fornitore quando ottiene l'aggiudicazione; da quel momento, inizia il tempo rendicontato, inteso come ore produttive (pianificate idealmente per un certo compito, poi migliorando).
Ogni persona ha un totale ore produttive assegnate a persona → entro l'intervallo 80 (min) – 95 (max).
Non dovrebbe succedere di superare il budget di ore (magari perché si è pianificato male o perché la gente scompare); ci viene chiesto un esperimento, non un prodotto, rispetto al budget che si ha.

Ogni gruppo è composto da ruoli con un costo orario riconosciuto secondo una tabella:

Ruolo	Costo orario (€)	Responsabilità
Responsabile	30	• Elabora piani e scadenze • Approva il rilascio di prodotti parziali o finali (SW, documenti) • Coordina le attività del gruppo
Amministratore	20	• Assicura l'efficienza di procedure, strumenti e tecnologie a supporto del *way of working*
Analista	25	• Svolge le attività di analisi dei requisiti
Progettista	25	• Svolge le attività di progettazione
Programmatore	15	• Svolge le attività di codifica
Verificatore	15	• Svolge le attività di verifica

Costo minimo ammesso in candidatura →12.000 € (gruppi di dimensione 6 – 7 persone).
Il costo accettato all'aggiudicazione dell'appalto costituisce limite superiore invalicabile

Le attività vengono verificate tramite un repository, esaminata dal verificatore. Ogni componente ha assunto *tutti* i ruoli per un tempo congruo, descrivendo le singole ore per ruolo rispetto al totale (il ruolo rimane fisso per un certo tempo).

Ogni gruppo fornitore monitora attentamente il rapporto tra costi e avanzamento tramite il

Piano di Progetto, programmando in anticipo cosa verrà fatto dopo, impegnandosi a mettere sul piatto tutte le ore per ruolo con un costo min/max di ore rispettato e un costo economico previsto (se stabilito, esso non crescerà).

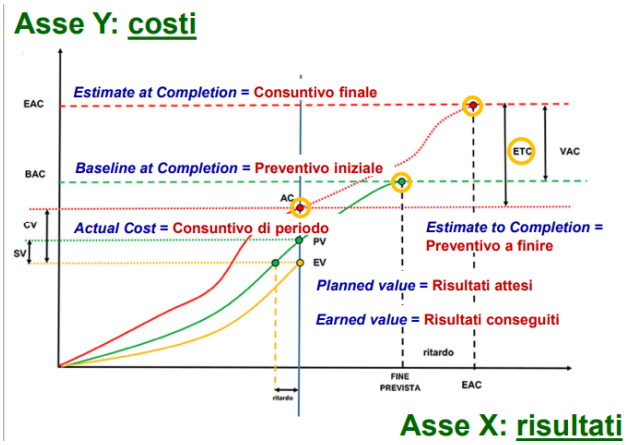
1. Calendario di massima del progetto, stima dei costi di realizzazione (preventivo iniziale)
2. Rischi attesi e loro mitigazione
3. Suddivisione del lavoro in molteplici (brevi) periodi successivi (pianificando le attività conformemente allo studio e impegni personali, rispettando il monte ore produttivo)

Fissati i compiti, progressivamente si fa il punto della situazione.

Per ogni periodo:

- Obiettivi attesi, obiettivi raggiunti
- Rischi occorsi, conseguenze, mitigazione (se succede una certa cosa, si cerca di risolvere)
- Costi osservati (consuntivo di periodo – calcolo accurato di quanto speso fino a quell'istante. Per raggiungere gli obiettivi finali rimane un certo budget, organizzandosi sulla base di quello)
- Aggiornamento calendario futuro e aggiornamento stima dei costi finali (preventivo a finire – sulla base del calcolo, si capisce quando finirà)

Come si ragiona sui costi (controllo di avanzamento):

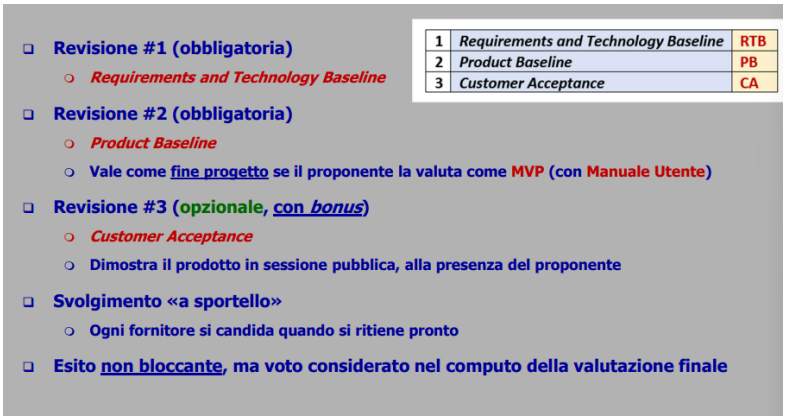


Curva di attesa e curva di misurazione, oltre agli assi, permettono di comprendere quanto si è conformi alle attese rispetto ai risultati.

Altro documento esposto è il Piano di Qualifica, dunque una specifica degli obiettivi quantitativi di qualità di prodotto e di processo. Esso è una misurazione del raggiungimento di tali obiettivi allo stato corrente, tale che sia un cruscotto di valutazione della qualità (con indicatori di prestazione rispetto alle attese, tale che le cose siano misurate in automatico in un repository). Fornisce retrospettive e iniziative di auto-miglioramento, in forma di TODO, con scadenze e misurazioni di completamento.
L'attività di qualifica unisce verifica e validazione.
Calendario di progetto:

- **14 ottobre 2022: formazione gruppi**
- **28 ottobre 2022: presentazione candidature**
- **3 novembre 2022: aggiudicazione degli appalti**
  - **A cura del committente, sentito il corrispondente proponente**
- **29 settembre 2023: tempo limite di completamento**
- **Per completare il progetto aggiudicato, ogni gruppo deve superare due (min) o tre (max) revisioni di avanzamento**
- **Sequenza analoga per i gruppi del II lotto**

Come sono fatte le revisioni:



Le revisioni vengono svolte quando *noi siamo pronti*, nello specifico:

1) Requirements and Technology Baseline/RTB
   1. Fissa i requisiti che il fornitore si impegna a soddisfare, in accordo con il proponente
      a. Documento Analisi dei Requisiti
   2. Motiva le tecnologie, i framework, le librerie selezionate per la realizzazione del prodotto
   3. Ne dimostra adeguatezza e fattibilità
      a. Tramite un Proof of Concept (demo eseguibile) coerente con gli obiettivi del capitolato (concept: idea di massima vicina alle aspettative, manipolando tecnologie spesso non conosciute)
      b. Baseline (per concetto o per codice) per avanzamenti futuri
      c. Posto in un repo accessibile ai committenti.
   4. Sottopone il PoC (Proof of Concept) alla valutazione del docente Cardin (bloccante)
      a. Prenotandosi entro specifiche finestre di opportunità
   5. Chiude la revisione con una presentazione al committente (Tullio)
      a. Solo dopo «semaforo verde» al passo 4

2) Product Baseline/PB
1. Vaglia la maturità della baseline architetturale del prodotto software e sua realizzazione
      a. Design definitivo, coerente con PoC, ma migliorativo rispetto a essa
      b. Avanzamento sostanziale di codifica (prodotto dimostrabile)
      c. Presentando Specifica Architetturale [*seguirà definizione precisa delle attese*]
2. Passa al vaglio del docente Cardin (bloccante)
   a. Prenotandosi entro specifiche finestre di opportunità
3. Chiude la revisione con una presentazione al committente
   6. Esame del grado di raggiungimento obiettivi (Tullio)
   a. Solo dopo «semaforo verde» al passo 2

3) Customer Acceptance/CA (opzionale con bonus)

L'obiettivo è creare un Minimum Viable Product (MVP), dunque una versione preliminare del prodotto atteso dotata di funzionalità sufficienti per:
- Valutare la bontà della visione iniziale
- Consentire adeguato uso esplorativo
- Permettere futuri avanzamenti

In merito allo scadenziario di progetto, si pongono delle *milestone* (tempo/data di calendario posta tra inizio progetto e tempo limite) entro le quali raggiungere certi risultati (punti obiettivo nel tempo). Esse ci aiutano a capire gli «stati di prodotto» attesi al superamento delle revisioni di avanzamento, per cui gli stati di prodotto attesi in esse sono *baseline*: risultati che sostanziano il raggiungimento di una data milestone.

Scritto da Gabriel

Tutto ciò viene misurato con le *scorecards*, schede di valutazione bilanciata di quanto fatto.

Attività	*Milestone*		
	@ RTB	@ PB	@ CA
Analisi dei requisiti	*Acceptable* (***requirements baseline***)	*Addressed*	*Fulfilled*
***Design* e codifica**	*Architecture selected* (***technology baseline***)	*Demonstrable* (***product baseline***)	*Usable* ➲ *Ready*

Esistono tre tipi di obblighi:

1) Obblighi documentali

Documenti gestionali esterni [diario di bordo]
- Piano di Progetto, Piano di Qualifica
- Costantemente aggiornati, esposti al committente secondo calendario

Documenti gestionali interni
- Norme di Progetto
- **Verbali**: breve rendiconto di riunioni ufficiali (agenda, decisioni, azioni tracciabili)
- **Glossario**: spiega ogni termine specializzato del dominio del progetto

Documenti tecnici esterni
- Analisi dei Requisiti, Specifica Architetturale, Manuale Utente

Posti nel *repo* del fornitore e tenuti sotto controllo di versione
- Ogni contenuto del *repo* ha suo proprio **Registro delle Modifiche**

2) Obblighi procedurali

Relazionare agli eventi «diario di bordo» secondo il calendario

Candidarsi alle revisioni di avanzamento
- **Quando il progresso effettivo soddisfa le attese della corrispondente *milestone***
- **Esponendo puntatore a sezione *repo* dedicata alla revisione**
  - Includendovi **Lettera di Presentazione** che elenca i contenuti della consegna e descrive brevemente lo stato di avanzamento
  - Senza invio di allegati

3) Obblighi operativi

Ogni gruppo deve dotarsi di
- **Nome, logo, riflettore di posta elettronica, *repo* (SW, documentale)**

Ogni componente di gruppo fornitore deve
- **Ruotare su tutti i ruoli secondo regole documentate**
- **Assumere ogni ruolo per un tempo totale congruo**
- **Assumere non più di un ruolo alla volta**

Le regole di rotazione devono assicurare assenza di conflitto di interessi
- **Garantendo verifica indipendente per ogni attività**

Metodi agili e Gestione di progetto (Vardanega)

Tutti i modelli esposti fino ad ora prevedono delle fasi separate, per fare in modo il flusso di lavoro avanzi verso una certa direzione; si tende a seguire dei requisiti fissi (*gates*), tale che *a cascata* si realizzi tutto con ordine, arrivando sempre alla realizzazione in una sola passata (cosa non realistica)
Si può "tornare indietro" con

- Iterazione (a piccoli passi)
- Incremento (producendo risultati ad ogni passo)

La creazione in questi modi porta ad un software che cerca di migliorare con il tempo, ma che si concentra più sulle singole fasi, non mantenendo una coerenza di fondo di *integrazione continua*.

I modelli agili si oppongono ai modelli fissi/a cascata e uniscono i vantaggi dei modelli iterativi/incrementali ma si basano molto di più sulla comunicazione/collaborazione. Si basano su un manifesto (in senso di rivendicazione) noto come *Agile Manifesto*, con lo scopo di migliorare incrementalmente il prodotto/la soluzione ottenuta con il tempo, non agendo secondo un piano ma adattandosi alle esigenze del cliente e del contesto, essendo vicini all'utente.
Il loro scopo è suddividere il lavoro a piccoli incrementi a valore aggiunto, realizzati in sequenza continua dall'analisi all'integrazione, dimostrando trasparentemente al cliente quanto fatto e tracciando l'avanzamento secondo un progresso reale e concreto.

Secondo l'Agile Manifesto, vi sono quattro principi di base:

- *Individuals and interactions over processes and tools*
  - (parliamo direttamente con il customer, è la cosa più utile)
  - L'eccessiva rigidità ostacola l'emergere del valore
- *Working software over comprehensive documentation*
  - Piuttosto che avere documenti parlanti, si vuole software demo (di fatto parlante)
  - La documentazione non sempre corrisponde a SW funzionante
- *Customer collaboration over contract negotiation*
  - Discussione collaborativa con il cliente (*chiavi in mano*; i needs espressi sono pochi, il prodotto è di per sé parlante; volendo avere altre funzionalità, si deve pagare)
  - L'interazione con gli stakeholder va incentivata e non ingessata
- *Responding to change over following a plan*
  - La capacità di adattamento al cambiare delle situazioni è importante

Il software, di per sé, non è comprensibile, pertanto occorre un mezzo definito per poter comunicare lo scopo del prodotto e della sua implementazione agli stakeholders, in modo tale da aiutare la manutenzione e la formazione del prodotto (non basta commentare codice, naturalmente)
Senza un piano, non si possono valutare rischi e avanzamenti; si cerca di fare un'analisi a lungo termine, cambiando con consapevolezza del rapporto costo/benefici.

La traduzione dei requisiti crea funzionalità significative che l'utente richiede e, come tale, si cerca di determinare le funzionalità significative che questo vuole rispetto al software richiesto, in termini di scenario d'uso. Come visto dagli UML, si parla di *user story*; ognuna è definita da:

- Un documento di descrizione del problema individuato
- Il verbale delle conversazioni con stakeholder effettuate per discutere e comprendere il problema
- La strategia da usare per confermare che il SW realizzato soddisfi gli obiettivi di quel problema

Essi hanno due principali applicazioni:

- Scrum (termine usato per indicare la "mischia" nel rugby, che sottolinea l'importanza del lavoro del team in senso collaborativo), framework basato sull'apprendimento continuo e un adattamento a fattori variabili, basando il proprio avanzamento su piccoli incrementi continuativi.

Il framework Scrum è composto da tre ruoli principali:

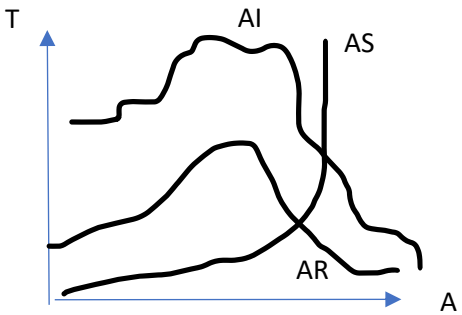
- *product owner*, responsabile di definire e dare priorità alle caratteristiche del prodotto. Essi definiscono il *backlog*, cioè l'insieme di elementi che devono essere completati per definire gli scopi del progetto, definendo cosa fare e quali priorità dare;
- *Scrum master*, che ha il compito di garantire il rispetto del framework Scrum e di aiutare il team a rimuovere gli ostacoli al progresso;
- *development team*, che è responsabile della consegna di incrementi funzionanti del prodotto alla fine di ogni iterazione, chiamata Sprint. Questo seleziona un insieme di elementi dal backlog e ci lavora.

Nell'ambito del framework Scrum, si ragiona a periodi chiamati sprint, di solito a tempo determinato e della durata variabile da una a quattro settimane, durante il quale viene completato un insieme specifico di lavoro reso pronto per la revisione. Gli sprint sono una parte fondamentale del processo Scrum, in quanto forniscono al team di sviluppo un modo per pianificare, monitorare i progressi e fornire incrementi di prodotto funzionanti.
Esistono due tipi di backlog: di prodotto (mantenuto dal product owner) e di sprint (periodo appena terminato). Ogni prodotto fisico/digitale di lavoro ottenuto è un *artefatto*, ottenuto a termine degli *Increment*, quindi lavoro completato a fine sprint.

Il framework Scrum comprende anche diversi eventi chiave, o cerimonie, che vengono utilizzati per pianificare, rivedere e porre in retrospettiva il lavoro del team di sviluppo:

- *Sprint Planning*: All'inizio di ogni Sprint, il team pianifica il lavoro che sarà completato durante lo Sprint.
- *Daily Scrum*: Il team tiene una riunione giornaliera di stand-up per discutere i progressi e gli eventuali problemi emersi.
- *Sprint Review*: Alla fine di ogni Sprint, il team rivede il lavoro completato durante lo Sprint con il Product Owner.
- *Sprint Retrospective*: Il team riflette sullo Sprint precedente e identifica i modi per migliorare in futuro.

Il modello Scrum è via di mezzo tra incrementale e iterativo (quello incrementale è il culmine del modello Agile, perché significa che sto convergendo). Il grafico seguente ne dà un'idea:



AS = Azioni Svolte/Superate
AI = Azioni In Corso
AR = Azioni in Ritardo

Legenda:

- A/Azioni
- T/Tempo

Il modo di lavorare risulta quindi *quantificabile*.
Qui interviene SEMAT (*S*oftware *E*ngineering *M*ethod *A*nd *T*heory), iniziativa nata per rimodellare il software engineering come una disciplina rigorosa. Non solo si occupa di definire e distinguere gli elementi di progetto, ma anche di indicare per essi dei "termometri" che indicano il progresso del progetto.
Queste sono le *scorecard* (metodo di valutazione sul livello di maturità in termini di aderenza) che stabiliscono sulla base di una serie di domande un punteggio di soddisfazione dei singoli punti.

Altri, citati a scopo di completezza:

- *Kanban*, che è un metodo per gestire e visualizzare il lavoro di un team. Si basa sui principi della produzione just-in-time ed è progettato per aiutare i team a ottimizzare il flusso di lavoro e a ridurre gli sprechi.
    - In Kanban, il lavoro è rappresentato da schede su un tabellone, diviso in colonne che rappresentano le diverse fasi del processo di lavoro. Man mano che il lavoro viene completato, le carte vengono spostate da una colonna all'altra e il team utilizza metriche come il tempo di ciclo e il lead time per monitorare il flusso di lavoro e identificare le aree di miglioramento.

- *Scrumban*, che è un approccio ibrido che combina elementi di Scrum e Kanban. Viene spesso utilizzato quando un team utilizza già Scrum ma vuole incorporare alcuni dei principi di Kanban per migliorare il flusso di lavoro.
    - In Scrumban, il team mantiene la struttura principale di Scrum, compresi gli sprint, i ruoli e le cerimonie, ma utilizza anche i principi di Kanban per ottimizzare il flusso di lavoro. Ciò può comportare l'uso di metriche Kanban per tracciare i progressi, l'uso di una lavagna Kanban per visualizzare il lavoro in corso o l'uso di una pianificazione basata sul pull piuttosto che sul push.

Diagrammi dei Casi d'Uso (Cardin)

(Riferimento esempio visto in classe: https://github.com/rcardin/swe-imdb)

Nel software, normalmente, si parte creando dei requisiti con lo scopo di soddisfare le esigenze del cliente le tecniche usate per individuare i requisiti funzionali (caratteristiche di prodotto) sono i *casi d'uso/use cases,* che descrivono *interazioni* tra il sistema e gli *attori* (qualsiasi entità esterna interagisca con il mio sistema non modificabile, descrivendo un ruolo).

I casi d'uso sono insiemi di *scenari*, cioè sequenze di passi che descrivono interazioni tra gli attori ed il sistema rappresentanti delle possibilità; ognuno di questi (principale o alternativi) condividono uno scopo. Descrivono l'insieme di funzionalità del sistema come sono percepite dagli utenti (senza nessun dettaglio implementativo; per questo, si deve pensare ad un caso d'uso come ad un'interfaccia [descrivendo *cosa* viene fatto e non *come*]).
Gli attori sono i *ruoli* degli utenti nell'interazione con il sistema; si consideri che non si parla solo di una persona ma anche di altri sistemi esterni (non l'applicazione stessa, attenzione). Essi svolgono il caso d'uso per raggiungere l'obiettivo. Occorre individuare la lista degli attori e comprendere i loro obiettivi per capire come interagire con il sistema (ruoli e funzionalità) per capire come trovare i casi d'uso.

Gli use case sono puro testo (UML descrive solo gli *use case diagrams/diagrammi dei casi d'uso*), in cui il valore aggiunto è nel contenuto testuale (nome-identificatore/scenari e loro tipi/precondizioni/effetti/attori principali-secondari). Normalmente, vengono anche definite *user story*, in quanto si cerca di catturare dalla prospettiva di un utente finale una caratteristica del prodotto.

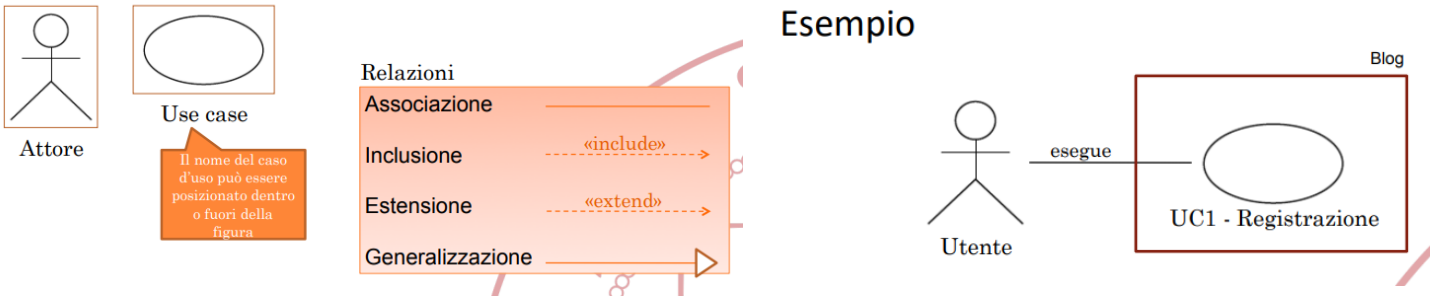
Caso d'uso: UC1 - Registrazione
Attore primario: Utente
Precondizioni: L'utente non è ancora autenticato presso il sistema
Postcondizioni: L'utente possiede un'account presso il sistema, contraddistinto da una username e da una password
Scenario principale:
1. L'utente accede al sistema
2. L'utente seleziona la funzionalità "Registrati"
3. L'utente inserisce una username univoca nel sistema
4. L'utente inserisce una password che rispetta i vincoli imposti

Estensioni:
a. Nel caso in cui l'utente inserisca una username già censita a sistema:
    1. L'utente non viene registrato presso il sistema
    2. Viene visualizzato un errore esplicativo
    3. Viene fornita all'utente la possibilità di scegliere un'altra password

Normalmente, si ha:

- Un solo scenario principale per caso d'uso
- Scenari alternativi (0..*)
    - Prendono in considerazione solo la parte che differisce dallo scenario principale
- Si cerca la *granularità*, quindi soddisfare lo scopo di un attore
    - Più piccolo di un processo di business
        - Non fornisce dettagli significativi, ma individua le funzionalità del sistema
    - Più grande di una singola operazione su un componente
        - Dettaglio eccessivo allontana il focus dall'obiettivo

I principali componenti di un diagramma sono attore, use case e relazioni, come segue:



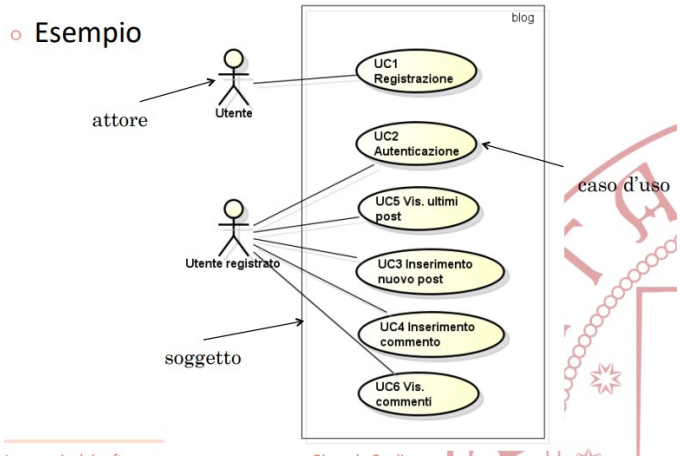
L'associazione attore - use case è una <u>partecipazione</u>, intesa come comunicazione diretta (utilizzo del sistema); normalmente, questa deve essere descritta anche in versione testuale, dato che pre/post condizioni non possono essere desunte.
Qui possiamo individuare:

- Attore primario: L'attore primario è il principale utente o stakeholder che avvia il caso d'uso e ha un obiettivo da raggiungere utilizzando il sistema.

- Attore di supporto: Gli attori di supporto sono utenti o stakeholder secondari che sono coinvolti nel caso d'uso, ma che non iniziano l'interazione con il sistema. Gli attori di supporto possono fornire ulteriori input o risorse necessarie per completare il caso d'uso.

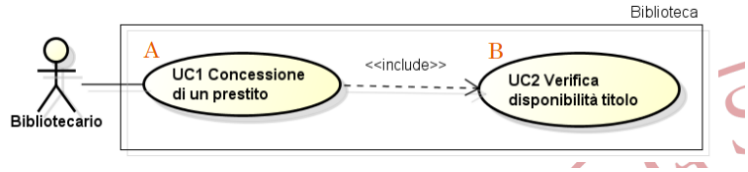
- Sistema: Il sistema è l'oggetto del caso d'uso e rappresenta la funzionalità o il comportamento del sistema descritto.
- Ambiente: L'ambiente comprende tutti i fattori o i vincoli esterni che possono avere un impatto sul caso d'uso, come sistemi esterni o fonti di dati esterne.

Un esempio è come segue:



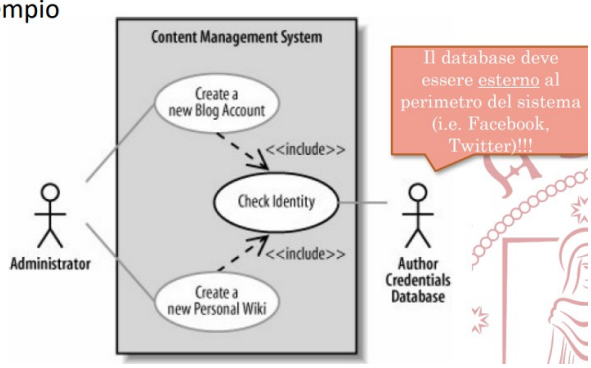
Quando si ha a che fare con una funzionalità/comportamento comune fra più *use case*, si parla di inclusione. In questo caso, ogni istanza di A esegue B, sapendo che la responsabilità di esecuzione è solo di A che non conosce i dettagli di B ma solo i suoi risultati, mentre B non conosce di essere incluso da A ed è incondizionatamente incluso nella sua esecuzione.

Questa serve ad evitare la ripetizione ed aumentare il *riutilizzo* e anche la modularità (mettendo più funzionalità insieme, aumentando la chiarezza) e la mantenibilità (a prescindere dall'aumento di funzionalità).

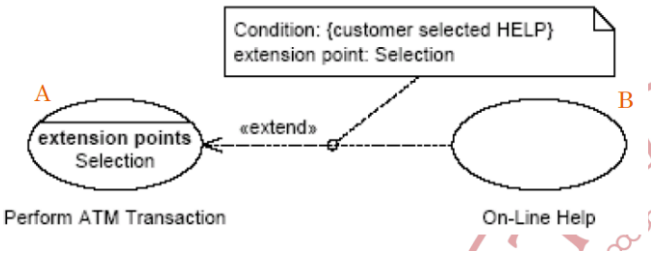


Caso di applicazione segue:

- Esempio



Esiste inoltre un'estensione, quindi *aumento* delle funzionalità di uno use case.
Ogni istanza di A esegue B in modo *condizionato*, l'esecuzione di B interrompe A e la *responsabilità* dei casi di estensione è *di chi estende* B. Si ricordi che questa *non* rappresenta l'ereditarietà nei linguaggi di programmazione.



Un'estensione in un caso d'uso è una variazione del flusso principale di eventi del caso d'uso che rappresenta uno scenario alternativo o eccezionale. Le estensioni sono utilizzate per catturare comportamenti o interazioni aggiuntive che possono essere richieste in determinate circostanze, ma che non si applicano a tutte le istanze del caso d'uso.

Ad esempio, si consideri un caso d'uso per il check-out di un carrello della spesa in un negozio online. Il flusso principale di eventi potrebbe includere l'aggiunta di articoli al carrello, la revisione degli articoli nel carrello e il completamento del processo di checkout. Un'estensione di questo caso d'uso potrebbe essere un flusso "applica coupon", che consentirebbe all'utente di inserire un codice coupon e applicare uno sconto all'ordine. Altro esempio molto comune; *gestione delle eccezioni*.

L'uso di estensioni nei casi d'uso comporta diversi rischi:

- Complessità: L'aggiunta di estensioni a un caso d'uso può aumentarne la complessità, poiché aggiunge ulteriori scenari e condizioni da considerare.
- Test: Il test di un caso d'uso esteso può essere più complesso, in quanto può richiedere la verifica di più scenari e condizioni.
- Manutenzione: Se il caso d'uso viene modificato o aggiornato, può essere necessario aggiornare anche le estensioni per garantire che siano ancora pertinenti e accurate.

Scritto da Gabriel

- Confusione: I flussi delle estensioni possono non essere immediatamente evidenti agli utenti o alle parti interessate, il che può generare confusione o fraintendimenti sul comportamento del sistema.

È importante considerare attentamente i rischi associati all'uso delle estensioni nei casi d'uso e usarle con parsimonia quando sono necessarie per catturare comportamenti o interazioni importanti.

L'inclusione e l'estensione hanno delle caratteristiche in comune, poiché entrambi:
- *fattorizzano* comportamenti comuni a più use case
- *aumentano* il comportamento di uno use case base
- *modularizzano* gli use case chiarendoli

Tuttavia, hanno delle differenze:
- Le inclusioni incorporano il comportamento o le interazioni di un altro caso d'uso nel caso d'uso corrente, mentre le estensioni rappresentano uno scenario alternativo o eccezionale all'interno del caso d'uso corrente.

- Le inclusioni sono indicate con una linea e una punta di freccia che punta al caso d'uso incluso, mentre le estensioni sono tipicamente rappresentate con una linea tratteggiata e una punta di freccia "extend".

- Le inclusioni possono essere utilizzate per riutilizzare i casi d'uso esistenti ed evitare la duplicazione degli sforzi, mentre le estensioni sono utilizzate per catturare comportamenti o interazioni supplementari o eccezionali che potrebbero non applicarsi a tutte le istanze del caso d'uso.

- Le inclusioni possono essere più semplici da comprendere e utilizzare, in quanto incorporano semplicemente il comportamento o le interazioni di un altro caso d'uso nel caso d'uso corrente. Le estensioni possono essere più complesse, in quanto richiedono la considerazione di condizioni e scenari aggiuntivi.

- Nelle estensioni, l'attore può non eseguirle tutte (condizioni non verificate), mentre nelle inclusioni, l'attore le esegue sempre tutte

L'estensione si usa quando un caso d'uso aggiunge passi a un altro caso d'uso di prima classe. Questa si occupa di aggiungere (non riutilizzare) funzionalità e anche qualsiasi funzionalità opzionale.

Ad esempio, immaginiamo che "Prelevare contanti" sia un caso d'uso di uno sportello automatico (ATM). " Tassa di valutazione " estenderebbe "Prelievo di contanti " e descriverebbe il "punto di estensione" condizionale che viene istanziato quando l'utente del bancomat non effettua operazioni bancarie presso l'istituto proprietario del bancomat. Si noti che il caso d'uso di base "Prelievo di contanti " sta in piedi da solo, senza l'estensione.

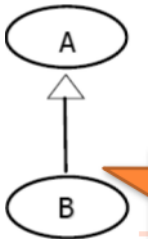
L'inclusione è usato per estrarre frammenti di casi d'uso che sono duplicati in più casi d'uso. Il caso d'uso incluso non può stare in piedi da solo e il *caso d'uso originale non è completo senza quello incluso*. Questa opzione deve essere usata con parsimonia e solo nei casi in cui la duplicazione è significativa ed esiste per progetto (piuttosto che per coincidenza). L'inclusione riutilizza funzionalità.

Ad esempio, il flusso di eventi che si verifica all'inizio di ogni caso d'uso del bancomat (quando l'utente inserisce la carta bancomat, inserisce il PIN e gli viene mostrato il menu principale) sarebbe un buon candidato per un'inclusione.

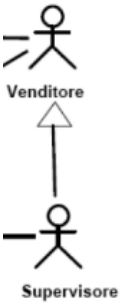
In generale:
- Inclusione: una funzionalità si ripete in più use case
- Estensione: si vogliono descrivere variazioni dalla funzionalità standard

Scritto da Gabriel

Anche negli use case si ha una <u>generalizzazione</u>, che permette di aggiungere o modificare caratteristiche base (imm. sx. primo caso, imm. dx. secondo caso):

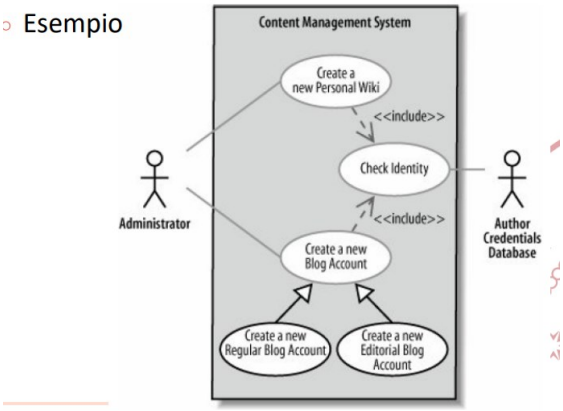


- Nel caso degli *attori*, A è generalizzazione di B se B condivide almeno le funzionalità di A.
    - normalmente si identifica questa cosa rispetto ad utente admin, generalizzazione di utente normale
- nel caso degli *use case* (più raro), i casi d'uso figli possono aggiungere funzionalità rispetto ai padri, o modificarne il comportamento
    - tutte le funzionalità non ridefinite nel figlio si mantengono in questo come definite nel padre



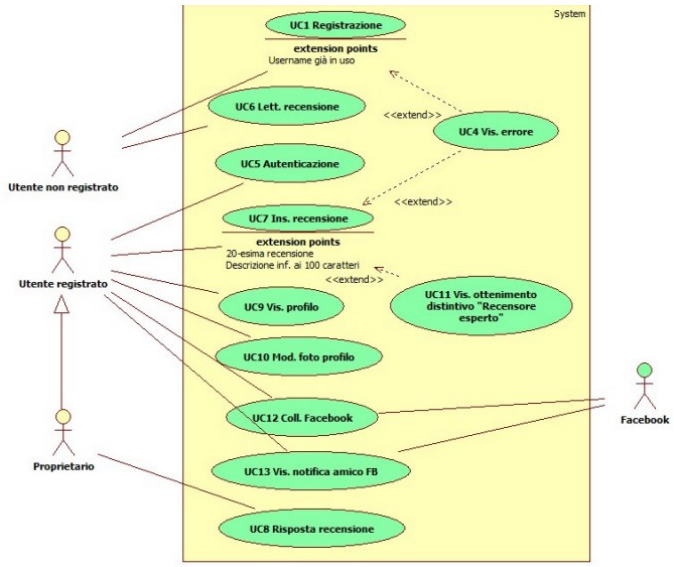
Esempio di generalizzazione sui casi d'uso (tutti i sottocasi fanno parte del caso principale; esempio → Visualizzazione Profilo e sottocasi Visualizzazione Nome, Visualizzazione Stato, etc.)

- Esempio



Esempio utile di caso d'uso con descrizione e individuazione attori; si noti la presenza di Facebook come attore esterno, in quanto coinvolto nel processo di collegamento account. Per il resto si intende che quasi ogni funzionalità sia un caso d'uso. L'errore estende situazioni, mentre l'inclusione va usata solo in casi limitati

Tripadvisor è un noto sito di viaggi diffuso in tutto il mondo. Per accedervi, è necessario registrarsi fornendo una username e una password. Come in molti altri sistemi, la usename deve essere univoca: il sistema, quindi, non permette ad un nuovo utente di registrarsi utilizzando una username già scelta da un altro utente. All'interno del sito sono presenti le recensioni di numerose attrazioni turistiche, ristoranti, hotel, ecc...Le recensioni sono visibili pubblicamente e possono essere lette anche dagli utenti non registrati. La scrittura delle recensioni è disponibile unicamente per gli utenti registrati. Ogni recensione contiene un giudizio riassuntivo che l'utente inserisce utilizzando le "stelle" (da una a cinque) e da una descrizione di almeno 100 caratteri. Nel caso si cerchi di inserire una recensione di lunghezza inferiore, il sistema avvisa l'utente con un messaggio di errore. È possibile per l'eventuale proprietario dell'attrazione turistica rispondere brevemente ad una recensione, inserendo a sua volta un commento. Il profilo di un utente è caratterizzato oltre che dal suo nome e dalla sua foto, che può essere modificata, dai distintivi che ha ottenuto. I distintivi sono legati al numero di recensioni scritte: ad esempio, dopo 20 recensioni l'utente diviene un "Recensore esperto" e il sistema lo notifica con un messaggio opportuno. È infine possibile collegare il proprio account con il proprio profilo Facebook. In questo caso il sistema notificherà l'utente ogni qualvolta un proprio amico inserisce all'interno di Tripadvisor una recensione.



Il *processo di individuazione degli use case* definisce il contesto sulla base di una serie di caratteristiche:

1. Identificazione attori e responsabilità
2. Identificazione degli obiettivi da raggiungere per ciascun attore
    - Prima approssimazione use case
3. Valutare attori e use case e raffinarli
    - Divisione e accorpamento
4. Trovare le relazioni di inclusione
5. Trovare le relazioni di estensione
6. Trovare le relazioni di generalizzazione

Diario di bordo – Inizio Progetto

Domande emerse:

- Documenti candidatura/Stima dei costi-ore
    - La candidatura si svolge tramite mail, senza allegati, ma riferendo nella stessa candidatura dove si trovano i documenti (sulla repo, secondo una o più cartelle a nostra scelta). In questa si trova una lettera di presentazione (la candidatura è composta dei documenti X, Y, Z, ecc.). Ogni singola cosa ha una versione documentata. Ci si candida per una sola scelta, motivando in sintesi pro e contro della decisione da noi fatta (con un riassunto, preferendo il capitolato X per i motivi Y, dando giudizi sui vari capitolati).

    - Evidenza degli incontri con i proponenti tramite i verbali che mostrano le discussioni tra di noi, inserendo anche le discussioni con i proponenti. Non sono loro a scegliere noi, ma Tullio a scegliere tutto.

    - La dichiarazione impegno nella candidatura viene fatta per dedicare ore al progetto (gruppo di persone mette sul tavolo X ore, facendo promesse serie, considerando gli impegni di tutti). Tutti portano il loro calendario, affinché ciascuna persona garantisca X fino ad una certa data.

- Piano di progetto composto da calendario di massima, stima dei costi possibili, rischi attesi e mitigazione e suddivisione del lavoro. Nel preventivo dei costi garantisce X ore * 7 (componenti del gruppo).
  - Responsabile
  - Strumenti automatizzabili
  - Quantità disponibile

- In merito ai costi, cerchiamo di pianificare in modo reale e non ottimistico (elencando eventuali difficoltà/rischi; es. tecnologie che si pensavano più facili, il capitolato non piace più). Essendo che c'è poco tempo, l'analisi dei rischi considera questa cosa (anche eventuale disponibilità oraria delle persone, se uno non riuscisse si ripartisce su altri più disponibili, ma si tiene conto del debito).

- (Verbalizzazione) → Racconto sintetico con X partecipanti, descrivendo ciò che è stato prodotto a livello di decisioni in merito all'agenda, mostrando in modo coerente la gestione attraverso la (Piattaforma attività), gestendo chi sta facendo cosa, entro quando deve essere fatto e perché viene fatto.

- (Rotazione dei ruoli) → Difficilmente nel mondo reale cambiano ma, nel progetto, si è obbligati a cambiare i ruoli. L'uso di una certa persona nel ruolo è coerente con l'organizzazione, sapendo che ogni ruolo viene svolto per X ore produttive (rotazione settimanale/bisettimanale/mensile, ecc., aggiustando in corso d'opera), non facendo più ruoli insieme.

Gestione di progetto (Vardanega)

(Eventuali approfondimenti e link utili dati dal prof:
https://www.math.unipd.it/~tullio/IS-1/2006/Dispense/L05b.pdf
https://www.math.unipd.it/~tullio/IS-1/2008/Approfondimenti/IEEE_Software_25-5_Bohem.pdf
https://www.slideshare.net/corvinno/pert-presentation-824238
https://www.agileuprising.com/2017/01/01/10-retrospectives/
https://www.math.unipd.it/~tullio/IS-1/2004/Approfondimenti/CHAOS.html
https://www.gantt.com/
www.researchgate.net/publication/265986910_LA_STIMA_DEI_COSTI_DEI_SISTEMI_INFORMATIVI_AUTOMATIZZATI
http://www.slideshare.net/Samuel90/project-management-technology-report)

Ogni attività è pianificata *prima* di essere svolta, secondo specifici obiettivi e vincoli e ricercando economicità. Si usa un modello *adotta-ed-adatta*:

- Adotta → Non "reinvento la ruota", ma utilizzo processi/tecnologie esistenti in modo utile
- Adatta → Le attività vanno prese e rimodificate in base ai vincoli del progetto

I fondamenti della gestione di progetto sono basati sul way of working; si ricordi infatti che *non c'è project senza way of working*. Seguono i principi:

- Stabilire il proprio il way of working (esso non rimane fisso, ma viene costruito incrementalmente)
  - Adattando processi di ciclo di vita e istanziandoli in attività di progetto
- Avere una pianificazione per capire le cose da fare e come collocarle nel tempo
- Determinare le risorse disponibili (ore-persona e calendario)
- Fissare gli obiettivi di avanzamento
  - In una successione di milestone (punti d'arrivo di determinati obiettivi), da quella finale (la prima che viene fissata, cioè la data di fine) all'indietro (che determina il percorso compiuto fino a quel momento). Cerchiamo il cammino minimo per arrivare a destinazione (cerco economicità), riconoscendo milestone utili per capire lo stato di avanzamento
  - Orientamento delle attività al raggiungimento di quegli obiettivi, cercando di rispettare le scadenze e calcolando l'impegno temporale per persona

- Determinare le risorse necessarie per svolgere quelle attività
    - Questo si chiama «*preventivo*»
- Adattare gli obiettivi alle disponibilità effettive, bilanciando il progresso in base agli obiettivi dichiarati e capacità reali (questi vengono rimodificati anche dagli stakeholder).
    - Non si tengono mai immutati, dato che i requisiti rimangono elastici e negoziabili in funzione dell'avanzamento.

Per controllare l'avanzamento regolarmente e capire se si sono raggiunti degli obiettivi secondo le milestone, si adotta una baseline, che visualizza i risultati raggiunti e i passi successivi da definire (notifica del completamento delle azioni). Essa si basa su un calendario e associa le milestone da raggiungere nel tempo, ragionando a *periodi*. I costi sostenuti all'interno di un singolo periodo è il *consuntivo di periodo*.
La tecnica agile usata è la *retrospettiva*, attività di valutazione/revisione effettuata alla fine del periodo che riflette su cosa è stato fatto (raccogliendo dati, identificando trend, generando e prioritizzando idee, etc.)

Occorre darsi una disciplina allo stato dell'arte per capire come dividersi i compiti, bilanciando il ruolo per specializzazione in funzioni, normalmente incarichi permanenti e delegati a singole persone.
Anche nel Progetto si assumono *ruoli*, per cui una persona ha delle *ownership/oneri,* nel senso di responsabilità sui compiti da svolgere. Nei progetti reali rimangono fissi, nel Progetto Didattico ruotano per ragioni formative.

Suddividiamo i ruoli:

- Gli *analisti*, cioè coloro che sono responsabili della raccolta e della documentazione dei requisiti e delle specifiche di un sistema software. Lavorano a stretto contatto con le parti interessate per comprendere le loro esigenze e tradurle in requisiti tecnici. Essi non servono fino alla fine del progetto ma ne servono tanti per garantire in modo coordinato e collaborativo lo svolgimento delle attività e il corretto risultato finale in termini di prodotto finito.

- I *progettisti/designer*, che hanno competenze tecniche e tecnologiche aggiornate e decidono *come* il software stia insieme nel suo complesso sulla base delle scelte realizzative. Il codice viene dopo il design, in quanto l'architettura è uno schema fisso come visione d'insieme e il design è come si pensa il prodotto venga effettivamente costruito. Normalmente ci sono pochi progettisti che seguono lo sviluppo, non la manutenzione.

- I *programmatori*, che servono moltissimo alla realizzazione e manutenzione del prodotto e contribuiscono collaborativamente alla realizzazione del design del prodotto finale. Se ben organizzato, essendo ruolo con deleghe limitate e compiti precisi, si può raggiungere massimo parallelismo e velocità, dato che la scrittura di codice avanza secondo regole precise e ben delineate. La scrittura del software comporta aver compreso bene il design e realizzarlo secondo uno schema preciso.

- I *verificatori*, che sono presenti per tutta la durata del progetto. Essi sono molti, dato che tutte le attività svolte sono soggette a verifica che determina *per certo* che un compito abbia terminato. La pianificazione prevede sempre la verifica, che deve essere *già nota* all'inizio (cioè, la procedura con cui sarà fatta l'analisi del risultato e che rimanga sempre quella nel tempo). Idealmente, ogni azione è sottoposta a verifica, al fine di mantenere il software ad alto livello, in cui ogni aspetto è precisamente definito e mantenuto.

- Il *responsabile/project manager*, che governa il team e rappresenta il progetto verso l'esterno (livello customer verso gli stakeholder) ed è ruolo ricoperto da una singola persona per ragioni di economicità. Collabora con il team per definire l'ambito, la tempistica e il budget del progetto e si assicura che il progetto rimanga in linea con i tempi. Gestisce inoltre i rischi e comunica i progressi alle parti interessate, gestendo le risorse con economicità in base a conoscenze/capacità tecniche

che gli consentano di farlo per certo. Egli si occupa di approvare i documenti e redigere l'organigramma e il Piano di Progetto.

- L'*amministratore di sistema/sysadmin*, che definisce, controlla, e manutiene l'ambiente IT di lavoro. Le risorse informatiche utili sono decise da lui, cercando di automatizzare lo sviluppo (in modo proattivo, cioè prevenendo i problemi possibili). Questa figura ha il ruolo di garantire che le risorse rimangano sempre funzionanti, garantendo una gestione veloce e sicura e gestisce anche i *ticket*/segnalazioni in merito all'infrastruttura progettuale (server in particolar modo). Lui è il responsabile del versionamento e della documentazione.

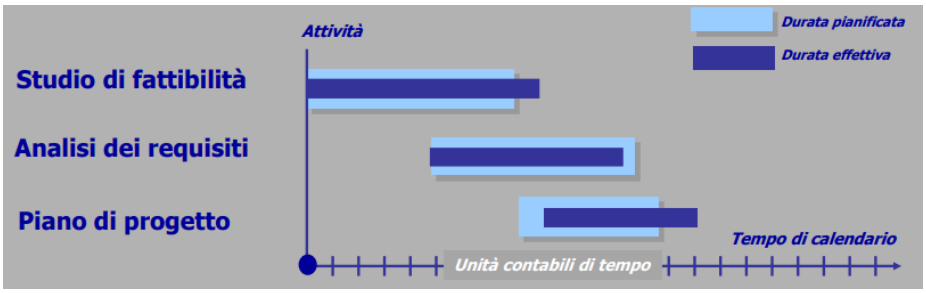
- Il *gestore della qualità* è una funzione aziendale che ha avuto introduzione più recente. Essa interessa sia il committente che la direzione aziendale e riguarda sia i prodotti che i processi. Questa richiede applicazione rigorosa dei processi adottati (producendo *confidenza*), data anche da una continua *manutenzione* migliorativa (ciclo PDCA – Plan Do Check Act): la qualità deve produrre *confidenza* ("noi lavoriamo così, sapendo che funziona").

La pianificazione di progetto deve scendere a un dettaglio idoneo a individuare attività brevi (secondo un quanto *rapido quanto le attività di pianificazione*, per esempio per giorni [non per settimane]), pianificandone lo svolgimento e valutandone il progresso, avendo una base per allocazione di risorse e stime scadenze/costi. Si deve ripartire bene: occorre avere compiti brevi e assegnabili ad un singolo incaricato. Lo stato di avanzamento di un prodotto è rilevante solo se fornisce informazioni per riflettere sulla pianificazione.

Partendo da un problema:
- si identificano le attività e le loro dipendenze
    - normalmente, questo sono a struttura gerarchica (ad albero), dislocate secondo vincoli di precedenza per non creare attese
- si stimano le risorse assegnando delle persone per tali attività
- si creano i grafici di progetto

Uno strumento di project management molto usato sono i diagrammi di Gantt, che sono un tipo di diagramma a barre che illustra la pianificazione di un progetto. Essi mostrano le attività da completare sull'asse verticale e gli intervalli di tempo sull'asse orizzontale. Ogni attività è rappresentata da una barra che inizia alla data di inizio dell'attività e termina alla data di fine dell'attività. La lunghezza e la posizione della barra riflettono la durata dell'attività e la sua relazione con le altre attività del progetto.

In particolare rappresentiamo:
- durata delle singole attività
- vediamo sequenzialità e parallelismo, anche in termini di gerarchie (vincoli di precedenza e di attesa)
- confrontiamo le stime con i progressi



In generale, un diagramma di Gantt non è particolarmente atto alla gestione delle dipendenze.

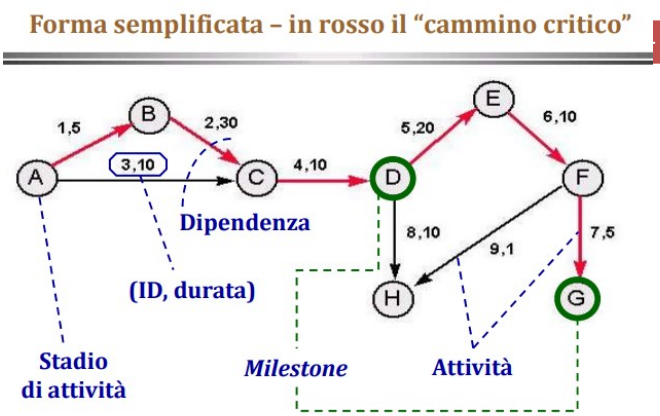
Altro strumento spesso utilizzato sono i diagrammi PERT (*Program Evaluation Review Technique*), che sono rappresentazioni visive delle attività di un progetto e delle relative dipendenze. Ciascuno è composto da nodi, che rappresentano le attività, e da frecce, che rappresentano le dipendenze tra le attività. La lunghezza della freccia rappresenta la durata dell'attività. Un'attività non può iniziare finché tutte le sue dipendenze non sono state completate.

Scritto da Gabriel

Ogni attività ha delle dipendenze temporali con le altre per ragionare all'indietro sulle scadenze di progetto, individuando il possibile margine temporale (slack time).

Esso non viene scelto a *zero-latency* (senza alcun ritardo, subito, a latenza pari a zero) né a *zero-laxity* (senza alcun margine, il più tardi possibile, a margine zero).

Individuiamo i *cammini critici,* che sono sequenze di attività ordinate, con esito importante, e dipendenze temporali tra loro molto strette. Si cerca di arrivare ad una soluzione trovando il cammino critico che impiega minore tempo/risorse.



Portano quindi a ragionare e valutare le scadenze di progetto, fornendo per ogni dipendenza un tempo di slack che rappresenta la quantità di tempo di cui un evento può essere ritardato senza influenzare però l'andamento del progetto. Molte risorse sono impegnate su più progetti, pertanto si deve operare con economicità, avendo spesso a che fare con più "cammini critici" su più progetti, quindi sequenze di attività da completare per arrivare all'obiettivo prefissato nel minore tempo possibile.

Strumenti utili in questo senso possono essere:

- WBS (Work Breakdown Structure), diagramma che mostra le attività di progetto e ne rappresenta la gerarchia tramite una struttura ad albero fino a mostrare i work affidabili a una singola persona.
    - Esso può utilizzare il CoCoMo (Constructive Cost Model), modello algoritmico impiegato per la stima dei costi in fase di pianificazione di progetto. Esso stima le risorse necessarie ed esprime i costi misurati in mesi/persona (MP).

Anche l'attività di stima dei costi va effettuata con degli strumenti che permettono di organizzare le attività ed evidenziare le criticità studiando tutti i possibili scenari. Normalmente, questa viene eseguita dal Responsabile. Questi sono misurati in tempo di calendario/persone necessarie.

Due massime in questo senso (screen da Sweky, ottima enciclopedia FIUP di SWE):

> *Il lavoro si espande per riempire tutto il tempo disponibile.*
>
> *— Legge di Parkinson*

> *Minore è il prezzo di un **servizio** o di una commodity, maggiore sarà la quantità richiesta.*
>
> *— Legge della domanda*

La stima si esegue con esperienza, tramite analogia, analizzando la competizione, adottando un algoritmo predittivo, eseguendo dei raffinamenti: a grana grossa sull'insieme, a grana fine entro periodi brevi.

I *fattori di influenza* sulle stime sono diversi:

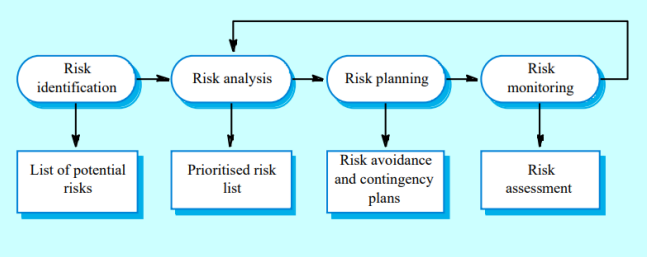
- identificazione delle attività e costi
- rischi legati alle attività
- costi legati alla gestione delle risorse
- dimensione del progetto
- esperienza del dominio
- familiarità con le tecnologie
- produttività dell'ambiente di lavoro (anche in termini di disponibilità/organizzazione)
- qualità attesa

Scritto da Gabriel

Le possibili *fonti di rischio* sono:
- Tecnologie di lavoro e di produzione SW
- Rapporti interpersonali
- Organizzazione del lavoro
- Requisiti e rapporti con gli stakeholder
- Tempi e costi

L'attività di gestione dei rischi permette di identificare i rischi potenziali, valutarne la probabilità e l'impatto potenziale sul successo di progetto; si cerca di sviluppare strategie per mitigarli o eliminarli. Chiaramente ne esistono di diversi tipi (tecnici, business, gestionali).

Essa si compone di più fasi:
- *identificazione* nel progetto/prodotto/mercato, listando i potenziali rischi;
- *analisi*, cercando di capire le probabilità di occorrenza e le possibili conseguenze e capendo a quali rischi dare priorità;
- *pianificazione*, cercando di sviluppare strategie per mitigare/eliminare i rischi;
- *controllo*, dando attenzione ai problemi che emergono e assicurandosi vengano ben gestiti.



Si evidenzia, secondo le statistiche, una scansione dei tipi di progetti:
- *di successo*, quindi progetti realizzati in tempo, senza costi aggiuntivi e che creano prodotto soddisfacente (una minoranza, sotto il 20% metà anni Novanta, circa 35% metà anni Duemila)
- *a rischio*, quindi progetti ormai fuori tempo, con costi aggiuntivi e che portano ad un prodotto difettoso (oltre il 50%)
- *fallimenti*, quindi progetti cancellati prima della fine (oltre il 30% metà anni Novanta, circa 15% metà anni Duemila)

I *fattori di successo* di un progetto sono nell'ordine dato (di importanza in senso decrescente):
- Coinvolgimento del cliente
- Supporto della direzione esecutiva
- Definizione chiara dei requisiti
- Pianificazione corretta
- Aspettative realistiche
- Personale competente

I *fattori di fallimento* di un progetto sono nell'ordine dato (di importanza in senso decrescente):
- Requisiti incompleti
- Mancato coinvolgimento del cliente
- Mancanza di risorse
- Aspettative non realistiche
- Mancanza di supporto esecutivo
- Fluttuazione dei requisiti

Software Architecture Patterns (Cardin)

(Riferimenti libri: *Eric Evans - Domain Driven Development / Chris Richardson – Microservices patterns*)
Codice presentato a lezione: https://github.com/rcardin/swe-imdb/tree/main/layered
https://github.com/rcardin/hexagonal
https://github.com/rcardin/swe-imdb/tree/main/hex)

Le applicazioni normalmente non hanno un'architettura formale e sono *accoppiate* (*coupled*) e difficili da modificare (disorganizzate e difficili da mantenere). Subito nascono vari problemi: come scala l'architettura nel corso del tempo? Come risponde l'applicazione in merito ai cambiamenti? Come viene fatto il deploy?

Per questo motivo esistono i pattern architetturali, che nel contesto del software sono considerate soluzioni generali/ad alto livello, utili per organizzare la struttura e le funzionalità di un sistema, dando dei vincoli sulle decisioni da adottare in un contesto specifico.

Un sistema non viene visto come insieme ma *in parti*; una prima idea di questa strutturazione si ha con il 4+1 architectural view model, che cerca di fornire una vista comprensiva del sistema e delle sue componenti per capire come si integrano tra loro.

- Logical View: Questa visione si concentra sui requisiti funzionali del sistema e sulle relazioni tra i vari componenti che lo costituiscono.
- Process View: Questa visione si concentra sulle interazioni tra i vari componenti del sistema e sul modo in cui lavorano insieme per realizzare i compiti.
- Development View: Questa visione si concentra sugli aspetti tecnici del sistema e su come viene implementato e distribuito.
- Physical View: Questa vista si concentra sull'infrastruttura hardware e software necessaria per supportare il sistema.
- Use Case View: Questa vista si concentra sulle interazioni tra il sistema e i suoi (scenari)

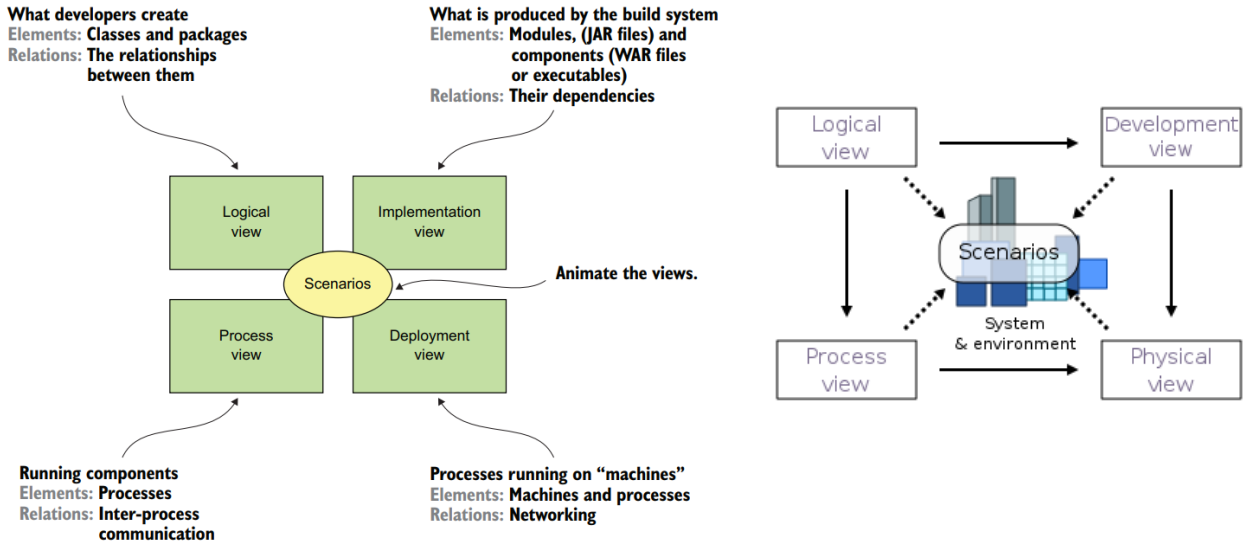


Figura 2 - C. Richardson - Microservices Patterns

Il pattern architetturale più usato è il layered architecture/architettura stratificata, in cui un sistema software è strutturato in una serie di layers/strati, ognuno dei quali fornisce un insieme specifico di servizi o funzioni. I livelli sono tipicamente organizzati in base al livello di astrazione, con i livelli inferiori che forniscono servizi di base e quelli superiori che si basano su di essi per fornire funzionalità più complesse.

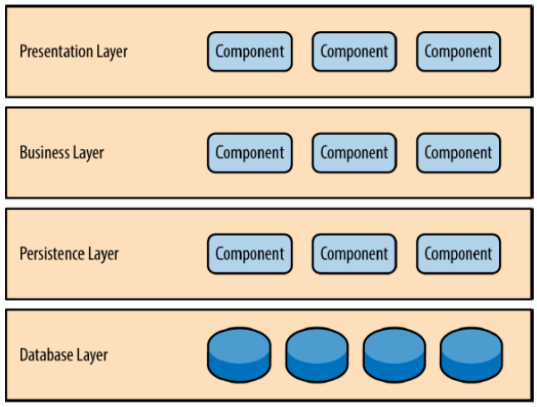
- Un esempio comune di architettura a strati è il modello OSI

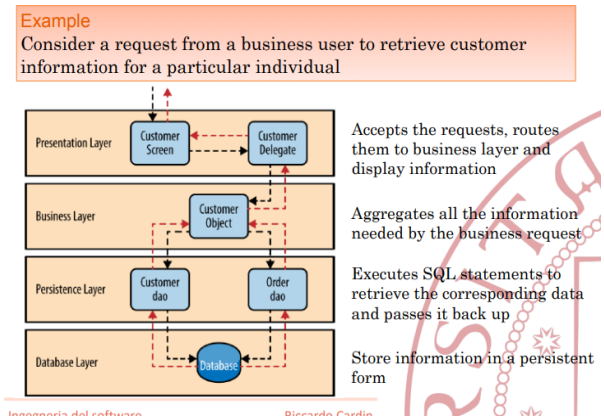
Non vi è un numero fisso di strati; normalmente, è possibile averne fino ad *N* (*N-Tier architecture*), riflettendo la struttura organizzativa ritrovata nella maggior parte dei sistemi IT.
La loro organizzazione sempre un principio noto nel software come *legge di Conway*, secondo cui "la struttura di un sistema è fortemente influenzata dai pattern di comunicazione che lo sviluppano, arrivando ad essere una copia della struttura organizzativa alla sua base".

Questo significa che:

- se un'organizzazione ha una struttura gerarchica, con chiare linee di comunicazione e autorità, anche i sistemi software che sviluppa avranno una struttura gerarchica;
- se l'organizzazione ha una struttura piatta, con una comunicazione e un processo decisionale più decentralizzati, i sistemi software che sviluppa tenderanno ad avere una struttura più decentralizzata e modulare.

I componenti di un'applicazione di questo tipo sono organizzati secondo layer *orizzontali*, in cui ognuno esegue uno specifico ruolo. La più comune implementazione è quella che segue:





1) *presentation layer*, noto anche come livello dell'interfaccia utente (UI), è responsabile della presentazione delle informazioni e della gestione degli input. In genere è costituito dagli elementi visivi del software, come l'interfaccia grafica utente (GUI), e dal codice che controlla come l'interfaccia viene visualizzata e come risponde alle azioni dell'utente.

2) *business layer*, noto anche come livello logico dell'applicazione, è responsabile dell'implementazione della logica di business del software. In genere consiste nel codice che elabora i dati ed esegue i compiti relativi alla funzionalità principale del software.

3) *persistence layer*, noto anche come livello di accesso ai dati, è responsabile della memorizzazione e del recupero dei dati da un sistema di memorizzazione persistente, come un database. In genere è costituito da codice che interagisce con il database e gestisce operazioni quali l'inserimento, l'aggiornamento e la cancellazione dei dati.

4) *database layer*, è responsabile dell'archiviazione e dell'organizzazione dei dati in modo strutturato. In genere consiste in un sistema di gestione di database (DBMS) che fornisce strumenti per creare, interrogare e aggiornare un database.

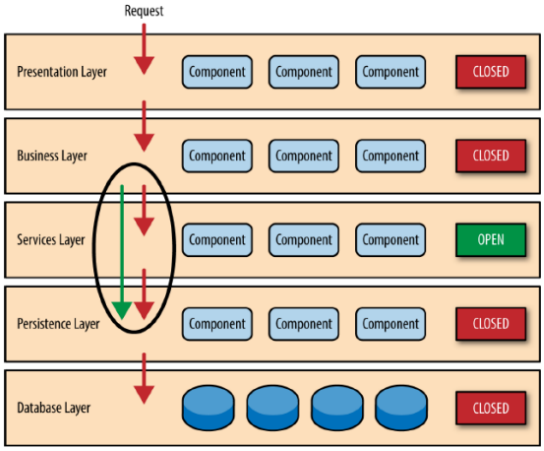
Ogni strato forma *un'astrazione* di una certa richiesta e comunica solo con componenti che stanno nello stesso strato; la classificazione rende facile capire ruoli e responsabilità (se sono limitate, spesso, si parla di interfacce). In questo modo, si realizza il separation of concerns/separazione dei ruoli (possibilmente disaccoppiando dal contesto e creando *interfacce* di componenti ben definite).

Con la decomposizione in layer il sistema viene visto come una gerarchia di sottosistemi. Un layer è un raggruppamento di sottosistemi che forniscono servizi correlati. I layer per implementare un servizio potrebbero usare servizi offerti dai layer sottostanti ma non possono usare servizi dei livelli più alti.

Con i layer si possono avere due tipi di architettura: *chiusa* e *aperta*.

- Con l'architettura chiusa, un layer può accedere solo alle funzionalità del layer immediatamente a lui sottostante, con quella aperta il layer può accedere alle funzionalità del layer sottostante e di tutti gli altri sotto di esso.
    - Si parla qui di *layer isolation*
- Con l'architettura aperta, si ottiene un alta manutenibilità e portabilità, nel secondo una maggiore efficienza in quanto si risparmia l'overhead delle chiamate in cascata.

Esempio del service layer visto come aperto; in questo modo, le richieste verso gli strati sottostanti "filtrano" più facilmente; questo concetto si estende e può diventare dannoso, in quanto molte richieste possono passare tra gli strati automaticamente (*sinkhole/voragine*).



Di sicuro un'architettura a strati è un buon punto di partenza per molte applicazioni, essendo molto *general purpose*, tuttavia richiede una *complessità aggiunta* (strati che tendono a separarsi e a dover comunicare solo tramite interfacce complicando il testing), data da problemi di *performance* (causa overhead lo spostamento/modifica di processi e le relative richieste) e di *tight coupling* (se non implementati correttamente, gli strati diventano difficilmente modificabili/mantenibili).

Questo tende ad andare verso le monolithic applications, sviluppate come "blocco unico" e relativamente semplici da sviluppare e distribuire, poiché tutti i componenti del sistema sono contenuti in un'unica unità. Tuttavia, possono anche essere più difficili da mantenere e scalare nel tempo, poiché le modifiche a una parte del sistema possono influenzare l'intero sistema. Inoltre, possono essere meno flessibili e adattabili alle mutevoli esigenze aziendali, avendo le componenti fortemente accoppiate tra di loro.

Ecco un'analisi delle architetture layered:

Characteristic	Rating	Description
Overall agility	↓	Small changes can be properly isolated, big one are more difficult due to the monolithic nature of the pattern
Ease of deployment	↓	Difficult for larger applications, due to monolithic deployments (that have to be properly scheduled
Testability	↑	Very easy to mock or stub layers not affected by the testing process
Performance	↓	Most of requests have to go through multiple layers
Scalability	↓	The overall granularity is too bread, making it expensive to scale
Ease of development	↑	A well know pattern. In many cases It has a direct connection with company's structure

L'architettura a microservizi è un pattern alternativo all'architettura monolitica che realizza la scomposizione di un'applicazione in servizi più piccoli e indipendenti, che possono essere sviluppati, distribuiti e scalati indipendentemente (*separately deployed units*), senza influenzare il resto del sistema.

Il principale vantaggio è andare verso la *decomposizione funzionale/functional decomposition*, tale che un servizio possa essere scomposto in moduli/componenti più piccole a prescindere dalla sua dimensione (*granularità*), tali che possa funzionare anche in remoto (accesso *distribuito*) tramite API.
Ogni applicazione a microservizi ha *il suo database separato*, implementato a seconda dell'applicazione, gruppi di applicazioni/API/gruppi di eventi/modalità di accesso, ecc.

Rispetto alle architetture precedenti:

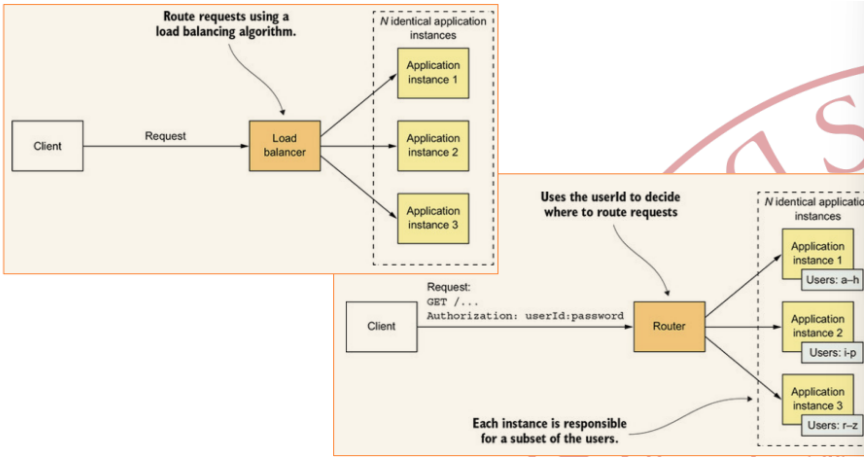
- È aperta alla *continuous delivery (CD)*, permettendo cicli di sviluppo rapidi non essendo le componenti strettamente accoppiate, ma testabili e consegnabili (deploy) indipendentemente
- Cerca di *semplificare* la nozione di servizio, permettendo l'implementazione ubiqua senza necessità di organizzazione da zero

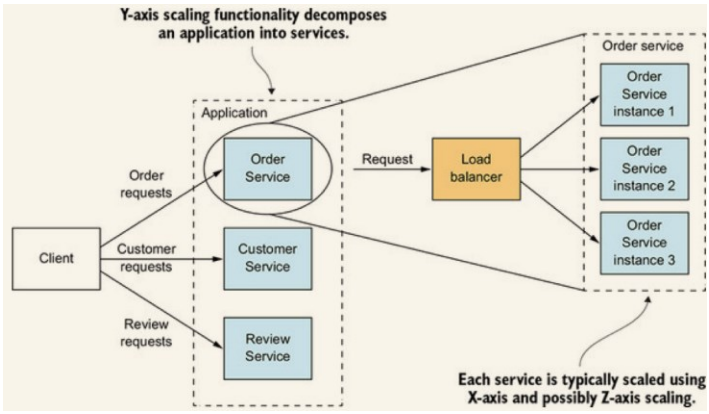
Lo *scaling* (ridimensionamento e adattabilità) di un'applicazione avviene secondo:

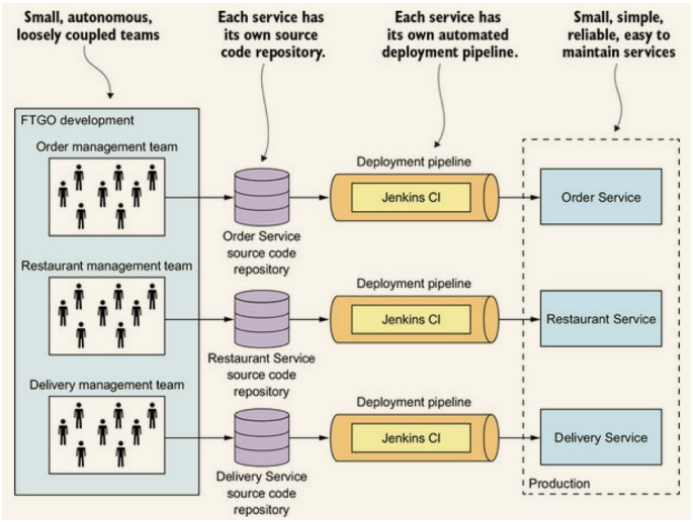
- Un'asse verticale - Y (la citata decomposizione funzionale)
- Un'asse orizzontale - X (in termini di numero di istanze)
- Un'asse obliquo - Z (in termini di numero di partizioni)

Abbiamo che:

- Il monolite scala solo lungo gli assi X e Z (immagine sopra)
- I microservizi utilizzano la decomposizione funzionale (coppia di immagini sottostanti la precedente)







Scritto da Gabriel

Un esempio sotto forma di codice Python:

```
# Service A: A simple Flask API that exposes a route to get the current time
from flask import Flask
import requests

app = Flask(__name__)

@app.route('/time')
def get_time():
    # Call Service B to get the current time
    response = requests.get('http://localhost:5001/time')
    current_time = response.json()['time']
    return {'time': current_time}

if __name__ == '__main__':
    app.run(port=5000)
```

In questo esempio, ci sono due servizi: Il servizio A è una semplice API Flask che espone un percorso per ottenere l'ora corrente e chiama il servizio B per ottenere l'ora corrente. Il servizio B è un'altra API Flask che ottiene l'ora corrente dal sistema e la invia al servizio A tramite una coda di messaggi utilizzando il broker di messaggi RabbitMQ.

```
# Service B: A simple Flask API that gets the current time from the system and sends it to Service A via a message queue
from flask import Flask
import datetime
import pika
app = Flask(__name__)

@app.route('/time')
def get_time():
    current_time = str(datetime.datetime.now())

    # Connect to the message broker
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    channel = connection.channel()

    # Declare a queue for the messages
    channel.queue_declare(queue='time')

    # Send the current time to the message queue
    channel.basic_publish(exchange='', routing_key='time', body=current_time)

    return {'time': current_time}

if __name__ == '__main__':
    app.run(port=5001)
```

Si può accedere al servizio A facendo una richiesta alla route /time e questo restituirà l'ora corrente ottenuta dal servizio B. Questa architettura consente di sviluppare e distribuire i due servizi in modo indipendente e di comunicare in modo asincrono attraverso la coda di messaggi.

L'Architettura Esagonale/Hexagonal Architecture è un pattern architetturale che consente agli input degli utenti o dei sistemi esterni di arrivare all'applicazione attraverso *porte* (interfacce di comunicazioni con attori esterni) e *adattatori* (che usano una tecnologia specifica per permettere ad un client di comunicare), tale che l'output venga inviato da una porta ad un adattatore.

In un'architettura esagonale, la logica di business centrale dell'applicazione è posta al centro, mentre gli strati circostanti, o "porte", rappresentano i vari modi in cui l'applicazione può interagire con il mondo esterno. Queste porte possono essere di ingresso, attraverso le quali l'applicazione riceve dati da fonti esterne, o di uscita, attraverso le quali l'applicazione invia dati a fonti esterne.

Scritto da Gabriel

Uno dei principali vantaggi dell'utilizzo di un'architettura esagonale è il fatto di isolare la business logic di base di un'applicazione dall'infrastruttura che la supporta. In questo modo è più facile testare e mantenere la logica di business, nonché cambiare o sostituire l'infrastruttura, senza influenzare la logica di base.

I vantaggi dell'utilizzo dell'architettura di porte e adattatori sono molti, quali:

- la possibilità di isolare completamente la logica dell'applicazione e la logica del dominio in un modo completamente testabile;
    - poiché non dipende da fattori esterni, il test diventa naturale e il mocking (moduli che simulano funzionamento di alcuni oggetti) delle sue dipendenze è facile;
- Consente di progettare tutte le interfacce del sistema "in base allo scopo" piuttosto che in base alla tecnologia, facilitando l'evoluzione dello stack tecnologico dell'applicazione nel tempo.

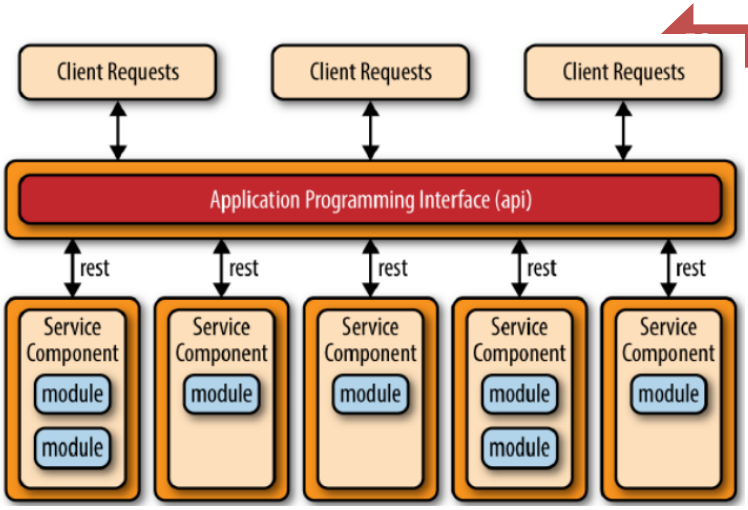
Un esempio di Hexagonal Architecture in Python:

```
# Core business logic
class Calculator:
    def add(self, x, y):
        return x + y

# Input port
class CalculatorInputPort:
    def get_numbers(self):
        return (3, 4)

# Output port
class CalculatorOutputPort:
    def print_result(self, result):
        print(result)

# Adapter
class CalculatorAdapter:
    def run(self):
        # Get input from input port
        input_port = CalculatorInputPort()
        x, y = input_port.get_numbers()

        # Use core business logic to calculate result
        calculator = Calculator()
        result = calculator.add(x, y)

        # Send result to output port
        output_port = CalculatorOutputPort()
        output_port.print_result(result)

# Client
adapter = CalculatorAdapter()
adapter.run()
```

In questo esempio, la classe *Calculator* rappresenta la business logic di una semplice calcolatrice in grado di sommare due numeri. La classe *CalculatorInputPort* rappresenta una porta di ingresso attraverso la quale la calcolatrice riceve input e la classe *CalculatorOutputPort* rappresenta una porta di uscita attraverso la quale la calcolatrice invia output.

La classe *CalculatorAdapter* funge da adattatore per collegare le porte di ingresso e di uscita alla logica aziendale di base. Infine, il metodo run della classe *CalculatorAdapter* viene chiamato dal client per eseguire il calcolo.

Piccola citazione a REST (Representational State Transfer), uno stile architetturale usato nei sistemi distribuiti che fornisce un insieme di vincoli da utilizzare per la creazione di servizi web. Le API REST si basano sul protocollo HTTP e utilizzano i metodi HTTP standard (come GET, POST, PUT, DELETE, ecc.) per eseguire diverse operazioni sulle risorse esposte dall'API.



Le API REST sono progettate per essere semplici, flessibili e scalabili e sono diventate una scelta popolare per la creazione di servizi web, perché possono essere facilmente consumate da un'ampia gamma di client, tra cui browser web, applicazioni mobili e altri server.

I servizi REST sono acceduti tramite API all'esterno (es. Google, Amazon, Yahoo).

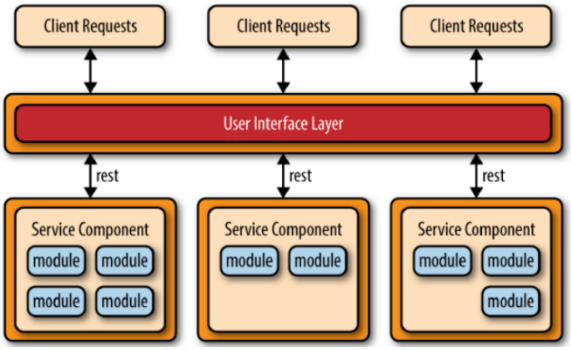
Un esempio di REST-based API:

```
GET /users HTTP/1.1
Host: api.example.com

HTTP/1.1 200 OK
Content-Type: application/json

[
  { "id": 1, "name": "Alice" },
  { "id": 2, "name": "Bob" },
  { "id": 3, "name": "Charlie" }
]
```

In questo esempio, il client invia una richiesta GET all'endpoint */users* dell'API e il server risponde con un elenco di utenti in formato JSON.

Questo è solo un semplice esempio e un'API REST reale potrebbe includere molti più endpoint e funzionalità. Tuttavia, il principio di base rimane lo stesso: l'API espone un insieme di risorse e consente ai client di recuperare e manipolare tali risorse utilizzando un insieme standard di metodi HTTP (come GET, POST, PUT e DELETE).

- Una topologia comune per un sistema che utilizza REST è un'architettura client-server, in cui un client invia richieste a un server e il server restituisce le risposte. In questo modello, il server espone un insieme di endpoint (URL) che rappresentano le risorse disponibili attraverso l'API e il client invia richieste a questi endpoint utilizzando metodi HTTP come GET, POST, PUT e DELETE.

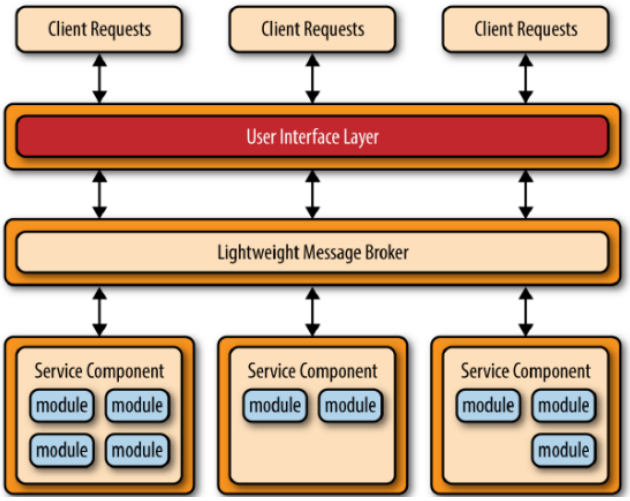


- Un'altra topologia comune per un sistema che utilizza REST è un'architettura a microservizi, in cui il sistema è suddiviso in un insieme di piccoli servizi indipendenti che comunicano tra loro tramite API. In questo modello, ogni servizio espone un insieme di endpoint che rappresentano le risorse che fornisce e i client possono inviare richieste a questi endpoint per recuperare e manipolare le risorse.

In generale, la topologia di un sistema che utilizza REST viene acceduta tramite web based client o *fat* client (basati su desktop e non solo sul server, in quel caso sarebbero *thin*). Dipende tutto dagli scopi (se andare a grana grossa oppure a grana fine).

La topologia a messaggi centralizzati (centralized message topology) è un modello di progettazione per la comunicazione tra sistemi in cui un *broker* di messaggi centrale è responsabile dell'instradamento dei messaggi tra i sistemi. In questo tipo di topologia, i sistemi che producono e consumano messaggi sono chiamati *client* e comunicano con il message broker attraverso un modello *publish-subscribe*.

Il message broker agisce come *hub* centrale del sistema ed è responsabile di ricevere i messaggi dai client, memorizzarli in una coda e instradarli verso la destinazione appropriata. Questo permette ai client di disaccoppiarsi l'uno dall'altro e di comunicare in modo asincrono, senza la necessità di connessioni dirette tra loro.



Uno dei principali vantaggi dell'uso di una topologia di messaggi centralizzata è che consente un'elevata scalabilità, in quanto il broker di messaggi può gestire un gran numero di client senza diventare un collo di bottiglia (*broker clustering*). Inoltre, offre un certo grado di affidabilità, in quanto il broker di messaggi può memorizzare i messaggi in una coda e consegnarli ai client anche se questi ultimi sono offline o hanno problemi.

La sfida principale consiste nel definire la giusta granularità dei componenti del servizio:

- Servizi a grana grossa (coarse-grained)
  - Non sono facili da distribuire, scalare, testare e non sono accoppiati in modo lasco
- Servizi a grana troppo fine (too fine-grained)
  - Richiedono l'orchestrazione e si trasformano in applicazioni SOA (Service Oriented Architecture), con tanti servizi stratificati a loro volta
  - Richiedono una comunicazione interservizi per elaborare una singola richiesta

Si usi sempre la comunicazione tramite DB, evitando quella service-to-service. Questo comporta la violazione del principio DRY (Don't Repeat Yourself), secondo cui "Ogni conoscenza deve avere una rappresentazione unica, univoca e autorevole all'interno di un sistema" → *no share of business logic*. Non si ha inoltre lavoro *transazionale*, essendo il sistema distribuito.

I pattern a microservizi hanno questa analisi:

Characteristic	Rating	Description
Overall agility	↑	Changes are generally isolated. Fast and easy deployment. Loose coopling
Ease of deployment	↑	Ease to deploy due to the decoupled nature of service components. Hotdeploy and continous delivery
Testability	↑	Due to isolation of business functions, testing can be scoped. Small chance of regression
Performance	↓	Due to distributed nature of the pattern, performance are not generally high
Scalability	↑	Each service component can be separately scaled (fine tuning)
Ease of development	↑	Small and isolated business scope. Less coordination needed among developers or development teams

Amministrazione di progetto e aggiudicazione appalti (Vardanega)

In questa lezione viene introdotto uno dei successivi modi di affrontare due specifiche lezioni: le *flipped classroom/lezioni rovesciate*. Ci vengono dati due temi su cui ragionare a livello di discussione e di proposte da parte nostra dovendo dare per forza un contributo di gruppo del 1° Lotto, in cui si discute "tra pari" di tematiche, coinvolgendo maggiormente l'esperienza di apprendimento e di classe.
Saranno affrontate in questa modalità le seguenti due tematiche:

- 15 novembre → Documentazione
- 17 novembre → Come si organizza lo sviluppo software

Il riferimento dell'amministrazione di progetto è il *way of working*, basato su specifiche piattaforme e strumenti (non tanto documenti). Questo è *a servizio* e non *a guida* del progetto.
L'amministratore ha il compito di equipaggiare, organizzare, e tenere corrente l'ambiente di lavoro di progetto, tramite l'utilizzo di *procedure*, regole e strumenti a supporto del way of working adottato.

Il way of working è linea guida per tutte le attività di progetto, che spiegano gli obiettivi dei processi (di sviluppo) adottati e ne guidano l'attuazione sistematica e disciplinata.
In particolare, vogliamo stabilire:

- quali attività svolgere (con quali input e con quali output)
- quali procedure eseguire (step-by-step)
- quali strumenti usare per farlo e come

Le azioni compiute devono essere sempre *approvate*, quindi si decide a priori in base a quali regole riteniamo valida una certa cosa, sulla base di un pensiero comune. Questo viene fatto per poter avere un processo fisso e possibilmente automatizzabile, soprattutto per quanto riguarda la verifica.

In merito *all'ambiente di lavoro*, si deve configurare un ambiente di progetto, tale da autenticarsi e trovare una cosa uguale per tutti, nel posto atteso con gli strumenti utili ai processi di produzione (primari, supporto, organizzativi). Normalmente, questi vengono creati in cloud, al fine di massimizzare portabilità e semplicità d'uso. Usare solo l'ambiente personale porta a copie non conformi e non compatibili tra di loro.
Esso deve essere:

- completo di tutti gli strumenti per svolgere le attività prevista
- ordinato per trovare facilmente ciò che si cerca
- aggiornato, tale da evitare copie locali non conformi oppure obsolete

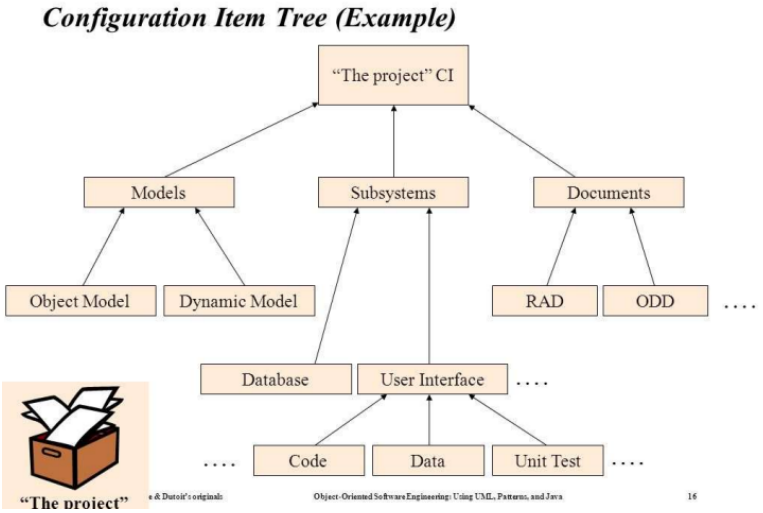
La gestione di progetto deve essere fatta sulla base di stime e pianificazioni, secondo uno strumento imparziale di rilevazione ("cruscotto"). Ne esistono vari tipi:

- strumenti di allocazione risorse e gestione "tempo-persona" (diagrammi di Gantt)
- strumenti collaborativi di controllo gestionale/qualità/coordinamento attività/gestione dei problemi-ticket (Issue Tracking System [ITS])
- strumenti collaborativi di gestione documentale, con controllo di accesso, sulle modifiche e sul ripristino (Google Docs, Overleaf)
- strumenti di versionamento e configurazione

Un progetto è un aggregato composito di parti distinte (specifiche, test, analisi, ecc.).
Ogni sottoprodotto (eseguibile/documento) composto da un insieme di parti è chiamato build.
Le componenti di una build devono cambiare insieme in maniera distinta (normalmente, facendo più cose, avremo più eseguibili/manuali/file sorgente), secondo una specifica *configurazione* (che va messa in sicurezza; nella continuous integration, ogni build è *baseline*). Ogni parte evolve a modo suo ed occorre metterle insieme in modo automatizzato.

Il way of working definisce delle regole precise di configurazione secondo il *Configuration Management*.

Il loro spazio di gestione è descrivibile come albero, in cui un oggetto è un *Configuration Item/CI* (ogni oggetto ha una sua identità unica, per nome, data, etc.) e, essendo un albero, ciascuno di questi è organizzato gerarchicamente (struttura complessiva è *configuration item tree*) e deve essere definita a monte *in modo automatizzato*.



In particolare, le attività di configurazione si compongono in varie fasi:

1) Attività di configurazione, composto di parti specifiche/sottoprodotti (configuration item)
2) Controllo di baseline, quindi insieme di CI consolidato ad istanti di tempo (milestone)
    1. Un progetto avanza tramite successioni di baseline, con più informazioni di controllo (più milestone)

Una milestone è una data di calendario che denota un punto di avanzamento atteso, sostanziato da una o più baseline. Le baseline garantiscono:

- Riproducibilità, ripristino
- Tracciabilità (dove siamo rispetto agli obiettivi)
- Analisi, valutazione, confronto (rispetto alle attese)

La qualità delle milestone viene definita sulla base di alcuni principi. Esse devono essere:

- *specifiche* per obiettivi di avanzamento e dimostrabili per risultati attesi;
    - in una successione naturalmente incrementale
- *coerenti* con la strategia di progetto e con esigenze di calendario;
    - significative per il team e per gli stakeholder
- *delimitate* per ampiezza e ambizioni
    - realisticamente raggiungibili
- *misurabili* per quantità di impegno necessario;
- *traducibili in compiti* assegnabili a *singoli* individui;
    - corrispondenti a uno sprint di metodo agile

Scritto da Gabriel

Sempre le attività di configurazione si prefigurano su:

- Controllo di versione, basato su un repository, DB centralizzato dove risiedono tutti i CI di ogni baseline con la loro storia completa. Ogni istanza è identificata per numero/caratteristiche/modifiche.
    - Permette di lavorare su vecchi e nuovi CI senza rischio di sovrascritture (*check-out*)
    - Permette di condividere il lavoro sullo spazio comune (*check-in*/*commit*)
    - Permette di verificare la bontà di ogni modifica di baseline (*build*)
- In questo senso:
    - una *versione* è un'istanza di prodotto funzionalmente distinta dalle altre
    - una *variante* è un'istanza del prodotto funzionalmente identica ma diversa per caratteristiche non funzionali
    - una *release* è un'istanza di prodotto resa disponibile a utenti esterni.
- Gestione delle modifiche, che avviene secondo una serie di richieste di modifiche (che provengono da utenti/sviluppatori/competizione, come difetti/mancanze/valore aggiunto ricavato)
    - Ogni richiesta di modifica va sottoposta a un rigoroso processo di analisi e verifica
    - Ogni decisione presa in esso va documentata in uno specifico verbale (con verifica di esito)

L'intero processo di gestione delle modifiche è noto come change management.
In particolare, ogni richiesta di modifica va gestita in modo sistematico e disciplinato:

- Tramite procedura di change request che identifica autore, motivo, urgenza
- Con stima di fattibilità (cosa fare, come, e come verificarne l'esito)
- Con costo atteso e valutazione di impatto
- Con decisione di approvazione del responsabile
- Documentato in verbale dedicato

Di ogni richiesta di modifica bisogna tenere traccia:

- Tramite issue tracking / ticketing
- Tracciandone lo stato di avanzamento fino a chiusura

Il Pattern Dependency Injection (Cardin)

(Eventuali riferimenti:
https://www.baeldung.com/scala/cake-pattern
http://blog.rcard.in/design/programming/fp/monad/2020/03/07/and-monads-for-all-reader-monad.html
https://www.baeldung.com/scala/tagless-final-pattern
Codice di esempio:
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/di)

A livello architetturale, vogliamo separare il comportamento di una componente dalla risoluzione delle sue dipendenze, in quanto collegare due componenti in modo esplicito ne aumenta l'accoppiamento, rendendo difficile la creazione di unit test e complicando riutilizzo e manutenzione.
Le dipendenze vanno *minimizzate*; sorgono immediatamente due tipi di problemi:

1) Dependency injection → Come dichiarare una dipendenza

In questo caso, gli oggetti o funzioni ricevono altri oggetti o funzioni da cui dipendono (quindi, vengono *iniettate* le dipendenze). La componente dichiara solo le sue dipendenze (minimizza il livello di accoppiamento) e, in questo modo, quando un oggetto o una funzione vuole usare un certo servizio, non deve sapere come sono costruiti. Ne esistono di vario tipo:

- *Field injection*, in cui la dipendenza si ha direttamente sui campi utilizzati.
    - (Pro) Si ha un codice relativamente semplice e compatto, in quanto basta *un'annotazione* Java (cioè dei metadati/informazioni sui blocchi di codice per gestire il tutto (@Autowired)

- (Contro) Si può continuare ad aggiungere dipendenza ai parametri liberamente, ma ciò appesantisce la responsabilità dei blocchi di codice. Inoltre, *la classe non è più in grado di gestire le proprie dipendenze*. Non è possibile creare oggetti immutabili e, dato che il codice è accoppiato con i campi, il codice non può essere istanziato (complica il testing).

```
public class MovieLister {
   // Risolta automaticamente dall'injector. Piu' conciso ma
   // meno verificabile
   @Inject MovieFinder finder;
}
```

- *Constructor injection*, in cui le dipendenze sono fornite come parametri del costruttore.
  - (Pro) Permettono di costruire oggetti validi dallo loro istanziazione, creando oggetti immutabili. *Bisogna avere fisicamente un oggetto completo e costruito*.
    Risulta inoltre più facile da testare, in quanto gli oggetti non vengono istanziati senza le dipendenze necessarie.
  - (Contro) Difficilmente i parametri hanno un comportamento isolato; pertanto, per far interagire il costruttore con gli altri elementi, è necessario adattare il framework usato sulla base delle dipendenze tra i singoli elementi.

```
public class MovieLister {
   private MovieFinder finder;
   public MovieLister(MovieFinder finder) { // Dichiarazioe dip.
      this.finder = finder;
   } // ...
}
```

- *Setter injection* (*Method injection*), in cui le dipendenze sono impostate chiamando un metodo, in particolare con quelli impostanti (*setter*). In questo modo, gli oggetti permettono agli iniettori di manipolare le dipendenze in qualsiasi momento.
  - (Pro) Offre flessibilità, in quanto non crea nuove istanze dell'oggetto e passa prima per il costruttore di default, poi per il metodo setter. Si ha un'iniezione *parziale* delle dipendenze: iniettiamo dipendenze solo se necessario. È più leggibile e lavora meglio con le gerarchie di classi.
  - (Contro) Rende difficile assicurare che tutte le dipendenze siano iniettate e valide prima che l'oggetto sia usato. Può succedere infatti che si faccia l'override di alcune dipendenze delle sottoclassi, complicando la gestione e lanciando anche eccezione (dipendenza circolare).

A livello di ordine, preferiamo:

(1) Constructor injection → Facile da testare (basta una *new*), immutabile (le dipendenze non cambiano), codice chiaro e sicuro (senza stati intermedi) e semplifica il design
(2) Method injection → Utile quando gli oggetti hanno più di una proprietà da dover testare e rende più semplice la gestione
(3) Field injection → L'opposto della constructor, peggiore da gestire ma utile in casi limitati

Andando verso un caso concreto, normalmente si creano degli *unit test*, al fine di verificare l'esecuzione e il comportamento delle singole componenti.

- Struttura → Given (Causa) – When (Condizione) – Then (Lancio eccezione/Verifico)
- Problema → Le dipendenze rendono le classi non isolate e si testa un codice *che non è più quello iniziale*.
  - Questo è ulteriormente complicato quando si hanno istanze di database (non controllabili univocamente) oppure metodi statici (dipendenze nascoste tra i metodi e le classi)
- Possibile soluzione: i *mock* (metodi vuoti usati come test istruiti per eseguire un compito specifico).
  - Questi servono per verificare sicuramente un comportamento singolo e *isolare* completamente il singolo caso dalle dipendenze.

2) Risoluzione delle dipendenze

In generale, le dipendenze vengono usate per astrarre dalle implementazioni e *iniettare* un servizio direttamente quando deve essere utilizzato. In questo modo, evito di creare tipi concreti e non creo istanze di oggetti inutili. Il testing del codice è migliore (isolo le componenti e le riutilizzo), così come la resilienza (per la manutenzione e correzione delle componenti) e la flessibilità (posso scegliere varie componenti a runtime). Normalmente, le dipendenze sono tante; ecco perché esiste il *grafo delle dipendenze/grafo degli oggetti/albero delle dipendenze*.

Per poterle risolvere singolarmente all'interno del grafo:
- in caso di un'applicazione piccola, è possibile testare i singoli tipi ricreandosi delle istanze manualmente (in quanto poche, scomodo ma fattibile)
- in caso di un'applicazione grande, si usa un *dependency injector* che costruisce un'istanza sulla base del *contesto* dell'oggetto (interpretando il tipo come chiave-valore come fosse una mappa [map], ritornando la dipendenza dell'oggetto e costruendolo quando serve).

Tutto ciò non è facile da realizzare: non abbiamo più il controllo sul ciclo di vita degli oggetti.
La dependency injection segue il principio di *Inversion of Control* (detto anche *Hollywood Principle*), in cui il framework ha fisicamente il controllo sugli oggetti e non noi.
Questo principio si realizza nel momento in cui si usa un *injector*, che ci permette di risolvere le dipendenze in automatico e *ci consente di concentrarci a programmare la business logic*. Inoltre, si estende la modularità del programma, dato che previene i side effects quando si rimpiazza un modulo e si disaccoppia l'esecuzione di un task dall'implementazione.

Gli injector utilizzati sono:
- *Google Guice* (letto Juice, in *italiano* "gius"), framework proprietario di Google, piccolo in dimensioni e risolve le dipendenze facilmente.
    - Esso permette di definire le dipendenze automaticamente alla creazione degli oggetti, rendendo il codice più modulare e semplice da comprendere, dato che rimpiazza alcune componenti con dei mock. Riduce il codice ripetuto e permette di gestire grafi di dipendenze complessi, promuovendo soprattutto la constructor injection.
    - La sua sintassi può essere molto verbosa al fine di eseguire i *bindings* con gli oggetti e anche difficili da leggere. Si basa sulla reflection, dunque può avere un impatto negativo sulle performance.
    - Realizza la *class injection* (*@Inject*) e viene configurato in Java tramite l'estensione di *AbstractModule*. Non è possibile avere due oggetti dello stesso tipo.

- *Spring*, framework di sviluppo Java presente e consolidato.
    - Si configurano direttamente i *bean*
        - Essi sono oggetti gestiti da un container Spring, che hanno un loro file di configurazione XML con classe, dipendenze e argomenti/costruttori
    - L'iniezione di dipendenze in Spring consente di specificare le dipendenze dei bean in un file di configurazione, anziché codificarle nelle classi dei bean. Quando il contenitore Spring viene inizializzato, crea i bean e li collega tra loro utilizzando la configurazione fornita.
    - Per utilizzare la dependency injection in Spring, si definiscono innanzitutto i bean in un file di configurazione, in genere scritto in XML. A ogni bean viene assegnato un ID e si possono specificare le sue dipendenze includendo riferimenti ad altri bean tramite il loro ID. Si possono anche specificare gli argomenti del costruttore o le proprietà che devono essere impostate sul bean quando viene creato.

Ecco alcuni vantaggi dell'uso della dependency injection:

- Promuove la *separation of concerns* tra i diversi componenti di un sistema, poiché ogni componente è responsabile di un compito specifico e riceve le sue dipendenze dall'esterno.
- Rende più facile testare i componenti in modo isolato, poiché le loro dipendenze possono essere testate con mock o con stub.
- Consente una maggiore flessibilità e una manutenzione più semplice, poiché le dipendenze di un componente possono essere modificate o sostituite senza modificare il componente stesso.

Tuttavia, l'uso della dependency injection presenta anche alcuni potenziali svantaggi:

- Può aggiungere complessità al codice, poiché richiede l'uso di classi o metodi aggiuntivi per gestire le dipendenze.
- Può rendere più difficile la comprensione dell'architettura complessiva di un sistema, poiché le dipendenze di un componente non sono definite esplicitamente all'interno del componente stesso.
- Se il meccanismo di iniezione delle dipendenze non è ottimizzato, le prestazioni possono essere scarse.

Esempio concreto di una Constructor Injection con Python e che esprime i pro e contro detti:

```
# A service that logs messages to the console
class Logger:
    def log(self, message):
        print(message)

# A component that uses the logger service
class Component:
    def __init__(self, logger: Logger):
        self.logger = logger

    def do_something(self):
        self.logger.log('Doing something...')

# Create an instance of the logger service and inject it into the component
logger = Logger()
component = Component(logger)

# Use the component
component.do_something()
```

In questo esempio, la classe *Component* dipende dalla classe *Logger* per registrare i messaggi. Invece di creare un'istanza della classe *Logger* all'interno della classe *Component*, l'istanza del *Logger* viene iniettata nella classe *Component* attraverso il costruttore. Ciò consente alla classe *Component* di utilizzare il servizio *Logger* senza doversi preoccupare di come è implementato o da dove proviene.

Diario di bordo – Way of Working e Documentazione

Dubbi riscontrati:

- Documentazione
    - I documenti vanno presentati tra proponenti e committenti e si definiscono tecnologie e periodi di riferimento

- Organizzazione asincrona
    - Definizione dei temi di discussione e riscontro sugli avanzamenti. Ogni giorno di calendario ha momenti diversi per ciascuno, fissando vari momenti per ciascuno all'interno di un calendario unico

- Atomizzazione compiti e Calcolo tempistiche
    - Dimensionare i compiti sula base del quanto di tempo definito. Si parte dal backlog, che considera le cose da fare e si veda facilmente l'insieme delle cose da fare. Per esempio, apprendere una tecnologia in un modo unico usando fonti diverse. Ogni azione è completata sulla base di compiti più piccoli, ma ognuno va spezzato come si deve.
    - Strumento di base è un tabellone delle azioni (backlog), gestito in modo dinamico. Ogni work item deve essere spezzabile, stando dentro gli intervalli utili. Vari strumenti ci aiutano a farli. Esso è un documento che cresce verticalmente e si spezza orizzontalmente.

    - Al suo interno, ci sono cose da fare e gradi di urgenza (richiede esperienza mettere date sensate). Esso è reversibile, perché si può andare avanti e tornare indietro. Atomizzare richiede compiti piccoli, sottostimati, per non sprecare troppo tempo anche in modo asincrono. Sulla base degli sprint, si definisce in modo agile il tabellone.
    - I verbali vanno fatti ogni volta che si ha un avvio e una conclusione di uno sprint. Essi sono un'azione sincrona, perché serve che tutti capiscano *a che punto siamo*, dando un giudizio d'urgenza sulle singole azioni insieme.
    - Sulla base del backlog, si verificano le attività a fine sprint e si pianifica tutto in modo successivo e concreto alla *cerimonia* (fine del periodo agile). L'intento del diario di bordo è esplorativo; piuttosto che "ho fatto" dico "ho provato e sono/non sono riuscito".

- Quante/i repo/struttura
    - Sulla base della definizione di "progetto", i repo sono "di prodotto", quindi sono molteplici. La visione è lato committente (definendo incontri e avanzamento prodotto in determinati punti temporali), quindi è lui stesso che definisce quante e quali sono. Alla meno di casi particolari, al committente serve software e documentazione utile *al suo lavoro*.
    - Si creano le build nel corso del tempo, facendo capire lo stato del lavoro al committente. Per esempio, si pensi agli aggiornamenti software: si va su un link in cui sta l'eseguibile, all'interno di un posto designato. Le build sono automatiche e non manuali.

Analisi dei requisiti (Vardanega)

(Eventuale approfondimento: https://ieeexplore.ieee.org/document/720574
In passato l'Analisi dei Requisiti, nei corsi di Tullio, era nota come "Ingegneria dei Requisiti")

Partiamo dalla definizione di *requisito*, prodotto dal glossario IEEE ("Institute of Electrical and Electronics Engineers", pronunciato I-triple E, associazione internazionale di scienziati professionisti con l'obiettivo di promuovere e standardizzare le tecnologie).
1) La capacità necessaria a un utente per risolvere un problema o raggiungere un obiettivo:
   - lato *bisogno*/*need* (da parte di chi pone il problema)
2) La capacità necessaria a un sistema per soddisfare una aspettativa/adempiere ad un obbligo:
   - lato *soluzione* (quello che il software deve fare per rispondere ai needs)
3) Descrizione documentata di una condizione di una delle due tipologie precedenti

Si ricordi che:
- il capitolato esprime il punto di vista di lato bisogno, quindi requisiti utente/needs;
- i gruppi devono trasformare i requisiti utente in lato soluzione, definendo così il *design*, trasformando il tutto in requisiti software.

Concettualmente, abbiamo i seguenti punti:
- Il *committente* studia un problema di interesse e fissa le proprie aspettative sul prodotto finale, sulla base dei *requisiti utente* fissati dal *capitolato*.
- Il *fornitore* deve capire come soddisfare quelle richieste, valutandone la fattibilità (tecnica, gestionale, economica), insieme al proprio interesse strategico (obblighi, opportunità).
- La prima *milestone* del calendario di progetto arriva quando committente e fornitore trovano accordo su: gli obiettivi funzionali (*analisi dei requisiti*), gli obiettivi di qualità (*piano di qualifica*), i tempi e i costi del progetto (*piano di progetto*).
- Il *contratto* termina con la valutazione del committente sul prodotto, con accettazione condizionata a buon esito di collaudo (*validazione*), con dimostrazione di soddisfazione di ogni requisito utente

Fra le principali cause di fallimento di un progetto sta al primo posto l'aver capito male i requisiti e la maggior parte delle cause collegate ruotano attorno a questa motivazione.
Possiamo dire infatti che:
- Un requisito può essere compreso solo *parzialmente* (piccolo/*coarse*) e deve essere maggiormente espanso, possibilmente semplificando la sua realizzazione all'interno del prodotto finale (rispondendo al bisogno *grande/grossolano*).
- Esistono inoltre requisiti *impliciti*, non definiti esplicitamente dalla richiesta del cliente ma compresi realizzando il prodotto o ampliando la propria conoscenza del contesto.
- Il requisito utente è soddisfatto quando tutti i requisiti *tecnici* (*needs*) a livello *macroscopico* sono soddisfatti. Spezzati in parti piccole, si ottengono i requisiti *soluzione* (a livello *microscopico*).

Per essere sicuri di non essersi dimenticati requisiti durante lo sviluppo, è definito il tracciamento dei requisiti. La via logica è *bidirezionale* (dal problema alla soluzione e viceversa); al committente interessa il senso inverso (quanto fatto per soddisfare un need).
L'analisi dei requisiti deve facilitare sia il tracciamento che la verifica. Chi fissa un requisito di lato soluzione deve anche stabilire come verificarne il soddisfacimento, facendo attenzione al costo e complessità di verifica, al fine di non dimenticare nessun requisito.

Abbiamo molti più needs lato utente che soluzione. Occorre quindi tracciarli mantenendo le dipendenze, spezzando 1 need in n requisiti. Per mantenerli, ognuno ha un suo identificativo (per esempio, R $1.1.4$). Per riuscire a collegarle, vengono normalmente tracciate le dipendenze (chiave – valore), di solito attraverso strumenti appositi nel mondo reale, per noi tramite basi di dati.

Scritto da Gabriel

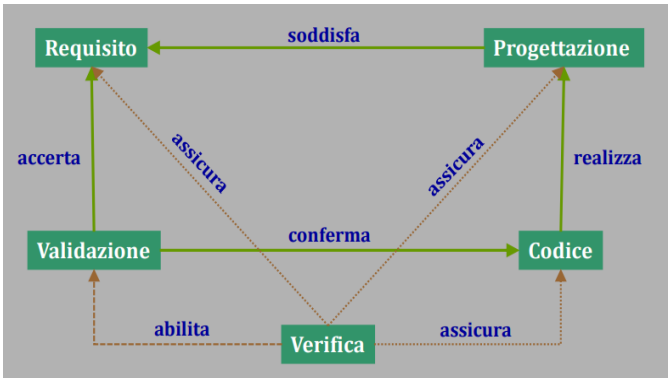
Individuando un *glossario* utile in questo contesto:

- Verifica
  - Accertarsi che lo svolgimento delle attività di sviluppo non introduca errori
  - *Did I build the system right?* - attenzione rivolta al *way of working*
- Validazione
  - Accertarsi che il prodotto corrisponda alle attese
    - Se necessario, i requisiti utente sono modificati da incontri con il proponente
  - *Did I build the right system?* - attenzione rivolta al *prodotto finale*
- Piano di Qualifica
  - Dire come svolgeremo le attività di *V&V* (*Verification* & Validation) nel progetto
    - Essa è attività definita *nel momento stesso* di individuazione dei requisiti
  - E con quali obiettivi di qualità

V & V = Qualifica

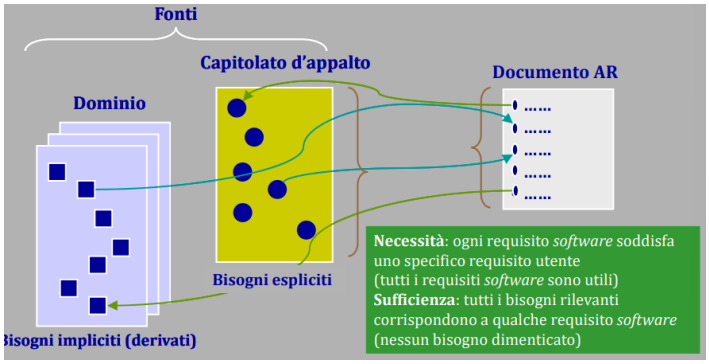
L'approccio sensato è arrivare alla validazione avendo la certezza di fare la cosa giusta.

Cio è possibile.
Avendo tracciato il mio albero/grafo delle dipendenze, riesco ad avere un approccio *top-down*, spezzando il requisito in sottorequisiti e un approccio *bottom-up*, risalendo con coscienza di aver risolto i problemi "più piccoli" e quindi il requisito intero.



Una possibile idea di tracciamento secondo l'analisi dei requisiti, ponendo requisiti, identificatori e raggruppamenti. Quando arrivo qui, mi devo chiedere:

- I requisiti microscopici sono tutti necessari?
  - Se sono lì, sono un pezzo del problema e sono parte di uno scenario. Esistono infatti i needs *impliciti* (derivano da aspettative naturali del dominio d'uso/abitudine/ovvietà che sia in quel modo lì) e i needs *espliciti* (richiesti dal committente)



- Come scoprire i requisiti impliciti?
  - Occorre fare un lavoro di interpretazione, identificando un modo nostro di agire e comprendere ed un certo numero di requisiti (normalmente, alcune decine). Un buon modo di trovarli è fare tante domande al proponente e farsi tante domande.

Sintetizzando tutti i passaggi di *analisi dei requisiti* (individuando il problema e possibile soluzione):

- Studio dei bisogni e del dominio d'uso
- Comprensione del problema dal lato dei bisogni tramite *scenari* di caso d'uso
- Raggruppamento degli scenari per affinità, individuando parti e gerarchie
- Classificazione e tracciamento dei requisiti, dialogando anche con il committente e gli stakeholders

In merito invece ai *processi di supporto all'analisi* (usando possibilmente grafici/tabelle):

- documentazione, che raccoglie i risultati dell'analisi, descrive dialoghi con il committente, carpisce feedback e pone le basi del design;
- verifica di adesione al way of working;

- gestione e manutenzione dei prodotti dell'analisi, basati su:
    - tracciamento dei requisiti;
        - per quelli impliciti, si discute con il committente e si mette tutto a verbale (partendo da parte nostra con scenari sulla base delle nostre idee sugli utenti e poi essi rispondono). Il verbale è parte del capitolato e del tracciamento;
    - gestione di versione e configurazione;
        - se dovessi sbagliare, torno indietro alle versioni precedenti, sapendo le relazioni che sussistono tra una e l'altra;
    - gestione dei cambiamenti (change management);
        - si fanno cose giuste e si comprende il senso delle cose sbagliate, correggendo quanto fatto. Il cambiamento è disciplinato (ragionamento collettivo sulla base dell'analisi del problema e delle possibili soluzioni);

Le qualità desiderabili per la specifica dei requisiti (idealmente, wishlist):
- priva di ambiguità (linguaggio ristretto/limitato/privo di interpretazioni)
- corretta (devo aver fatto l'attività di verifica)
- completa (la verifica individua, grazie al tracciamento, se c'è tutto)
- verificabile (operando una verifica che costi il meno possibile, possibilmente automatizzabile)
- consistente (non può contenere aspettative contraddittorie)
- modificabile (ci si accorge realizzando il prodotto che ci sono dei nuovi requisiti e devono essere implementati in modo scalabile)
- tracciabile (muoversi nelle due direzioni)
- ordinata per rilevanza (sulla base dell'urgenza, non tutto è uguale lato needs/soluzioni)

Continuazione Analisi dei Requisiti/RA e Info Lezioni Rovesciate/LR

Conviene ridurre i requisiti a «*grana fine*», affinando, e suddividendo i requisiti utente, aggiungendone ampliando lo sguardo sul problema; una buona tecnica di AR/Analisi dei Requisiti è un *brainstorming*, in cui tutti discutono e propongono in modo paritario idee e spunti in modo ordinato.
Ciascuno di questi ha un tema noto a priori, su cui tutti sono preparati partendo da alcuni input fissi.

Un brainstorming ha varie componenti:
- *timekeeper*, colui che controlla la gestione del tempo (entro la scansione oraria prevista), gestendo il lavoro sulla base di un tempo limitato (*time bounded*), tipicamente un'ora vera di lavoro. È la persona che mantiene la discussione equa, sia a livello di discussione che di ruoli, e fa in modo che ciascuno abbia un tempo di parola massimo e dica ciò che pensa entro quel limite;
- *scriba*, colui che tiene traccia di ciò che accade e mantiene i punti essenziali della discussione (*highlights*/punti salienti), senza interromperla.

Tornando all'analisi dei requisiti:
- Nella fase di *validazione*, i requisiti utente sono ampiamente dimostrabili tramite la fase di *test* e di revisione, capendo se sono stati implementati correttamente.
- Nella fase di *verifica*, il fornitore capisce se ha lavorato bene alla fine del periodo precedente e sta continuando a lavorare bene. L'obiettivo è arrivare alla validazione sapendo l'esito certo di correttezza; questo funziona se e solo se i passi precedenti sono corretti
    - Occorre sempre soddisfare requisiti lato soluzione: questo succede facendo già bene il design, verificabile tramite il cruscotto.

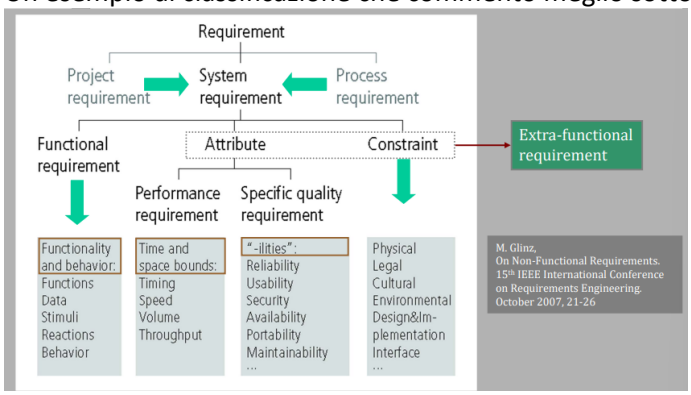
La corretta verifica dei requisiti avviene secondo una *checklist*, cioè una lista procedurale basata su una serie di controlli non ambigui e usati secondo un ordine specifico, al fine di verificare che tutti i passi siano stati eseguiti correttamente non avendo dimenticato nulla.
Una checklist fatta bene deve essere completa, corretta, non ambigua e tracciabile.

Scritto da Gabriel

La tecnica principale allo stato dell'arte per capire i requisiti nascosti è la *classificazione dei requisiti*, che cerca di comprenderli identificandoli sulla base di:

- Interviste al committente (analisi/discussione di scenari, riassunte in *verbali*)
- Acquisizione di conoscenze: *brainstorming*
- Consolidamento delle conoscenze di dominio: *glossario*
- Prototipazione interna/esterna

Un esempio di classificazione che commento meglio sotto:



- requisiti di progetto (*project requirements*), che sono le finalità, gli obiettivi e i risultati specifici che devono essere raggiunti per completare con successo un progetto (es. timeline/budget/risorse);
- requisiti di sistema (*system requirements*), che sono le caratteristiche o le capacità che un prodotto, un sistema o un componente deve possedere per soddisfare le esigenze delle parti interessate e adempiere allo scopo previsto. Questi possono includere:
    - requisiti <u>funzionali</u>: definiscono i compiti o le azioni specifiche che il sistema deve essere in grado di eseguire;
    - requisiti <u>non funzionali/attributi/extrafunzionali</u>: definiscono gli attributi o i vincoli che il sistema deve soddisfare. Questi requisiti possono includere affidabilità, usabilità, manutenibilità e scalabilità;
    - vincoli <u>qualitativi</u>: descrivono le qualità richieste dal prodotto e possono essere validati soltanto tramite verifiche ad hoc;
- requisiti di <u>vincolo</u> (*constraints*), che sono un tipo di requisiti di attributo che definiscono limiti o confini specifici che un prodotto, un sistema o un componente deve rispettare. Questi requisiti possono includere vincoli fisici, come limiti di dimensioni o di peso, o logistici, come vincoli di budget o di tempo.

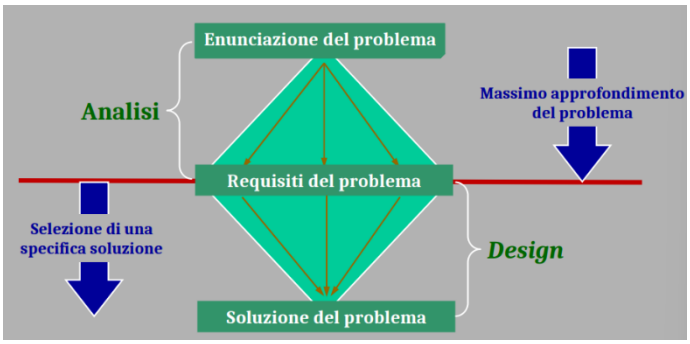
Anche i requisiti soluzione possono essere nascosti; in genere, i requisiti hanno diversa rilevanza e utilità, che va negoziata e concordata con il committente. *L'analisi cerca di trovarli tutti;* siccome il costo può essere troppo alto*, si ordinano per livelli di rilevanza,* come segue:

- <u>Obbligatori</u>
    - Irrinunciabili per qualcuno degli stakeholder
- <u>Desiderabili</u>
    - Non strettamente necessari ma a valore aggiunto riconoscibile
    - Implementati se possibile a livello di tempi/costi
- <u>Opzionali/Facoltativi</u>
    - Relativamente utili oppure contrattabili più avanti nel progetto

I requisiti sono *permeabili*, in quanto si possono spostare di rilevanza; ciò avviene discutendo direttamente con gli stakeholders. Ad un certo punto, diventa baseline e cambierà soltanto in casi particolari (se non riuscissimo ad implementarli per ragioni di tempo). Per fare ciò, non devono essere tra loro contradditori.

Nell'analisi, si cerca in ampiezza → si trovano *tutti* i bisogni lato soluzione
Nel design, si cerca in altezza → si trova *una* soluzione che soddisfi *tutti* i bisogni

Il numero di soluzioni deve lavorare secondo queste due dimensioni, considerando tempi, complessità e costi.



Il confine che accomuna analisi e design è l'individuazione a basso livello delle componenti (*moduli*) per scopi diversi; la prima cerca di capire punti di forza/debolezze del prodotto, mentre la seconda cerca di creare un impronta sul come il prodotto finale dovrebbe apparire.

La documentazione, di per sé, è ambigua a livello interpretativo; servono *norme redazionali* per evitare espressioni ambigue e garantire termini consistenti. L'uso di metodi formali può aiutare in questo (es. diagrammi/grafici/formule/UML piuttosto che semplice testo)

Un esempio standard ISO/IEC/IEEE per la documentazione di specifica dei requisiti:

1. **Introduction**
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Product overview
        - 1.3.1 Product perspective
        - 1.3.2 Product functions
        - 1.3.3 User characteristics
        - 1.3.4 Limitations
    - 1.4 Definitions
2. **References**
3. **Requirements**
    - 3.1 Functions
    - 3.2 Performance requirements
    - 3.3 Usability requirements
    - 3.4 Interface requirements
    - 3.5 Logical database requirements
    - 3.6 Design constraints
    - 3.7 Software system attributes
    - 3.8 Supporting information
4. **Verification**
    (parallel to subsections in Section 3)
5. **Appendices**
    - 5.1 Assumptions and dependencies
    - 5.2 Acronyms and abbreviations

In essa si ha il compito di:
- Ricercare chiarezza espressiva
  - L'uso del linguaggio naturale rende difficile coniugare chiarezza con facilità di lettura
- Ricercare chiarezza strutturale
  - Separazione tra requisiti funzionali e non-funzionali
  - Classificazione precisa, uniforme e accurata
- Ricercare atomicità e aggregazione
  - Requisiti elementari
  - Correlazioni chiare ed esplicite

Deve essere eseguita su un documento organizzato secondo la *checklist*:
- Tramite *walkthrough* (a largo spettro)
- Oppure tramite *ispezione* (lettura mirata/strutturata)

I requisiti specificati devono essere *necessari* e *sufficienti*; di ciò si tiene conto in apposita *matrice delle dipendenze* (cioè, una tabella in termini non "tulliesi", presenti sempre in documentazione).

Gli stati di progresso con cui poter valutare il grado con cui i requisiti sono individuati e implementati secondo *SEMAT* secondo alcuni termini precisi:

- Conceived – Concepito
  - Il committente è identificato e gli stakeholder vedono sufficienti opportunità per il progetto
- Bounded – Limitato
  - I bisogni macro sono chiari, i meccanismi di gestione dei requisiti (configurazione e cambiamento) sono fissati
- Coherent – Coerente
  - I requisiti sono classificati e quelli essenziali (obbligatori) sono chiari e ben definiti
- Acceptable – Accettabile
  - I requisiti fissati definiscono un sistema soddisfacente per gli stakeholder
- Addressed – Soddisfatto
  - Il prodotto soddisfa i principali requisiti al punto da poter meritare rilascio e uso
- Fulfilled – Realizzato

  - Il prodotto soddisfa abbastanza requisiti da meritare la piena approvazione degli stakeholder

L'approccio all'analisi, quindi, è data da:
- un approccio ripetibile, atto a considerare le *funzionalità* del sistema, capendo precisamente le relazioni e i termini secondo un linguaggio con una sintassi (UML), capendo:
  - generalizzazione (tutti i sottocasi fanno parte del caso principale);
  - estensione (comportamento imprevisto/passi aggiuntivi rispetto al caso principale);
  - inclusione (quando uno scenario di caso d'uso è duplicato per specifici sottocasi);
- ogni funzionalità si vede come oggetto, orientando il tutto al *riuso* delle *componenti;*
- effettuare uno studio di fattibilità, capendo costi, benefici, risorse e scadenze;
  - esso permette di stabilire se il sistema in questione sia redditizio e valuta nel complesso rischi, costi e benefici, descrivendo se il progetto aderisce agli obiettivi generali a livello tecnologico ed economico secondo le esigenze del gruppo fornitore
- classificare i requisiti studiando il dominio e assicurandosi siano necessari e sufficienti;
- siano validi atomicamente rispetto a tutte le loro qualità.

Model View Controller e Pattern Derivati (Cardin)

(Riferimento codice presentato a lezione: https://github.com/rcardin/swe-imdb/tree/main/cli
Altri riferimenti:
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/mv/mvc
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/observer)

Fino ad oggi ci siamo concentrati sul backend (architettura visibile all'esterno); ora ci concentriamo sul frontend (cioè, tutto ciò che possiede una UI), quindi architettura logica dotata di UI.
Alla persistence logic sostituiamo la presentation logic, ottenendo delle applicazioni B2C (Business to Consumer) e non solo B2B (Business to Business). Quindi, la corrispondenza è:

$$Presentation\ Logic \iff View$$
$$Application\ Logic \iff Controller$$
$$Business\ Logic \iff Service(Model)$$

Avendo la corretta separazione di compiti, riesco a sviluppare in modo parallelo e in meno tempo. La parte più complessa da sviluppare è la Business Logic, rimanendo sempre la componente più isolata. Inizialmente, si è cercato di andare sempre più verso la separazione logico-grafica (*separation of concerns*), frammentando le competenze tra persone e compiti dell'applicazione.

Questo problema nasce dall'interazione degli utenti con diverse piattaforme, evitando di duplicare il codice per frammentare tutti questi comportamenti. In particolare:
- si vuole permettere l'accesso ai dati tramite *viste* differenti (HTML, JS, XML, JSON, etc.), tale da modificare i dati attraverso diverse interazioni con i client (richieste HTTP, messaggi, etc.)
- il supporto a diverse viste non deve influire sulle componenti che forniscono e funzionalità base

Ormai abbiamo capito: ciò dipende dall'uso di diverse *interfacce*, che stabiliscono le regole di accesso.
In altri termini, i Model View ragionano con:
- Il Model che rappresenta la business logic
- La View che rappresenta l'interfaccia utente
- Il Controller che è il collegamento tra Model e View

Possiamo fisicamente mantenere e sviluppare Model e View indipendentemente rendendo più facile cambiare/aggiornare componenti senza influire sugli altri.
L'accoppiamento lasco si garantisce con comunicazione ad interfacce.

Prima di introdurre concretamente il concetto, apriamo una parentesi sull'*Observer pattern*, che consente di mostrare facilmente tutti i cambiamenti fatti ad un oggetto in modo veloce.
Fulcro di questo pattern sono i *Subject*, cioè degli oggetti il cui stato è osservato a lungo termine (si monitorano i cambiamenti, mandando notifiche agli *Observer* che si aggiornano in risposta) e viene notificato senza cambiamenti ai *Publisher*, avendo a loro volta altri oggetti che ne tracciano i cambiamenti, cioè i *Subscriber*.

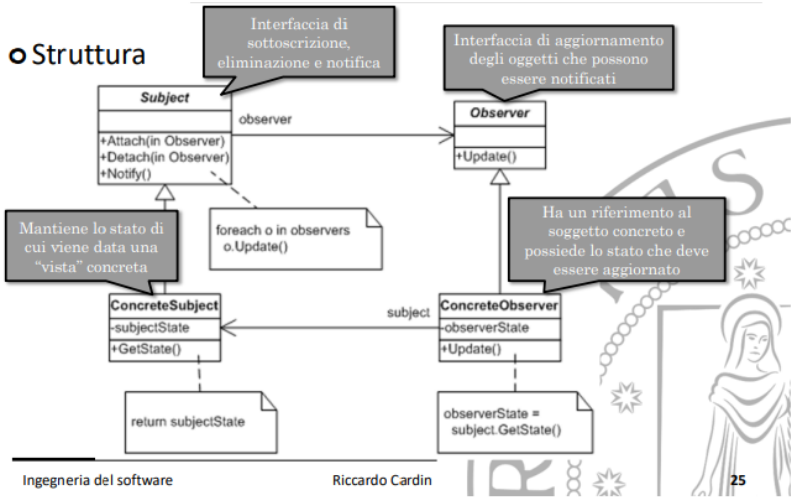
In questo pattern, il *Subject* e gli *Observer* sono disaccoppiati e gli *Observer* possono essere informati dei cambiamenti di stato del *Subject* senza che il *Subject* debba conoscere gli *Observer* o il modo in cui sono implementati.

Il pattern *Observer* è utile per implementare *sistemi guidati dagli eventi (event-driven)*, in cui il *Subject* rappresenta una sorgente di eventi e gli *Observer* rappresentano gli ascoltatori di eventi. Permette di notificare gli eventi agli *Observer* in modo flessibile e disaccoppiato, senza che l'origine dell'evento debba mantenere un elenco di *Observer* o sapere come notificarli.

Questo pattern viene introdotto perché è l'evoluzione naturale del concetto di separazione vista/struttura *preservando il controllo sui dati in modo facile.* In particolare:

- Siamo in grado di associare più "viste" ad una astrazione (aumentando il riuso dei tipi)
- Il cambiamento di un oggetto richiede il cambiamento di altri oggetti (pur non conoscendo quanti)
- Notificare oggetti senza fare assunzioni su quali siano questi oggetti (evita accoppiamento forte)

Nota: questa parte era originariamente approfondita dentro i Pattern Comportamentali. Decido di farne un approfondimento riprendendo quella parte e adattandola inserendo contenuto utile e dimostrativo.



Si usano modelli di aggiornamento in cui il Subject conosce i suoi Observer (push) oppure invia solo la notifica (pull).

Gli Observer si registrano su un particolare evento.

- I *Publisher* emettono eventi di interesse per gli altri oggetti. Questi eventi si verificano quando il *Publisher* cambia il suo stato o esegue alcuni comportamenti. I *Publisher* contengono un'infrastruttura di sottoscrizione che consente ai nuovi *Subscriber* di unirsi e agli attuali *Subscriber* di lasciare l'elenco.

- Quando si verifica un nuovo evento, il *Publisher* passa in rassegna l'elenco dei *Subscriber* e chiama il metodo di notifica dichiarato nell'interfaccia *Subscriber* su ogni oggetto *Subscriber*.

- L'interfaccia *Subscriber* dichiara l'interfaccia di notifica. Nella maggior parte dei casi, consiste in un singolo metodo di aggiornamento. Il metodo può avere diversi parametri che consentono all'editore di passare alcuni dettagli dell'evento insieme all'aggiornamento.

- I *Subscriber* concreti eseguono alcune azioni in risposta alle notifiche emesse dal *Publisher*. Tutte queste classi devono implementare la stessa interfaccia, in modo che il *Publisher* non sia accoppiato alle classi concrete.

- Di solito, i *Subscriber* hanno bisogno di alcune informazioni contestuali per gestire correttamente l'aggiornamento. Per questo motivo, gli editori spesso passano alcuni dati di contesto come argomenti del metodo di notifica. Il *Publisher* può passare sé stesso come argomento, lasciando che il *Subscriber* recuperi direttamente i dati richiesti.

- Il client crea separatamente gli oggetti Publisher e *Subscriber* e poi registra i *Subscriber* per gli aggiornamenti del *Publisher*.

Applicabilità e implementazione:
- Da usare quando le modifiche allo stato di un oggetto possono richiedere la modifica di altri oggetti, e l'insieme effettivo di oggetti non è noto in anticipo o cambia dinamicamente
- Quando alcuni oggetti dell'applicazione devono osservare altri, ma solo per un tempo limitato o in casi specifici
- Accoppiamento astratto tra Subject ed Observer (i primi non conoscono il tipo concreto dei secondi) e si possono aggiungere Observer liberamente (per osservare più Subject)
- Utilizzo di sistemi di lookup per gli osservatori (senza spreco di memoria)
- Evitare *dangling pointers*, notificando solo in stati consistenti

Ecco un semplice esempio del modello Observer implementato in Python utilizzando il modulo integrato *abc* (Abstract Base Class, che fornisce un modo standard di testare se un oggetto aderisce a specifiche precise, prevenendo di istanziare una sottoclasse che non fa l'override di un particolare metodo della superclasse; tutto ciò è implementato come modulo):

```
from abc import ABC, abstractmethod

# The subject interface
class Subject(ABC):
    @abstractmethod
    def attach(self, observer):
        pass

    @abstractmethod
    def detach(self, observer):
        pass

    @abstractmethod
    def notify(self):
        pass

# The observer interface
class Observer(ABC):
    @abstractmethod
    def update(self, subject):
        pass

# The concrete subject class
class ConcreteSubject(Subject):
    def __init__(self):
        self.observers = []
```

In questo esempio, l'interfaccia *Subject* definisce i metodi che il soggetto deve implementare per supportare la registrazione e la notifica degli osservatori, mentre l'interfaccia *Observer* definisce i metodi che gli osservatori devono implementare per ricevere gli aggiornamenti dal soggetto.

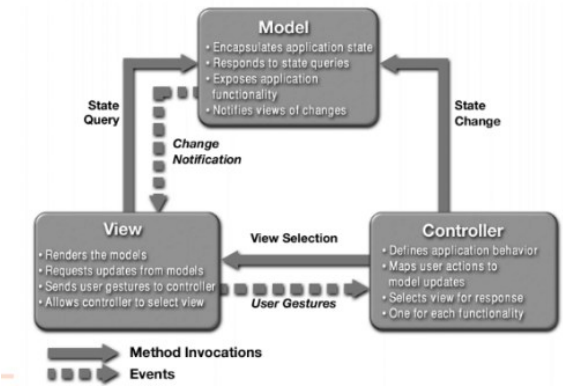
La classe *ConcreteSubject* è un'implementazione concreta dell'interfaccia *Subject*, mantiene un elenco di osservatori e fornisce metodi per collegarli, staccarli e notificarli. La classe *ConcreteObserver* è un'implementazione concreta dell'interfaccia *Observer* e fornisce un metodo di aggiornamento che viene richiamato dal soggetto quando il suo stato cambia.

Per utilizzare il pattern *Observer*, si crea un'istanza della classe *ConcreteSubject*, alla quale viene collegata un'istanza della classe *ConcreteObserver*. Quando lo stato del soggetto cambia, l'osservatore riceve una notifica e può reagire al cambiamento. In questo esempio, l'osservatore stampa semplicemente un messaggio in base allo stato dell'oggetto.

```
        self.state = None

    def attach(self, observer):
        self.observers.append(observer)

    def detach(self, observer):
        self.observers.remove(observer)

    def notify(self):
        for observer in self.observers:
            observer.update(self)

    def set_state(self, state):
        self.state = state
        self.notify()

# The concrete observer class
class ConcreteObserver(Observer):
    def update(self, subject):
        if subject.state == 'state1':
            print('State1 detected')
        elif subject.state == 'state2':
            print('State2 detected')

# Create a subject and an observer
subject = ConcreteSubject()
observer = ConcreteObserver()

# Attach the observer to the subject
subject.attach(observer)

# Change the state of the subject and notify the observer
subject.set_state('state1')
subject.set_state('state2')
```

Ora si scende nel dettaglio in merito alla *separation of concerns* che MVC realizza, ottenendo *disaccoppiamento*:

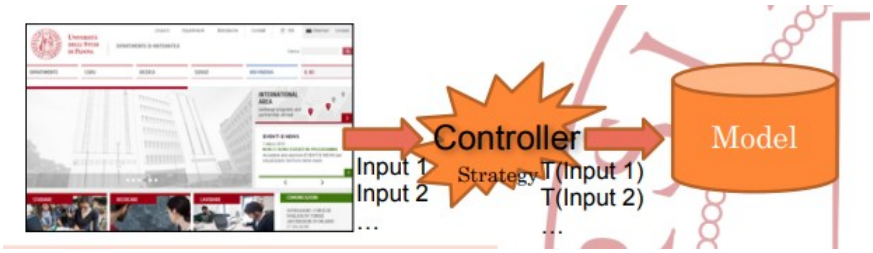
- *Model*: insieme di dati di business e regole di accesso (business logic). Esso definisce il modello dati, (tramite scambio dati/operazioni) con logiche orientate agli oggetti (*object oriented*). Esso notifica alla *view* aggiornamenti sui dati. Qui interviene l'*Observer Pattern*. Riceve query di modifica/cambiamenti di stato da View e Controller rispettivamente.



- *View*: rappresentazione grafica (presentation logic). Questa gestisce la logica di presentazione e i metodi di interazione con l'applicazione, catturando gli input user e *delegando* al controller l'elaborazione dei dati. Essa notifica le modifiche al Model e come evento al Controller le azioni utente (la View è Observer sul Model).

- Esistono due modelli d'uso delle view:
  - *push model*, in cui la vista deve essere *costantemente* aggiornata (usando l'*Observer pattern*). Normalmente questo approccio è adatto ad ambienti singoli di esecuzione, ad esempio quelli web based (single page application Javascript, in cui da solo è MVC)
  - *pull model*, in cui la vista richiede aggiornamento *solo quando è opportuno*. Questo approccio è utile quando si hanno diverse componenti che interagiscono tra loro (vari framework/servizi diversi che gestiscono MVC)

- *Controller*: reazione della UI agli input utente (application logic). Esso *trasforma* le interazioni dell'utente (view) in azioni sui dati (model) realizzando l'application logic. Il Controller è Observer sulla View, determinando cosa viene selezionato e da parte di questa viene notificato delle azioni utente. Trasmette poi il cambio di stato al Model, usando algoritmi che ne facilitano l'interazione.

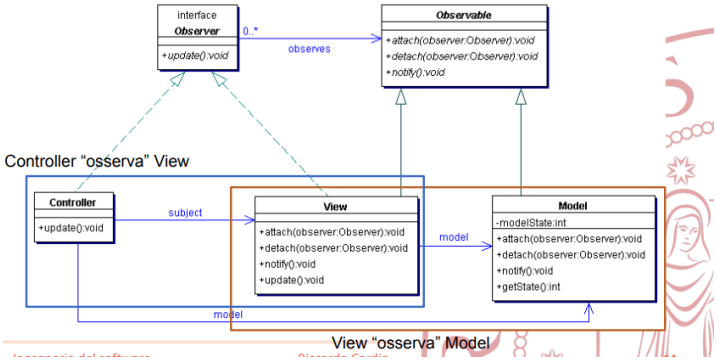


NB: Quasi sempre la relazione che c'è fra View e Model avviene attraverso l'utilizzo dell'Observer pattern; quanto è necessario cambiare il comportamento dell'applicazione a seconda delle circostanze, il controller usa anche lo Strategy.

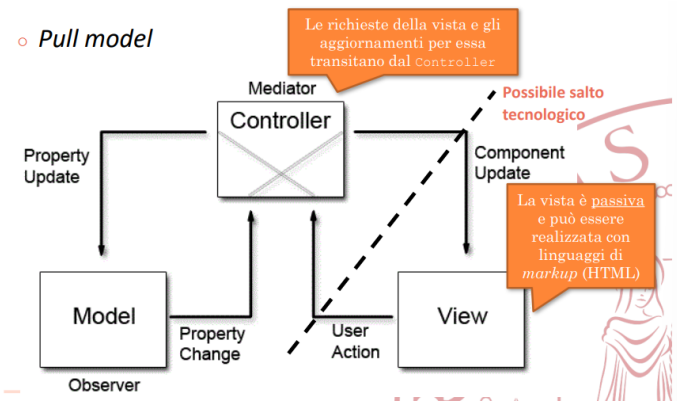
In generale quindi (seguono esempi grafici):

- Push model fa tutto da sé, in modo self-contained e fa da Observer/Subscriber e MVC a sé stessa
- Pull model, in cui si ha MVC classico ma la vista è *passiva*, in quanto trasmette solo aggiornamenti al Controller, che aggiorna costantemente il proprio Observer, quindi il model

- *Push model*



- *Pull model*



Conseguentemente:

- si ha il riutilizzo delle componenti del model; ciò migliora test e manutenzione
- si semplifica il supporto per vari tipi di client (tramite creazione nuova view/controller)
- vi è maggiore complessità di progettazione, dovuta all'introduzione di molte classi per garantire la separazione

Vediamo un esempio di MVC con Python:

```
from flask import Flask, render_template
app = Flask(__name__)

# The model
class Task:
    def __init__(self, description, completed=False):
        self.description = description
        self.completed = completed

 def __str__(self):
        return self.description

# The view
@app.route('/tasks')
def tasks():
    # Get the tasks from the model
    tasks = [Task('Finish this tutorial'), Task('Learn about MVC')]

    # Render the tasks template
    return render_template('tasks.html', tasks=tasks)

if __name__ == '__main__':
    app.run()
```

In questo esempio, la classe *Task* rappresenta il Model e contiene i dati di una singola attività. La funzione tasks rappresenta la View ed è responsabile del rendering di un modello che visualizza un elenco di task. Il template *tasks.html* rappresenta la View e contiene la logica di presentazione per la visualizzazione dei compiti. La funzione *tasks* agisce come Controller, in quanto recupera i task dal modello e li passa alla vista per renderli.

Per utilizzare lo schema MVC, il client effettua una richiesta alla *route /tasks*, che viene gestita dalla funzione *tasks*. La funzione *tasks* recupera i task dal Model e li passa al template *tasks.html*, che viene poi renderizzato e restituito al client come risposta.
Alcune considerazioni:

- Nativamente MVC si basa su *push model* nei casi web based (pagina web singola, caso JavaScript che fa da "one-man band" perché contemporaneamente MVC) oppure *pull model*, casi web-based server, con MVC separati, ad esempio rispettivamente con Hibernate come model (framework molto usato in Java), JSP/ASP come view e servlet come controller.

- Anche Spring realizza MVC, con classi *service* (business logic) come Model (scritte in JavaBean), un layer di visualizzazione/presentazione dati come view (file JSP) e un layer di gestione (dispatch) delle richieste e di controllo della logica applicativa come controller (sulle slide, vi sono due esempi on trattati di push e pull con MVC; questo blocco di testo sintetizza tutto ciò)

Oggi si va verso pattern perfezionati come la Model View Presenter (MVP), derivazione di MVC utilizzato per la costruzione di interfacce utente. La strutturazione è come segue:

- *Presenter* (*passive view*), che osserva passivamente il modello e semplicemente aggiorna la vista osservandola (*dumb*) tramite interfacce di comunicazione (*setter/getter*). Si comporta da via di scambio intermedia (*man in the middle*). In questo caso, la vista è business logic (*view business logic*).

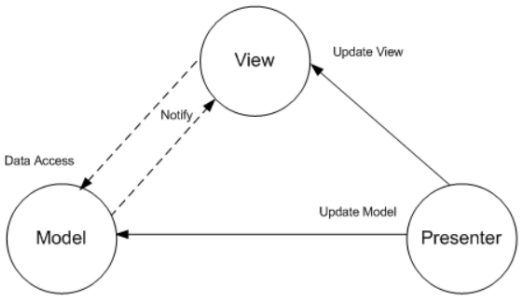


Figure 5: Simplistic MVP

- *View*, che si riduce ad un template di visualizzazione e ad *un'interfaccia* di comunicazione. Può essere sostituita da un mock in fase di test, in quanto è oggetto di riferimento ad un'interfaccia. Ha logica visuale e non ha logica applicativa solitamente.

- *Model*, che è un'interfaccia che definisce i dati da visualizzare o su cui agire in altro modo nell'interfaccia utente. Esso comunica attraverso le API con DB locale/con una cache (è indipendente dall'interfaccia utente e non contiene alcuna logica di presentazione)

In generale quindi:
- è più semplice scrivere i test di unità, data la specializzazione delle competenze tra gli oggetti
- mapping tramite interfacce tra View e Presenter, migliorando la gestione nel caso di View complesse
- aggiornamento simultaneo di Model e View e interazione unicamente tra Model e Presenter
- lo strato di presentazione è autonomo, grazie all'isolamento delle logiche di interazione

Mostriamo un esempio di MVP in Python (con a lato un diagramma delle classi):

```
class Model:
    def __init__(self):
        self.data = []

    def add_data(self, item):
        self.data.append(item)

    def get_data(self):
        return self.data

class View:
    def show_data(self, data):
        print(data)

    def get_input(self):
        return input("Enter data: ")

class Presenter:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def run(self):
        data = self.view.get_input()
```

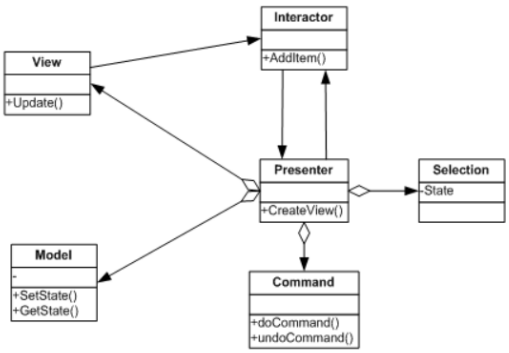


Figure 6: MVP

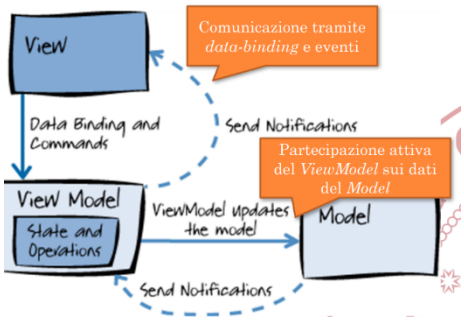
Scritto da Gabriel

```
    self.model.add_data(data)
    data = self.model.get_data()
    self.view.show_data(data)

model = Model()
view = View()
presenter = Presenter(model, view)
presenter.run()
```

In questo esempio, il *Model* rappresenta un elenco di dati, la *View* visualizza i dati all'utente e chiede all'utente di immetterli, mentre il *Presenter* recupera i dati dal *Model* e aggiorna la *View* con i dati. Il *Presenter* gestisce anche gli input dell'utente aggiungendoli al *Model* e aggiornando la *View* con i dati aggiornati.

Un esempio recente di pattern di implementazione popolarizzato da Angular JS è Model View Viewmodel, che separa lo sviluppo UI dalla business logic, affinché la View non sia dipendente da una specifica piattaforma. Questo modello è responsabile nel convertire ed esporre dati in modo facile e rappresentabile; in particolare la componente *viewmodel* gestisce tutta la logic. In generale:

- *ViewModel*, che realizza un binding diretto con vista e modello, in cui dati ed operazioni possono essere eseguiti su una UI. Quando si ha un cambiamento di stato, subito viene notificato ad un Model (*proiezione* del modello per una vista). Esso ha una sua logica interna: *quando cambio la vista, cambio anche il modello*, inviando notifiche alla view e aggiornando il model (*two-way data binding*)



- *View*, dichiarativa (usa linguaggi di markup) e un two-way data binding diretto con la ViewModel tramite appositi eventi/data binding. Essa è quindi *costantemente aggiornata* e non possiede più lo stato dell'applicazione, riceve solo notifiche.

- *Model*, che gestisce la logica del programma, ottenuta dal ViewModel tramite la ricezione dell'input utente con la vista. Letteralmente sono i dati che dobbiamo gestire. Esso ha un two-way data binding diretto con il ViewModel, inviando notifiche sullo stato dei suoi dati.

Vantaggi

- Facilita lo sviluppo parallelo di un'interfaccia utente e dei blocchi che la alimentano.
- Astrae la vista e riduce la quantità di logica di business necessaria nel codice dietro di essa.
- Il ViewModel può essere più facile da testare rispetto al codice event-driven.
- Il ViewModel (essendo più Modello che Vista) può essere testato senza preoccuparsi dell'automazione e dell'interazione con l'interfaccia utente.

Svantaggi

- Per le interfacce utente più semplici, MVVM può essere eccessivo.
- Sebbene i data binding possano essere dichiarativi e piacevoli da lavorare, possono essere più difficili da debuggare rispetto al codice imperativo, dove è sufficiente impostare dei breakpoint.
- I binding dei dati in applicazioni non banali possono creare una grande quantità di manutenzione necessaria. Inoltre, non vogliamo ritrovarci in una situazione in cui i binding sono più pesanti degli oggetti a cui sono legati.

Vediamo un esempio di MVVM in Python:

```
class Model:
    def __init__(self):
        self.data = []

    def add_data(self, item):
        self.data.append(item)

    def get_data(self):
        return self.data

class ViewModel:
    def __init__(self, model):
        self.model = model
        self.data = []

    def add_data(self, item):
        self.model.add_data(item)
        self.data = self.model.get_data()

    def get_data(self):
        return self.data

class View:
    def __init__(self, view_model):
        self.view_model = view_model

    def show_data(self, data):
        print(data)

    def get_input(self):
        return input("Enter data: ")

    def run(self):
        data = self.view_model.get_data()
        self.show_data(data)
        data = self.get_input()
        self.view_model.add_data(data)
        data = self.view_model.get_data()
        self.show_data(data)

model = Model()
view_model = ViewModel(model)
view = View(view_model)
view.run()
```

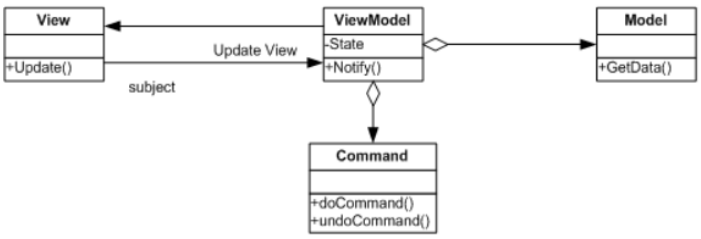


Figure 8: MVVM

In questo esempio, il *Model* rappresenta un elenco di dati, la *View* visualizza i dati all'utente e chiede all'utente di immetterli, mentre il *ViewModel* espone i dati e i comandi del *Model* alla *View* e gestisce gli input della *View* aggiornando il *Model* con le modifiche. Il *ViewModel* aggiorna anche i dati della *View* con i dati aggiornati del *Model*.

Diario di bordo: Verifica, parallelismo e proponente

I punti di discussione proposti sono:

- latenza proponente
    - Si definisce un mezzo utile di contatto che sia agile (es. breve meeting/semplice aggiornamento attraverso il canale), valorizzando il tempo proponente in modo asincrono. Se andiamo verso una modalità fruibile e immediata. Questo vale anche per l'interpretazione bisogni (user stories – casi d'uso), avviando uno studio da parte nostra e aggiornando in modo asincrono il proponente (forma di dialogo agile/agevole); es. piccolo video rispetto a cose da fare.
- parallelismo vs coordinamento (assegnazione compiti/ruoli)
    - Occorre fissare a priori un calendario e la suddivisione delle attività, non diventando eccessivamente parallela tra X persone e definendo prematuramente ruoli. Non è ragionevole prendersi un incarico e portarlo avanti esclusivamente da soli; tutti i compiti di questa fase richiedono discussione principalmente. Ciò richiede la giusta competenza, ma non è sufficientemente utile se non si ha consapevolezza del contesto d'uso. Si ritrova un equilibrio tra sincrono (fare tutto insieme), sminuzzando questo in sottogruppi dinamici di competenza e creatività (al fine di essere autonomi in modo asincrono).Questi variano per dimensione e numero, cercando di non essere monolitico (solo tutto il gruppo) oppure parallelo (ognuno all'estremo)
    - Il tempo di coordinamento di incontro deve essere breve ma non brevissimo (un periodo utile per l'avanzamento (entro la settimana, magari ogni X giorni) di cose nel backlog, esaminando il tutto con la retrospettiva. Questa aiuta la pianificazione, per vedere l'avanzamento effettivo in una cerimonia unica. Questo deve essere sempre visibile.
    - Evitare il parallelismo/monolitico → Sottogruppi dinamici, ben mescolati
- processo di verifica prodotti (cruscotto)
    - Normalmente si ha una gestione dei cambiamenti/change management che parte da piattaforme preesistenti, tali che si gestiscano le segnalazioni in ogni momento.
      In particolare, si definisce cosa fare e come verificarlo. All'origine ci sono i needs; attraverso un lavoro istruttorio, si individua cosa serve, rispondendo ai bisogni.
    - Si vuole rendere la verifica automatizzabile (proceduralizzabile), documentata nel Piano di Progetto, sapendo già a priori come intervenire. Questo potrebbe manifestarsi come approvazione da parte del proponente/stakeholder.
    - Basata su una checkbox che verifica univocamente il funzionamento → cruscotto
      Impensabile informare solo in un momento fisso dell'avanzamento tutto il gruppo (fatti e non opinioni). Serve uno strumento che definisce il calendario e che fa vedere subito tutto. Lo strumento di verifica è imparziale e la persona che ha definito la verifica è quella che decide per certo questa pratica.

Lezione rovesciata 1 – Documentazione (Vardanega)

La lezione rovesciata sulla documentazione cerca di rispondere almeno a queste tre domande:

1) Perché documentare e cosa documentare, per obblighi o per opportunità;
2) Con quali contenuti, tratti da dove;
3) Come produrre la documentazione necessaria in maniera efficace, per qualità informativa, ed efficiente, per impiego di risorse.

Essa permette di replicare il prodotto e comprenderlo senza il supporto umano, ma anzi, usandolo come mezzo di comunicazione, futura manutenzione, amministrazione e comprensione del prodotto.
Si consideri che la documentazione è processo *di supporto* all'attività di gestione di progetto, essendo un mezzo che, tramite versionamento e integrazione continua, norma attività di codifica, progettazione, analisi ed integrazione.

1) Viene messo a documentazione tutto ciò che può essere interessante a livello di vincoli (proponente/cliente), accordandosi per evitare ambiguità e definendo con precisione un dominio applicativo secondo il way of working.
In particolare, segniamo i motivi del *perché* documentare:

- Per assicurare che i processi produttivi si svolgano con la qualità attesa
- Per garantire circolazione di informazione completa, accurata, mirata, tempestiva
- Per facilitare controllo di avanzamento secondo il proprio modello di sviluppo
- Per segnare il confine tra creatività e disciplina

I cambiamenti sono frutto di un processo e si vuole capire la motivazione/storico delle scelte attuate (sistema di versionamento e change management). Questo processo porta alla creazione di uno standard di strutture ben definite, isolando caratteristiche deprecate (non più da usare) da quelle correnti (aggiornate), che consentano di risalire ai cambiamenti in modo semplice e di effettuarne qualora necessario.

Dato che il way of working evolve, *le modifiche devono essere implementate automaticamente*.
Per poter documentare i cambiamenti:

- si inserisce il *changelog* (storia dei cambiamenti) nel documento che ne indica lo stato delle modifiche, per meglio segnalarlo anche agli stessi stakeholders. È possibile automatizzarlo tramite il sistema di gestione delle modifiche, istituendo una procedura che dica ad ogni attore cosa viene modificato e perché
- ogni cambiamento è frutto di decisioni, tramite riunioni, documentate da apposito verbale (in esso sono riportate le decisioni e i motivi di cambiamento).

1-2) In particolare, individuiamo chiaramente *cosa* documentare:

- Piano di Progetto
  - Pianificazione a lungo periodo (macroscopica)
    - Scadenze-obiettivo, all'indietro
    - Analisi dei rischi
    - Preventivo dei costi
  - Pianificazione a breve (dettagliata)
    - Attività, in avanti
    - Preventivo minuto, alla luce del consuntivo di periodo precedente
    - Riscontro dei rischi, aggiornamento delle misure di mitigazione
- Piano di Qualifica
  - Obiettivi quantitativi di qualità
  - Cruscotto di misurazione
  - Analisi degli scostamenti e misure correttive

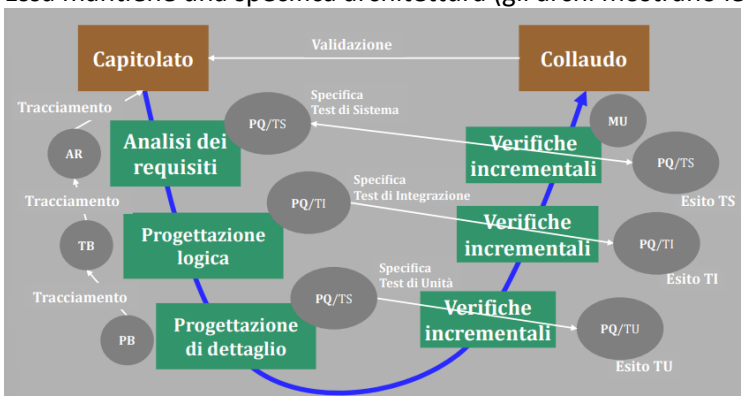
- Norme di Progetto (way of working)
    - Processi, procedure, strumenti
    - Metriche
- Analisi dei Requisiti
    - Casi d'uso
    - Tabelle di classificazione e di tracciamento
- Architettura logica e di dettaglio
    - In accompagnamento alla PB – Product Baseline
    - Si fornisce una descrizione gerarchica delle componenti del sistema (specifica software), la descrizione dell'architettura logica del sistema spiegando interfacce (funzione/input/output, etc.) in modo coerente (specifica tecnica)
    - Parliamo di *manuali* qui, cioè documenti che definiscono in senso trasversale (a livello aziendale) e ad alto livello come viene ricercata la qualità nell'organizzazione, con quali procedure, strategie ed obiettivi.
        - *Manuale Utente*
            - Nota come guida utente (*user guide*) o manuale d'uso (*user manual*), spiega agli utenti cosa faccia il prodotto, usando un linguaggio sempre e comprensibile
        - *Manuale Sviluppatore*
            - Anche noto come *Manuale del Manutentore*, ha lo scopo di fornire una linea guida per gli sviluppatori che andranno a mantenere o estendere il prodotto, dando informazioni relative a linguaggi e tecnologie utilizzate per la fase di realizzazione finale

3) In merito al *come* documentare:
- Lato produzione
    - Deve essere agile gestirne il ciclo di vita
    - Deve essere facile fare aggiornamenti e manutenzione migliorativa
- Lato consumo
    - Deve essere facile localizzare le modifiche
    - Deve contenere solo l'essenziale ed essere veloce da leggere

In documentazione software occorre scrivere *il meno possibile*.
Questo risparmia tempo al lettore e risorse adottate nel processo di scrittura, con un prodotto chiaro e più facilmente mantenibile nel corso del tempo. Concretamente, si possono usare template/automazione di termini inseriti nel glossario/strumenti di inserimento tabelle-indici-UML automatici (listo soluzioni adottate da gruppi nell'anno in corso del file, ndr).

Prima di scrivere, serve una *checklist* sull'indice dei contenuti (sapendo cosa deve essere scritto).
In generale, individuiamo *come* scrivere:
- Usare verbi in forma attiva
    - Per rendere esplicito il soggetto dell'azione
- Suddividere il contenuto in parti numerate e titolate
    - Per agevolarne la localizzazione
- Ricercare correttezza grammaticale e tipografica
- Usare frasi brevi e intorno a un solo argomento
- Preferire liste a elenchi narrativi
- Scrivere avendo chiaro il destinatario (committente/proponente/noi stessi)

Essa mantiene una specifica architettura (gli archi mostrano le relazioni biunivoche di tracciamento):



Più nello specifico, il tracciamento nella documentazione ha alcune basi:

- Le matrici di tracciabilità esplicitano la relazione di causalità tra i prodotti del processo di sviluppo
- Il tracciamento in avanti (*forward*) dimostra che non stiamo dimenticando nulla: *completezza*
    - Ogni nuova attività deve sapere esattamente cosa fare
- Il tracciamento all'indietro (backward) assicura che abbiamo fatto tutto il dovuto e nulla di superfluo: necessità
    - La verifica di fine attività deve saperne decidere l'esito

17/11/2022: Lezione rovesciata 2 – Metodi e strumenti per l'organizzazione dello sviluppo software

La lezione rovesciata sui metodi e strumenti per l'organizzazione dello sviluppo software cerca di rispondere almeno a queste tre domande:

1) Come assegnare azioni a persone e controllarne lo stato
2) Come verificarne il buon esito
3) Come riflettere questa informazione nel "cruscotto di valutazione" che guida le azioni di gestione di progetto.

1) Ci sono diversi metodi e strumenti che possono essere utilizzati per organizzare e seguire i progressi delle attività di sviluppo del software. Un metodo comune è l'utilizzo di uno strumento di gestione del progetto, come Trello, che consente ai project manager di assegnare compiti ai membri del team, di monitorare i progressi e di collaborare con i membri del team. Altri metodi includono l'uso di una board Kanban o di un diagramma di Gantt, che possono aiutare a visualizzare il lavoro da svolgere e le dipendenze tra le attività.

Utile considerare un ITS (Issue Tracking System) come GitHub oppure *Jira*.

Definiamo le terminologie utili:

- Una *roadmap* è una panoramica di alto livello del piano per un prodotto, un sistema o un componente. In genere include una tabella di marcia delle tappe e degli obiettivi principali e viene utilizzata per comunicare la visione e la direzione del progetto agli stakeholder.

- Le *istantanee* (*snapshots*) sono una funzione di Jira che consente agli utenti di scattare un'istantanea dello stato attuale di un progetto, compreso lo stato dei problemi e l'avanzamento del lavoro. Le istantanee possono essere utili per tracciare l'andamento di un progetto nel tempo e per identificare eventuali problemi da affrontare.

- Le *issue* in Jira sono attività o problemi che devono essere affrontati. Possono essere creati dagli utenti per tenere traccia del lavoro da svolgere o per segnalare i problemi da risolvere.
    - Utile in Jira la creazione di automatismi che eseguono esattamente tali compiti

- Le *boards* in Jira sono rappresentazioni visive del lavoro che deve essere svolto e sono tipicamente utilizzate per tenere traccia dell'avanzamento di un progetto. Esistono diversi tipi di schede in Jira, tra cui le schede Kanban, che consentono agli utenti di visualizzare il lavoro da svolgere e il suo avanzamento, e le schede Scrum, utilizzate per gestire progetti di sviluppo agile.

2) Per verificare il successo delle attività di sviluppo del software, è importante definire obiettivi e traguardi chiari e monitorare i progressi del lavoro verso tali obiettivi. Per misurare il successo del lavoro, si possono utilizzare metriche come le tempistiche del progetto, i budget e gli indicatori di qualità. Per valutare la qualità del lavoro e individuare eventuali problemi da risolvere, si possono utilizzare anche revisioni e ispezioni periodiche.

Partendo da una *checklist* predefinita, ha luogo il processo di verifica è basato su
(accettazione – commento – rifiuto) con un esito che non è noto a priori.
Normalmente, si evita il *polling* (continuo dover chiedere “come siamo messi”) mandando notifiche sia al responsabile della verifica che all’interessato.

3) Il cruscotto (*dashboard*) di avanzamento progettuale è automatizzato tramite meccanismi logici, individuando causa e conseguenze con una visione d’insieme; si ragiona quindi con una board, normalmente tramite Kanban (visualizzando tutto quanto serva in un certo momento) ma anche tramite Scrum (entro uno stesso sprint).

Anche il cruscotto ha una checklist specifica per assicurarsi che le attività siano complete:

- *Definition of Done/DoD*
    - É un insieme di criteri che devono essere soddisfatti prima che un prodotto, un sistema o un componente sia considerato completo. Il DoD viene tipicamente utilizzato per garantire che il prodotto soddisfi gli standard di qualità necessari e sia pronto per il rilascio.
- *Acceptance Criteria*
    - Sono i requisiti specifici che un prodotto, un sistema o un componente deve soddisfare per essere accettato dal cliente o dagli stakeholder. Questi criteri possono includere requisiti funzionali, come i compiti o le azioni specifiche che il prodotto deve essere in grado di eseguire, e requisiti non funzionali, come le prestazioni, l'affidabilità e l'usabilità.

Occorre avere nella dashboard uno strumento che permetta di identificare in modo chiaro attività e compiti. Le azioni sono un reticolo disposto partendo dalla *fine*, per obiettivi da raggiungere secondo le milestone. Per “visualizzare” tale reticolo, usiamo mezzi che già abbiamo definito chiaramente:

- Diagrammi di Gantt, per identificare attività, gerarchie, durata e tempi di completamento
- PERT, per costruire “reti” di compiti e legami precisi, identificando ordini/priorità/gerarchie/dipendenze e trovando il “cammino minimo” per arrivare a destinazione

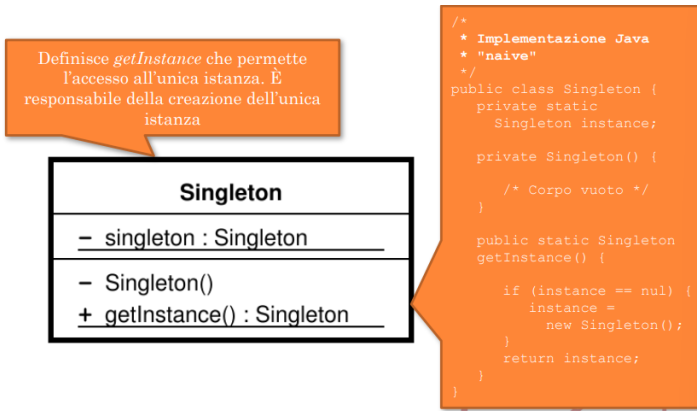
Design Pattern Creazionali (Cardin)

(Alcuni riferimenti:
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/singleton
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/abs/factory
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/builder)

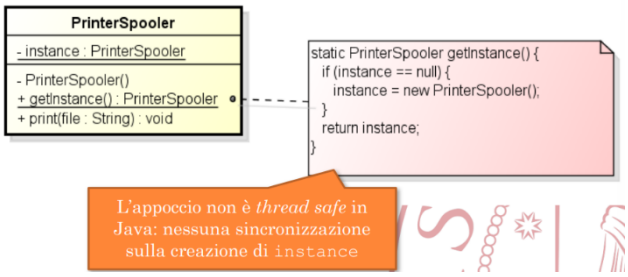
I design pattern creazionali servono a *rendere un sistema indipendente dall'implementazione concreta* delle sue componenti (nascondendo tipi concreti e dettagli di creazione/implementazione e *riducendo accoppiamento* e garantendo flessibilità). Tutto questo si realizza (chi lo avrebbe mai detto, ndr) grazie all'ampio uso di interfacce e di *astrazione*.

Partiamo dal Singleton Pattern, che assicura che *esista una sola istanza* di una classe, fornendo un punto di accesso *globale* a quest'ultima (si pensi ad esempio ad una connessione a un database, una richiesta di stampa), condivisa tra diverse parti di un'applicazione senza avere istanze multiple in quanto inefficiente/indesiderabile; occorre però tenere sempre traccia di questa istanza.
Questa istanza deve essere *estendibile* con ereditarietà (tale che i client non debbano modificare il codice).

Nell'esempio che si vede a lato, si vede che l'implementazione non è thread safe; non vi è la garanzia che venga creata una sola istanza della classe (perché non ci sarebbero *lock* sulle strutture sincronizzate o la parola *synchronized*, dato che ci sarebbe una race condition sulla creazione di *instance*), ma ne vengono create due.



- *Printer Spooler*



Un esempio concreto di implementazione non thread-safe con spooler di stampa a lato.

Pratiche dell'uso del Singleton:

- *Controllo* completo di *come* e *quando* i client *accedono* all'interfaccia (per i motivi precedenti)
- Evita il proliferare di variabili globali
- Permette la ridefinizione delle operazioni definite nel Singleton
- Può permettere un *numero massimo* e preciso di istanze attive
- Più flessibile delle operazioni di classe
    - Utilizzo del *polimorfismo*
- Occorre implementarlo in modo thread-safe quando si ha a che fare con ambienti condivisi

Implementazione:

- Assicurare un'unica istanza attiva (*lazy initialization*)
    - Si rende il/i costruttore/i privato/i (non accessibili)
    - Si rende disponibile un'operazione di classe di "recupero"

- Non è possibile utilizzare variabili globali (C++); gli oggetti sono a tutti condivisi
    - Non garantisce l'unicità dell'istanza
    - L'istanza è generata durante l'inizializzazione "statica"
    - Tutti i singleton sarebbero costruiti sempre, anche se mai utilizzati

Esistono vari modi di implementare un Singleton:

- Lazy initialization: In questo approccio, l'istanza della classe Singleton viene creata solo quando è necessaria per la prima volta. Può essere utile in situazioni in cui la creazione dell'istanza è costosa o richiede molto tempo.

- Eager initialization: In questo approccio, l'istanza della classe Singleton viene creata non appena la classe viene caricata. Può essere utile in situazioni in cui l'istanza deve essere creata e inizializzata il prima possibile.
- Ereditando da una classe Singleton
    - È difficile installare l'unica istanza nel membro *instance*; ecco perché tengo tracia dei Singleton tramite un registro (come si vede nell'esempio), in cui il Singleton d'utilizzo viene istanziato e registrato

```
public class Singleton {
   private static Singleton instance;
   private static Map<String, Singleton> registry;

   private Singleton() { /* Corpo vuoto */ }

   public static register(String name, Singleton)
   { /* ... */}

   protected static Singleton lookup(String name)
   { /* ... */}

   public static Singleton
   getInstance(String singletonName) {

      /* Utilizza lookup per recuperare l'istanza */
   }

}
```

```
public class MySingleton
   extends Singleton {

   static {

      Singleton.register("MySingleton",
         new MySingleton())
   }

   /* ... */
}
```

- Rendere il costruttore privato, facendo in modo che l'istanza sia *static final* e, quando debba essere istanziata, resa *volatile* (garantisce accesso singolo in memoria senza struttura di mutex).
    - Soffrono di attacco per *reflection* del costruttore (usato per ispezionare il comportamento a runtime di applicazioni senza conoscerne nomi/metodi/etc.)

- *Enum singleton*: In questo approccio, la classe Singleton è implementata come un'enumerazione, che garantisce l'esistenza di una sola istanza della classe. Questo approccio è semplice, non soffre di alcun attacco ed è ben leggibile, ma non è disponibile in tutti i linguaggi di programmazione.

```
public enum PrinterSpooler {
   INSTANCE;

   public void print(String file) { /* ... */ }
}
```

no *lazy*, *thread safe*, no *subclassing*, *serializable*

Essendo le Singleton classi *stateless*, è facile sovrascrivere più istanze dello stesso codice in modo non tracciato, realizzando *aliasing* con i dati esistenti, peraltro non garantendo la possibilità di testarli tutti.

Per questi motivi, viene considerato spesso un antipattern, in quanto:

- Il pattern Singleton può rendere difficile il test del codice che utilizza il singleton, poiché non è facile sostituirlo con un mock.
- Può rendere difficile modificare il codice che utilizza il singleton, poiché l'istanza singleton è tightly coupled/strettamente accoppiata al codice che la utilizza.

Il pattern *Telescoping* è un pattern di progettazione utilizzato per creare oggetti utilizzando una serie di costruttori con parametri che possono variare per numero e complessità a parità di costruzione.

Pro:

- Il pattern Telescoping è facile da capire e da implementare.
- Permette di creare oggetti con un gran numero di parametri opzionali, fornendo un costruttore separato per ogni serie di parametri opzionali.

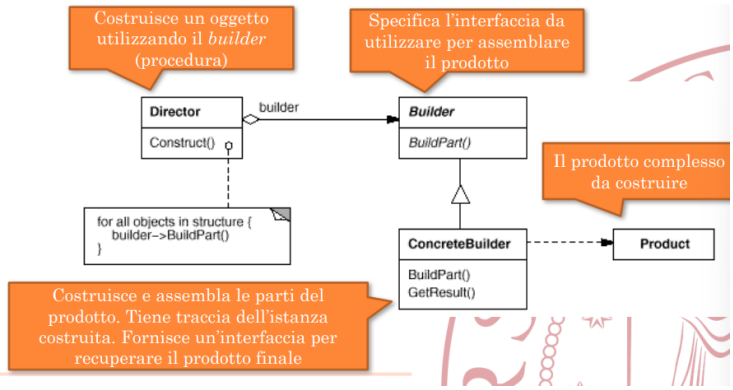
Contro:

- Il pattern Telescoping può generare un gran numero di costruttori, rendendo il codice difficile da mantenere e da leggere.
    - Il suo uso può essere rovinoso (antipattern)
- Può essere soggetto a errori, poiché si basa sul fatto che il chiamante fornisca l'insieme corretto di parametri nell'ordine corretto.
- Non è scalabile quando il numero di parametri opzionali aumenta.

Esempio reale di codice Telescoping:
car1 = Car("Ford", "Mustang")
car2 = Car("Ford", "Mustang", 2020)
car3 = Car("Ford", "Mustang", 2020, "red")
car4 = Car("Ford", "Mustang", 2020, "red", 30000)

Un altro pattern comune e che risolve il Telescoping è il Builder Pattern, che separa la costruzione di un oggetto complesso dalla sua rappresentazione. Consente la creazione di oggetti complessi attraverso l'uso di un processo di costruzione passo dopo passo.

È particolarmente utile quando il processo di costruzione di un oggetto (*indipendente* dalle parti che lo compongono) è complesso e l'oggetto deve essere creato in una serie di passaggi (in modo tale che, *con varie costruzioni possa ottenere diverse rappresentazioni*).



- L'interfaccia *Builder* dichiara le fasi di costruzione del prodotto che sono comuni a tutti i tipi di costruttori.

- I *Concrete Builder* forniscono diverse implementazioni delle fasi di costruzione. Essi possono produrre oggetti che non seguono l'interfaccia comune.

- I *Product* sono oggetti risultanti. I prodotti costruiti da costruttori diversi non devono necessariamente appartenere alla stessa gerarchia di classi o alla stessa interfaccia.

- La classe *Director* definisce l'ordine di chiamata delle fasi di costruzione, in modo da poter creare e riutilizzare configurazioni specifiche di prodotti.

- Il client deve associare uno degli oggetti costruttore al *Director*. Di solito lo si fa una sola volta, tramite i parametri del costruttore della direttrice. Poi la *Director* utilizza quell'oggetto costruttore per tutte le costruzioni successive. Tuttavia, esiste un approccio alternativo per quando il client passa l'oggetto costruttore al metodo di produzione della *Director*. In questo caso, è possibile utilizzare un costruttore diverso ogni volta che si produce qualcosa con la classe *Director*.

Di fatto, conoscendo il tipo concreto di riferimento, l'oggetto creatore notifica il Builder delle parti di cui ha bisogno e lo crea. Il client, tramite un metodo di recupero (*get()*), conoscerà il risultato.

Conseguenze:
- Facilita le *modifiche* alla rappresentazione interna di un prodotto e la costruzione di oggetti complessi
    - È sufficiente costruire un nuovo builder: questo evita il telescoping
- *Isola* il codice dedicato alla costruzione di un prodotto dalla sua rappresentazione
    - Il client non conosce le componenti interne di un prodotto (o i dettagli interni)

Scritto da Gabriel

    - L'orchestrazione dei processi di costruzione è unica
- Consente un *controllo migliore* del processo di costruzione
    - Costruzione step-by-step
    - Accentramento logica di validazione
    - A prescindere dall'oggetto, aggiungo vari parametri
- "*Fail fast*"
    - Questo aiuta a capire se sto costruendo oggetti troppo complicati aggiungendo codice non necessario; la costruzione, come nella realtà, è incrementale

Implementazione:

- Il builder definisce un'interfaccia per ogni parte che il Client può richiedere di costruire
    - Abbastanza generale per la costruzione di prodotti differenti
    - *Appending process* (mantiene le singole proprietà, creando un prodotto; quest'ultimo può avere un costruttore completo oppure una serie di setter)
- Nessuna classe astratta comune per i prodotti
    - Differiscono notevolmente fra loro
    - Se simili, valutare l'utilizzo di un *Abstract Factory Pattern*
- Fornire metodi *vuoti* come default
    - I builder concreti ridefiniscono solo i metodi necessari

Vediamo un esempio di Builder pattern in Python:

```
class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

    def __str__(self):
        return f'{self.year} {self.make} {self.model} ({self.color})'

class CarBuilder:
    def __init__(self):
        self.make = None
        self.model = None
        self.year = None
        self.color = None

 def set_make(self, make):
        self.make = make
        return self

    def set_model(self, model):
        self.model = model
        return self

    def set_year(self, year):
        self.year = year
        return self

    def set_color(self, color):
        self.color = color
```

Scritto da Gabriel

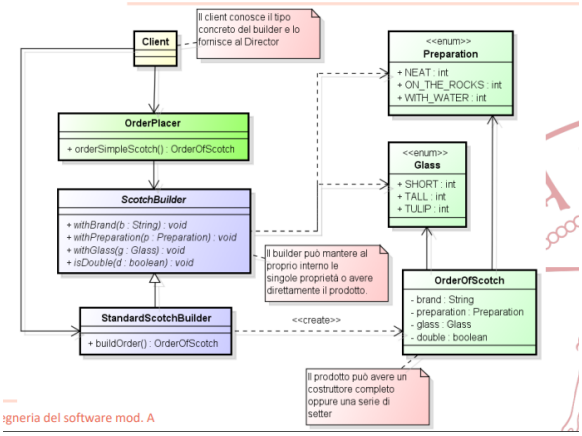
```
        return self

    def build(self):
        return Car(self.make, self.model, self.year, self.color)

builder = CarBuilder()
car = builder.set_make('Ford').set_model('Mustang').set_year(1969).set_color('Red').build()
print(car)
```

Questo codice definisce una classe *Car* e una classe *CarBuilder*. La classe *CarBuilder* ha metodi per impostare la marca, il modello, l'anno e il colore di un'auto e un metodo di costruzione che crea un nuovo oggetto Car utilizzando i valori specificati. L'oggetto *CarBuilder* può essere utilizzato per costruire un oggetto *Auto*, richiamando i suoi metodi in un processo graduale. Quando il metodo *build* viene richiamato, restituisce un nuovo oggetto *Car* con gli attributi specificati.

Vediamo un esempio dalle slide:

Si vuole ordinare un bicchiere di *scotch*. È necessario fornire al barman alcune informazioni: la marca del whiskey, come deve essere preparato (liscio, *on the rocks*, allungato) e se lo si desidera doppio. È inoltre possibile scegliere il tipo di bicchiere (piccolo, lungo, a tulipano).

[se fossimo veramente intenditori, anche la marca e la temperatura dell'acqua sarebbero fondamentali...]



Secondo questa implementazione Java, il builder diventa classe interna del prodotto:

```
public class OrderOfScotch {
   private String brand;
   private Praparation preparation;   // Si puo' dichiarare enum interna
   // ...
   private OrderOfScotch() { }   // Costruttore privato per il prodotto
   private OrderOfScotch(Builder builder) {
      this.brand = builder._brand;
      // ...
   }
   public static class Builder {
       private String _brand;   // Inserisco '_' per diversificare
       private Praparation _preparation;
       // ...
       public ScotchBuilder setBrand(String brand) {
          this._brand = brand;
          return this;   // Appending behaviour
       }
       // CONTINUA...
```

```
       // CONTINUA...
       public OrderOfScotch build() { return new OrderOfScotch(this); }
   }   // public static class Builder

   public static void main(String[] args) {
       // Il client non conosce il processo di costruzione e le classi
       // prodotto e builder sono correttamente accoppiate tra loro.
       // Non e' possibile costruire un prodotto se non con il builder.
       OrderOfScotch oos = new OrderOfScotch.Builder()
          .setBrand("Brand 1")
          .setPreparation(NEAT)
          // ...
          .build();
   }
}   // public class OrderOfScotch
```

Il Builder presenta alcuni problemi:

- Può portare a molto codice *boilerplate* (codice ripetitivo e non specifico ad un particolare compito, aggiunto esclusivamente perché serve [es. importazione librerie e costruzione oggetto in framework], poiché richiede la creazione di una classe costruttore separata per ogni oggetto che deve essere costruito.
- Può rendere il codice più difficile da leggere e da capire, perché il processo di costruzione è suddiviso in più metodi della classe costruttore.
- Può portare complessità non necessaria se devo costruire degli oggetti semplici

Immaginiamo di voler costruire un toolkit grafico, che dovrà avere diverse UI per più sistemi. Questo richiede codice differente, dovendo interagire con varie librerie grafiche. In questo caso serve costruire *famiglie* di prodotti (text-label, mouse, puntatori, etc) per ogni OS in modo coerente (non mischiando gli elementi).

Scritto da Gabriel

L'idea di coerenza di costruzione è proprio Abstract Factory, che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti, senza specificare le loro classi concrete e senza che il client debba conoscere i dettagli specifici dell'implementazione degli oggetti creati.

L'applicazione deve essere configurabile con diverse famiglie di componenti (toolkit grafico), definendo una classe astratta *factory* che definisce le interfacce di creazione (in cui i client non costruiscono direttamente i "prodotti"). Si definiscono le interfacce degli oggetti da creare e le classi che concretizzano *factory* vengono *costruite una volta sola*.

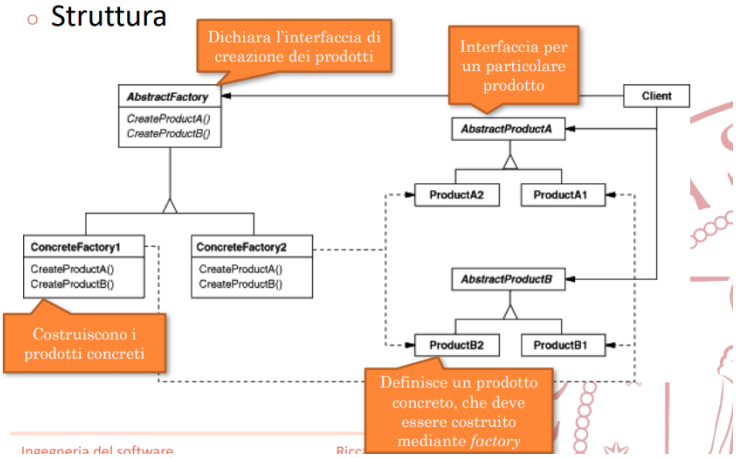
Alcune caratteristiche di base:

- Un sistema *indipendente* da come i *componenti* sono creati, composti e rappresentati
- Un sistema *configurabile* con più famiglie prodotti
- Le *componenti* di una famiglia devono essere *utilizzate insieme*
- Si vuole fornire una libreria di classi prodotto, senza rivelarne l'implementazione

Il client, quando ha bisogno di un'istanza si rivolge alla Factory; essa si occuperà di definire un'interfaccia di creazione generica, come tale in grado astrarre (quindi, rivolgendosi ad un particolare prodotto, a prescindere dal tipo) per costruire i prodotti *concreti*.

Ciò si nota brevemente nell'esempio che segue:

- Struttura



- Gli *Abstract Product* dichiarano le interfacce per un insieme di prodotti distinti ma correlati che costituiscono una famiglia di prodotti.

- I *Concrete Product* sono diverse implementazioni di prodotti astratti, raggruppati per varianti. Ogni prodotto astratto (es. sedia/divano) deve essere implementato in tutte le varianti date (nell'esempio di prima, e.g. vittoriana/moderna).

- L'interfaccia *Abstract Factory* dichiara un insieme di metodi per la creazione di ciascun prodotto astratto.

- Le *Concrete Factory* implementano i metodi di creazione della *Abstract Factory*. Ogni fabbrica concreta corrisponde a una specifica variante di prodotti e crea solo quelle varianti di prodotti.

Conseguenze (ipotizzando che di base tutti gli oggetti abbiano la stessa funzionalità):

- *Isolamento* dei tipi concreti
  - I client manipolano unicamente interfacce, i nomi dei prodotti sono nascosti
  - Non si dipende dalla Concrete Factory ma solo dalla interfaccia della Abstract Factory
- *Semplicità* maggiore nell'utilizzo di una diversa *famiglia di prodotti*
  - La Concrete Factory appare solo una volta nel programma
- Promuove la *consistenza* fra i prodotti

Scritto da Gabriel

- Difficoltà nel supportare nuovi prodotti
    - Modificare l'interfaccia della *factory* astratta costringe il cambiamento di tutte le sotto classi; ha poco senso usare il pattern quando gli oggetto non hanno temi/interfacce comuni
- Introduzione di molte nuovi classi e interfacce
    - Aumenta la complessità e la comprensione del codice

Implementazione:

- Solitamente si necessita di una sola istanza della factory (Singleton design pattern)
- Definizione di factory estendibili
    - Aggiungere un *parametro* ai metodi di *creazione* dei prodotti
        - Il parametro specifica il tipo di prodotto
        - Nei linguaggi tipizzati staticamente è possibile solo se tutti i prodotti condividono la stessa interfaccia
        - Può obbligare a down cast pericolosi

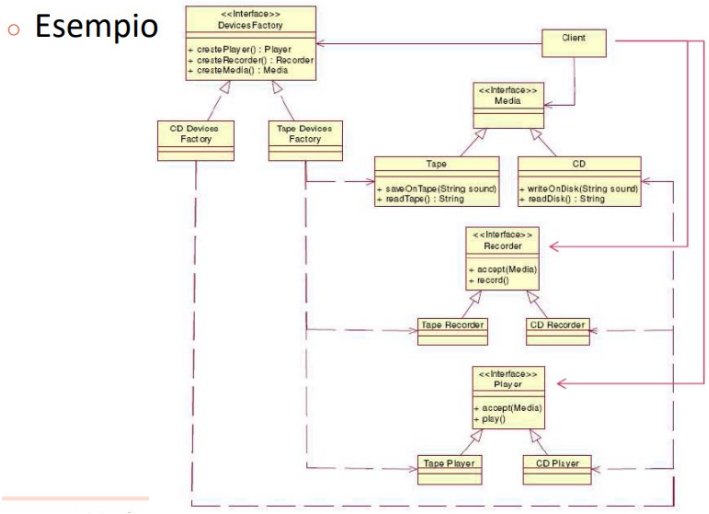
Un esempio del pattern dalle slide:

Esempio

Si vuole realizzare un negozio di vendita di sistemi Hi-Fi, dove si eseguono dimostrazioni dell'utilizzo dei prodotti.

Esistono due famiglie di prodotti, basate su tecnologie diverse:
- supporto di tipo nastro (*tape*)
- supporto di tipo digitale (CD).

Ogni famiglia è composta da:
- supporto (*tape* o CD)
- masterizzatore (*recorder*)
- riproduttore (*player*).



Un esempio di codice in Python:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

class AnimalFactory(ABC):
    @abstractmethod
    def create_animal(self):
        pass

class DogFactory(AnimalFactory):
    def create_animal(self):
```

Scritto da Gabriel

In questo esempio, la classe *Animal* è una classe astratta che definisce il metodo *speak* che tutte le classi animali concrete devono implementare. Le classi *Cat* e *Dog* sono implementazioni concrete della classe *Animal*.
La classe *AnimalFactory* è una classe astratta che definisce il metodo *create_animal*, utilizzato per creare oggetti della classe Animal. Le classi *DogFactory* e *CatFactory* sono implementazioni concrete della classe AnimalFactory che creano oggetti rispettivamente delle classi *Dog* e *Cat*.

Il codice client crea un'istanza del factory concreto appropriato (*DogFactory* o *CatFactory*) e lo usa per creare un oggetto della classe animale concreta corrispondente (*Dog* o *Cat*). Il codice client dipende solo dall'interfaccia astratta del factory e non dallo specifico factory concreto utilizzato, il che rende facile scambiare le implementazioni concrete degli oggetti creati.

```
        return Dog()

class CatFactory(AnimalFactory):
    def create_animal(self):
        return Cat()
dog_factory = DogFactory()
dog = dog_factory.create_animal()
print(dog.speak())  # prints "Woof!"

cat_factory = CatFactory()
cat = cat_factory.create_animal()
print(cat.speak())  # prints "Meow!"
```

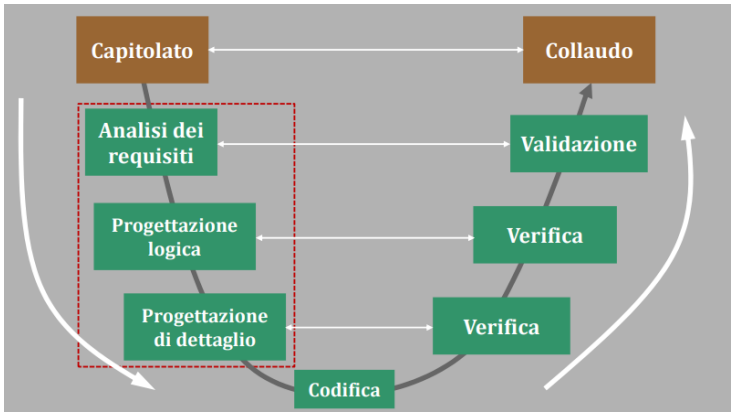
Progettazione Software (Vardanega)

(Eventuali approfondimenti:
https://www.math.unipd.it/~tullio/IS-1//2008/Approfondimenti/Christopher_Alexander_1999.pdf
https://www.math.unipd.it/~tullio/IS-1/2006/Approfondimenti/SEI-Software_Architectures.pdf
https://www.math.unipd.it/~tullio/IS-1/2004/Approfondimenti/Fan-in_Fan-out.html)

Una volta individuato in modo chiaro il problema tramite l'analisi dei requisiti(approccio analitico), il compito di trovare la soluzione per risolvere il problema nel modo più corretto è compito della progettazione software(approccio sintetico).

Il modello di progettazione software è il modello a V/V model, il quale è un processo di sviluppo del software utilizzato per rappresentare la relazione tra le varie fasi del ciclo di vita dello sviluppo del software (SDLC/Software Development Life Cycle) e i corrispondenti *deliverable* (risultati intermedi unici, misurabili e verificabili) prodotti in ciascuna fase e utilizzati come input per quella successiva.
Il ramo discendente produce le parti della soluzione (decomposizione), mentre il ramo ascendente compone le parti della soluzione nel prodotto finale (composizione).

Si ha corrispondenza tra le singole parti, seguendo i collegamenti in senso verticale/orizzontale quando ci serve. Lo stesso modo con cui vengono svolte queste azioni dipende dal modello adottato e come ottenere le singole responsabilità.



Il vantaggio di questo modello è più che altro la correlazione tra i singoli pezzi di prodotto, sapendo a cosa fare riferimento nel momento in cui modifichiamo dei pezzi e vogliamo invece implementare parti molto specifiche di prodotto; tuttavia, può essere abbastanza costoso temporalmente, non particolarmente efficiente e inflessibile nel caso di progetto con tanti cambiamenti di requisiti.

In questo caso, creiamo i requisiti soluzione con la progettazione e l'obiettivo è il soddisfacimento di tutti i requisiti con un sistema di qualità, ottenibile tramite la definizione di una buona architettura logica del prodotto che presenti componenti dalle specifiche chiare e coese, che sia realizzabile con risorse e costi fissati e che abbia una struttura che faciliti cambiamenti futuri(manutenibile).
La complessità di un sistema si domina scomponendo il problemi in pezzi dominabili, sia a livello requisiti che software, definendo *componenti*, raggruppanti *unità,* le quali sono composte da *moduli*

Scritto da Gabriel

Anche per questo motivo, nel corso di SWE si ha una revisione apposita, che dimostra, capiti i requisiti, di poter esprimere l'utilizzo di un certo numero di tecnologie del capitolato. Questa è la RTB (Requirements and Technology Baseline), per la quale serve un Proof of Concept (PoC), cioè una demo di prodotto completamente libera e con un numero di tecnologie da noi deciso e sul quale non ci sono vincoli in questo senso. La tecnologia è software che viene *addomesticato* per la soluzione, sulla base di quello che serve a noi. La RTB serve per poter avere un design, grazie alla comprensione e un minimo di utilizzo concreto delle tecnologie.

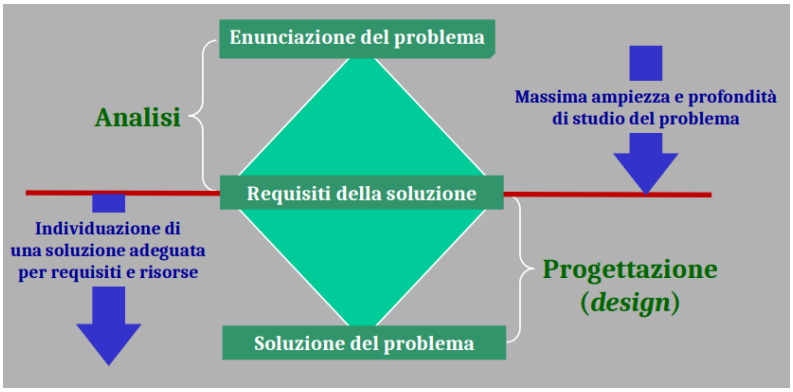
Graficamente nel V Model si colloca qui:

Lo stesso PoC è un oggetto usa e getta che serve per capire come poter realizzare la soluzione.
Infatti, nell'analisi dei requisiti, ci sta software esplorativo (quindi, non uguale al prodotto).

L'analisi dice per certo come fare la validazione sui requisiti, capendo se ognuno sia corretto oppure no.
Questo accade secondo una procedura algoritmica *automatizzabile* (con suddivisione certa tra parti).
La validazione attua le azioni di controllo che dice l'analisi dei requisiti.
Nel mondo reale, il collaudo viene deciso dal committente. A due passi di design corrispondono due passi di verifica; quest'ultima controlla se sto lavorando bene.

L'analisi cerca di comprendere ogni anfratto del problema (*ampliando* il raggio d'azione). Su un numero di requisiti ≤ 10, corrisponde un fattore di espansione molto più grande numericamente. La soluzione soddisfa all'inverso tutti i requisiti, arrivando al prodotto.

La codifica deve fare *solo le cose utili*, prendendo però in carico tutti i requisiti in modo tracciato e dimostrabile, scorrendo verso il basso (*scroll*) al punto utile.



Continuazione Progettazione Software – Standard e Design (Vardanega)

Il fatto di fare codice a regola d'arte indica che i passi precedenti siano *giusti*, perché ci si aspetta abbia un certo comportamento. Questi sono definiti come lato soluzione, mettendosi nei panni lato utilizzatore/attore; il PoC si pone il problema di *capire se le tecnologie scelte sono quelle giuste*, il cui valore è conoscitivo/esplorativo.

La codifica ha un significato di *trasformazione* in senso evolutivo del prodotto, dato che cambia e modifica l'espressione di come il codice viene ideato e creato. Essa *non può cambiare significato*, dato che si basa interamente sul design, dato che l'analisi dei requisiti dà i bisogni della soluzione.

La sua visione è puramente *esecutiva* rispetto a cose note e stabilite. Infatti:

- l'attività di analisi cerca di capire qual è il problema e decide la cosa giusta da fare, comprendendo il dominio e decidendo requisiti tecnici e funzionali;

- l'attività di progettazione ricerca una soluzione utile per tutti gli stakeholders, fissando la visione realizzativa delle cose utili, quindi l'*architettura*.

Secondo Dijkstra, la soddisfazione dei requisiti come singolo compito è composto da due parti:
1) indicare le proprietà di una cosa, in virtù delle quali essa soddisferebbe i nostri bisogni;
   - questo è responsabilità *dell'analisi*;
2) realizzare un oggetto che garantisca le proprietà dichiarate;
   - questo è responsabilità *della codifica*.

La progettazione (design) precede la codifica, perseguendo:
- *correttezza per costruzione* (*correctness by design*), ottenibile con la scrittura di codice chiaro e conciso, utilizzo di design pattern corretti, testing preciso e seguendo best practices;
  - ciò si persegue con l'uso di metodi di verifica sicuri (formali) sin dall'inizio dello sviluppo, sviluppando a piccoli passi e in modo corretto (agile);
- *correttezza per correzione* (*correctness by patch*), trovando errori o difetti e correggendo in corso d'opera in modo continuativo;
  - se un software ha tante pezze (*patches*), allora non è giusto in partenza, per *misconception* (idea sbagliata), addirittura portando ad errori clamorosi (*blunder*).

Arrivare ad un software per correzione è inutile; deve essere corretto in partenza.
A questo serve la progettazione (design), con una serie di responsabilità:
- dominare la complessità del prodotto ("divide-et-impera");
- organizzare e ripartire le responsabilità di realizzazione;
- produrre in economia (efficienza), quindi usare bene le risorse a disposizione;
- garantire qualità (efficacia), quindi aver usato bene le risorse a disposizione.

La costruzione di un software "giusto" parte dalla definizione di un'architettura, che è l'organizzazione di base di un sistema, espressa dai suoi componenti, dalle relazioni tra di loro e con l'ambiente, e i principi che ne guidano il progetto e l'evoluzione. Questa rappresenta il soddisfacimento dei requisiti, al fine di dominare la complessità del sistema scomponendolo in parti via via meno complesse, fermando la decomposizione quando l'attività è svolgibile da un singolo individuo (utile fermarla quando si riconosce sia troppo complesso il tutto).

In generale, possiamo dire che un'architettura software:
- individua parti componibili coerenti con i requisiti (SCUM) secondo specifica chiara e coesa;
- è realizzabile con risorse sostenibili e costi contenuti;
- sia organizzata in modo da facilitare cambiamenti futuri.

Lo standard *ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description* dettaglia una descrizione di un'architettura, dandone una vista di massima architetturale secondo i principi sotto: (https://cwe.ccsds.org/sea/docs/SEA-SA/Draft%20Documents/RASDS%20-%20Systems%20Architecture%20Background%20Materials/ISO-IEC-IEEE-42010-2011.pdf)
- decomposizione del sistema in parti componibili (componenti);
- organizzazione di tali componenti;
  - ruoli, responsabilità, interazioni (chi fa cosa e come);
- interfacce necessarie all'interazione tra le componenti tra loro e con l'ambiente di esecuzione;
  - come le componenti collaborano e interagiscono;
- paradigmi di composizione delle componenti;
  - regole, criteri, limiti, vincoli (anche a fini di manutenibilità).

Possiamo modularizzare logicamente le definizioni in questo modo:

- l'architettura è l'espressione tramite design di padroneggiare il sistema suddividendolo in parti di complessità individuale bassa, rendendo la codifica e verifica dei compiti rapidi:
  - le parti del software comunicano tramite *protocolli*, insiemi di regole che definiscono la distribuzione e composizione delle singole parti;
  - si spinge il design in profondità domina nel modo giusto la complessità del sistema;
- le componenti comunicano tramite interfacce (non tramite API), secondo un accordo definito a priori, guidando la codifica;
- la descrizione avviene attraverso diagrammi in linguaggio formale (UML).

Andare in profondità è compito della progettazione di dettaglio, in cui le singole parti sono chiamate *unità architetturali*, parti funzionali (o di responsabilità) ben definite, realizzabili da un singolo programmatore.
La costruzione del software è *modulare*, in cui si hanno vari moduli di codice/eseguibili con specifici ruoli organizzativi (nonché API/UI/interazioni tra altri sistemi e/o componenti in modo completo).
Le componenti hanno le unità che, a loro volta, sono composte di *moduli* di codice, a volte corrispondenti ad unità architetturali.

Il software deve essere *parlante*, perché il suo significato deve essere chiaro e comprensibile: l'architettura è una visione *oggettiva* del codice e ha un chiaro fine di utilizzo. Fare bene questa fase comporta capire come organizzare logicamente le componenti (architettura logica), guidandone *l'integrazione*. Per fare ciò, si richiede che la specifica di ogni unità architetturale sia ben documentata:

- perché la sua programmazione possa svolgersi in modo autonomo e disciplinato;
- deve assicurare il tracciamento di requisiti da e verso ogni singola unità.

Concretamente, la progettazione di dettaglio si occupa di assegnare le unità a componenti per organizzare il lavoro di programmazione, producendo della documentazione che possa disciplinarla, tracciando i requisiti alle unità e definendo le configurazioni ammissibili del sistema. Inoltre vengono definiti gli strumenti per eseguire le prove di unità, secondo opportune modularità dei livelli di test (definiti nel Piano di Qualifica, con successivi piani per la validazione).

Conclusione Progettazione Software – Approcci/Architetture/Parametri di qualità/Metriche (Vardanega)

L'architettura può essere automatizzata sulla base di una serie di requisiti ottenuti da una decomposizione del problema, riassumibile tramite una *checklist*. Essa esprime una serie di caratteristiche del contesto che si analizza, descrivendo cosa si deve controllare al fine di poter considerare "corretto" quanto fatto.
Ogni oggetto deve avere uno scopo e una specializzazione, rimanendo *generali*, dunque corrette e complete per problemi similari al contesto d'uso ma astraendo dall'implementazione sottostante.

Ci sono vari modi di progettare/fare design, spesso basati su *framework* (cornice di lavoro), che sono insiemi integrati di componenti software prefabbricate(simili ad una libreria) che forniscono una base facilmente riusabile per una grande varietà di applicazioni in un dato dominio. Quindi si ha:

- approccio *top-down* (*stepwise refinement*), che considera una decomposizione funzionale partendo dai dettagli ad alto livello (che devono essere stabiliti e ben compresi), andando poi ad implementare le parti a basso livello. Questo può essere adattato in modo incrementale e specializzabile, basato su *framework*;

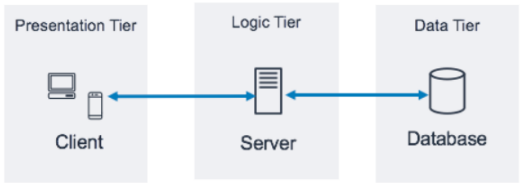
- approccio *bottom-up*, che concepisce il sistema ipotizzandone le parti che lo costituiscono e si attua una composizione funzionale:
  - le funzioni aggiuntive che si vanno a creare hanno proprietà fisse che non cambiano;

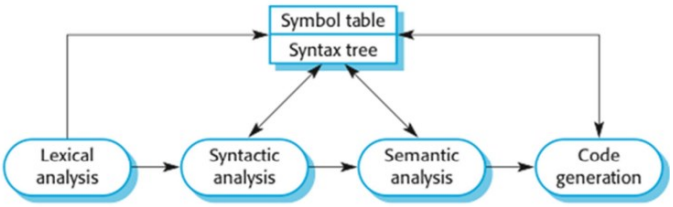
- ciò dà luogo all'utilizzo di funzionalità che possono essere messe insieme in vari modi, andando anche a creare sistemi complessi;
- questo richiede che i requisiti a basso livello siano stati ben compresi.

Il design usa stili architetturali, i quali forniscono una soluzione progettuale di alto livello, specificando poi come debba essere fatta la progettazione di dettaglio mediante l'uso di design pattern appropriati.
Ciascuno di questi è un aggregato coerente; sotto vediamo vari esempi utili:

- Catalogo di componenti standard (ricorrenti)
- Regole che vincolano la composizione di tali componenti tra loro
- Significato semantico di tali composizioni
- Catalogo di verifiche possibili di conformità su sistemi costruiti in tal modo

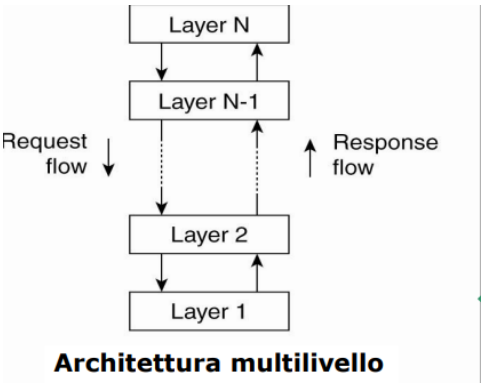
Un normale approccio è *separare* le parti del sistema, tra ciò che viene presentato, usato e memorizzato. Questo è l'approccio top-down, scomponibile in parti. Una sua realizzazione è l'architettura *Three-Tier* visibile a lato.

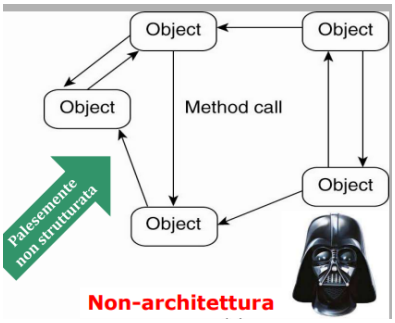




L'architettura *Pipe & Filter* (convoglia e filtra) è un altro pattern architetturale, che prevede entità indipendenti chiamate *filtri* (componenti) che eseguono trasformazioni sui dati ed elaborano l'input che ricevono, e *pipe*, che fungono da connettori per il flusso di dati da trasformare, come una catena di montaggio (pipeline).

L'architettura delle applicazioni multilivello fornisce un modello che consente agli sviluppatori di creare applicazioni flessibili e riutilizzabili, poiché vi è la possibilità di modificare o aggiungere un livello specifico, invece di rielaborare l'intera applicazione.





L'aderenza a stili preesistenti, spesso utili per quel problema e per quel contesto, garantisce questo principio. I pattern *portano vantaggi noti, a costo di rispettarli*.
L'esempio qui riportato non è chiaramente un'architettura, in quanto manca di qualsiasi strutturazione tra le parti, ma mette insieme chiamate tra moduli e basta in modo casuale e disorganizzato.

I design patterns (i nostri di riferimento sono quelli della Gang of Four/GoF) sono soluzioni progettuali a problemi realizzativi correnti, con un'organizzazione architetturale con proprietà provate, ottenibili solo con buona contestualizzazione e coerente implementazione.
Come dice giustamente Cardin, a inizio carriera si tende a vedere pattern ovunque (*design pattern euphoria*), ma avanzando, si comprende che i pattern modellano una cosa molto specifica con dei contratti; quindi, devono essere utilizzati in modo cosciente e se il contesto lo richiede effettivamente.

Una buona architettura viene definita come tale quando possiede una serie di qualità (sottolineando quelle che meritano maggiore considerazione e perché evidenziate nelle slide di riferimento):

- Sufficienza
  - Capace di soddisfare tutti i requisiti
- Comprensibilità
  - Capita da tutti gli stakeholder

- <u>Modularità</u>
  - Suddivisa in parti chiare e ben distinte, testabili e mantenibili in modo indipendente
  - Minimizza la dipendenza tra parti, creando una relazione buona e desiderabile
    - Determinando ciò che la parte deve esporre ai suoi utenti (l'interfaccia) e ciò che essa deve nascondere (l'implementazione)
    - Occorre determinare in maniera coesa le specifiche competenze delle parti software, facilitandone la manutenzione
  - Evitando rischio di *effetto domino*
    - Controllo specializzato degli accessi
    - Quando la modifica interna di una parte comporta modifiche all'esterno di sé
  - Vi sono due vie per modularizzare
    - Suddividere l'attività nei suoi blocchi logici principali, come una pipeline
    - Suddividere ricercando *information hiding*, mantenendo separate implementazione ed utilizzo; contribuisce alla longevità del software (*long lived*)
  - Ricercare modularità spinge a decomporre sempre di più
  - Esempi reali di modularità:
    - Uso di incapsulazione, astrazione, interfacce, dependency injection

- Robustezza
  - Capace di sopportare ingressi diversi (qualità/quantità) dall'utente e dall'ambiente
- Flessibilità
  - Permette modifiche a costo contenuto al variare dei requisiti
- Riusabilità
  - Le sue parti possono essere impiegate in altre applicazioni
- Efficienza
  - Nel tempo (CPU), nello spazio (RAM), nelle comunicazioni (banda)
- Affidabilità (reliability)
  - È probabile che svolga bene il suo compito quando utilizzata

- <u>Disponibilità</u> (availability)
  - La sua manutenzione causa poca indisponibilità totale, quindi rimanere usabile
  - Se eccessivamente pesante, può essere monolitico. Un sistema monolitico va ricostituito per intero a ogni piccolo cambiamento (modifica, aggiunta, rimozione), e poi il vecchio va sostituito dal nuovo.
  - Durante la sostituzione e le verifiche di buon esito, il sistema diventa indisponibile

- Sicurezza rispetto a malfunzionamenti (safety)
  - Abbastanza ridondante da funzionare anche in presenza di guasti
- Sicurezza rispetto a intrusioni (security)
  - I suoi dati e le sue funzioni non sono raggiungibili da intrusi

- <u>Semplicità</u>
  - Ogni parte contiene solo il necessario e niente di superfluo, sempre spiegabile facilmente
    - Semplifica la manutenzione e serve a trovare una soluzione immediata a problemi, anche complessi, a prescindere dal confronto

- Incapsulazione (*encapsulation - information hiding*)
  - L'interno delle componenti non è visibile dall'esterno
  - Rendere invisibile all'esterno l'interno delle componenti architetturali (*black box*)
  - Esporre solo l'interfaccia, nascondendo gli algoritmi e le strutture dati usate per realizzarla
  - Benefici:
    - L'esterno non può fare assunzioni sull'interno
    - Diventa più facile fare manutenzione sull'implementazione senza danneggiare gli utenti
    - Minori le dipendenze indotte sull'esterno, maggiore la libertà di cambiamento sul codice senza rompere il software preesistente e la modularità (esponendo *solo quello che serve* in quel contesto), aumentando il potenziale riuso

- Coesione
  - Ciò che sta insieme concorre agli stessi obiettivi, cioè parti che stanno bene insieme per scopo ben definito; in altri termini, tutti gli elementi permettono di raggiungere lo scopo/obiettivo definito inizialmente
  - Segue i principi SOLID (approfonditi a dovere nell'ultima lezione teorica di Cardin)
  - Funzionalità "vicine" stanno nella stessa componente, quindi tutte servono al contesto
    - Ciò che serve per soddisfare il contratto di interfaccia
  - Va massimizzata per ottenere
    - Maggiore manutenibilità e riusabilità
    - Minore interdipendenza fra componenti
    - Architettura del sistema più comprensibile

    La ricerca di coesione limita la spinta a decomporre sempre di più
  - Esistono vari tipi di coesione considerata buona (la migliore è quella che produce maggiore incapsulazione [information hiding]):
    - Funzionale: quando le parti concorrono allo stesso compito nel corso del tempo
      - Esempio: suddivisione in ruoli (come produttore/consumatore)
    - Temporale: quando alcune azioni sono «vicine» ad altre per ordine di esecuzione
      - Esempio: pipeline
    - Informativa: quando le parti agiscono sulle stesse unità dati
      - Esempio: *get()* e *set()* su una struttura dati
    - Esempio sbagliato: SIAGAS, piattaforma UniPD di gestione stage
      - Molte parti del suo codice realizzano funzioni simili:
        - Manutenzione complicata (una correzione locale non sana tutte le occorrenze del problema)
        - Rimedi possibili: Coesione *buona*, incapsulazione

- Basso accoppiamento (coupling)
  - Parti distinte dipendono poco o niente le une dalle altre
    - Esse stanno insieme in modo organizzato in cui ognuna ha un compito e, nel complesso, ha bisogno di ciascuna di queste
  - Quando parti diverse hanno dipendenze reciproche cattive
    - L'esterno fa assunzioni sul funzionamento dell'interno (variabili, tipi, indirizzi, …)
    - L'esterno impone vincoli sull'interno (ordine di azioni, uso di dati, formati, valori)
    - Esterno e interno agiscono su alias della stessa entità
  - Questo accoppiamento va minimizzato, cioè cose/oggetti che devo riuscire a controllare
  - Questo viene misurato tramite *metriche*
    - Proprietà esterna di componenti
      - Il grado U di utilizzo reciproco di M componenti
      - $U = M * M$ è il massimo grado di accoppiamento
      - $U = \emptyset$ ne è il minimo

- Metriche: fan-in e fan-out strutturale
  - SFIN (Structural Fan-In)
    - Numero di funzioni che chiamano la funzione attuale
    - È indice di utilità ed occorre massimizzarlo
      - Più alta questa metrica, maggiore il riuso
      - Dovrebbe essere un numero ragionevole, evitando *dead code* (codice mai chiamato e/o mai percorso nell'esecuzione)
      - La buona progettazione produce componenti con SFIN elevato
  - SFOUT (Structural Fan-Out)
    - Numero di funzioni che quella specifica funzione chiama
    - È indice di dipendenza ed occorre minimizzarlo
      - Più alta questa metrica, maggiore l'accoppiamento

Gli stati di progresso con cui poter valutare il grado di definizione ed implementazione di un'architettura sono secondo SEMAT i seguenti:

- *Architecture selected* (Milestone)
  - Selezione di una architettura tecnicamente adatta al problema e delle tecnologie necessarie
  - Decisioni su *buy, build, make*

- *Demonstrable* (Utile per la RTB, revisione obbligatoria di SWE – Requirements & Technology Baseline)
  - Dimostrazione delle principali caratteristiche dell'architettura: gli stakeholder concordano
  - Decisione sulle principali interfacce e configurazioni di sistema

- *Usable* (Utile per la PB, revisione obbligatoria di SWE – Product Baseline)
  - Il sistema è utilizzabile e ha le caratteristiche desiderate
  - Il sistema può essere operato dagli utenti
  - Le funzionalità e le prestazioni richieste sono state verificate e validate
  - La quantità di difetti residui è accettabile

- *Ready* (Utile per la CA, revisione opzionale di SWE – Customer Acceptance)
  - La documentazione per l'utente è pronta
  - Gli stakeholder hanno accettato il prodotto e vogliono che diventi operativo
  - La PB si colloca tra quest'ultima e la precedente

Diario di bordo: Rotazione ruoli, PDCA, PoC

Dubbi individuati:

- Interpretazione PDCA

Idealmente, noi agiamo con il PDCA (Plan Do Check Act), agisce secondo l'idea di miglioramento continuo. Normalmente, ciò che concorre a questo sono i "processi di supporto", cosa che è anche la documentazione stessa. In questi individuiamo:

- attività primariamente migliorabili (plan), cose già normate che iniettano miglioramenti ad attività da rifare in quanto con debolezze.
- usare quanto si pensa possa essere utile per migliorare, dato dalla pianificazione (do)
- controlla se l'esito migliorativo è arrivato (check)

Scritto da Gabriel

- attuazione del miglioramento sulla base di quello che funziona (act)

Si osserva da fuori e si cerca di migliorare.

- Proof of Concept (PoC)

Si parte dal concetto ideale (concept), che rappresenta una visione del prodotto e, come tale, sua effettivamente fattibile (*the proof is in the pudding*, tastando con mano). Questo comporta l'unione di varie componenti, in particolar modo legato alle tecnologie, usando sulla base dell'analisi ciò che il prodotto può fare (rispondendo a domande). Si dimostra sia la fattibilità concettuale che quella tecnologica, non essendo quindi baseline di prodotto. Nel PoC usiamo le tecnologie al meglio, limitando le funzionalità e operando una cernita, scegliendo quelle che possono andare meglio.
Si può fare una proposta di PoC al proponente, dando un'idea di quello che può essere il prodotto finale.

- "Comfort zone"

La rotazione dei ruoli è fondamentale per alternare le competenze e i compiti, tale da far acquisire dimestichezza con tutte le attività, sulla base della pianificazione temporale. Normale che durante uno sprint si possano assumere più ruoli ma, in un certo momento, ognuno assume un ruolo preciso.

- Orizzonte di pianificazione

La pianificazione dei compiti ragiona sulle attività e sull'impiego orario; possiamo distinguere:
- quella nel lungo periodo (a grana grossa), che difficilmente comprende le ore impiegate
- quella nel breve periodo (a grana fine), sulla base di uno sprint

Partendo da uno sprint, capiamo direttamente le ore impiegate/consumate; guardare oltre è il *preventivo a finire*, quindi il bacino d'ore sulla base delle disponibilità. Andando avanti con la *retrospettiva* del periodo, ci si rende conto di cosa fare e di cosa è stato fatto, migliorando la pianificazione.

Le differenze operative richiedono una pianificazione a lungo termine, mentre la pianificazione di dettaglio serve ad aggiustare sulla base della retrospettiva.
In questo modo, è possibile aggiornare ruoli/disponibilità sulla base delle azioni compiute e capacità orarie per farlo. La capacità di prevedere si affina col tempo, affermando cosa sarà fatto non sulla base del periodo ma degli obiettivi (avanzando a breve, cioè che sia significativo sulla base del contesto).
Il tempo deve essere congruo rispetto agli obiettivi, dando un orizzonte temporale prudente.

In quel momento, si hanno obiettivi a calendario; o viene raggiunto o "suona la sveglia". La retrospettiva fa capire gli impedimenti/problemi, inseguendo gli obiettivi in maniera conservativa, non ottimistica, vicina alla realtà. In questo modo, al posto di acquisire le certezze, le perdiamo.

- Misurare stato di avanzamento verso RTB

Adottiamo una checklist, quindi un elenco delle cose funzionanti/da fare per la revisione. Viene fatta partendo dalle regole, confrontandola con l'avanzamento rispetto agli obiettivi di periodo. In questo modo, accorciamo la distanza rispetto alla RTB. È ragionevole automatizzare il processo di ideazione documenti/creazione del PoC, capendo chi deve fare cosa rispetto a un elenco di obiettivi da raggiungere, smarcando gli obiettivi singolarmente.

Pit stop: Analisi dei requisiti "per davvero" (Vardanega)

Le attività più difficili dello sviluppo software sono l'analisi dei requisiti e la progettazione, dato che vengono fatte prima ancora di scrivere codice e quindi scaturire un effetto visibile.
Partiamo dalla distinzione di compiti e funzionalità, in cui si percepisce la prospettiva utente, cioè cosa il software deve fare (needs). Il compito degli sviluppatori è tradurre cosa il software dovrebbe fare in concrete possibilità; questo implica che si pensa alle funzionalità (scelta di sviluppo per permette di fare una certa cosa) e non ai compiti.

Per esempio, il computer è stato pensato lato funzionalità, dato che è stato deciso a priori che fosse fatto in quel modo (come macchina da scrivere). Occorre mettersi *nelle scarpe/in the shoes* dell'utilizzatore (come attore), partendo dai bisogni dell'utente, non inventandosi una soluzione e provando a darla.
La descrizione testuale facilita lo spostamento dell'attenzione rispetto a cosa (what) devo fare, nel senso di domanda aperta come esigenza e operando in senso generale senza scelta, a come (how) devo farla, facendo un richiamo a come realizzare il what. Per far emergere i compiti che l'utente deve avere (funzionalità, privilegi, norme), si scopre facendo/usando il software direttamente, con *requisiti impliciti*.

Si intende differenziare tra contenuto e contenitore: il primo è valore aggiunto e sta da solo indipendentemente come funzionalità, il secondo raccoglie concettualmente vari contenuti perché ne rappresenta un'associazione logica/di contenuto e mi fa capire *se il contenuto abbia senso*.
A questo concetto, si possono far corrispondere la risoluzione di casi d'uso complessi; per poterli fare, occorre risolvere varie cose, dipendenti dal problema o dall'ambiente in sé. Questi sono scomponibili in un insieme di azioni collegate logicamente/gerarchicamente, organizzabili concettualmente.

L'attore è la *manifestazione* dei bisogni, non è una semplice persona; esprime cosa deve essere fatto in quel momento. Per esempio Word che salva un file e diventa attore quando lo memorizza nel filesystem. In generale, pensiamo alle conseguenze logiche; se c'è un contenuto, esiste un contenitore (per esempio, inserimento/salvataggio, etc.), definendo le funzionalità. Se viene posto un vincolo sull'attore da parte del mediatore/uno stakeholder, questo si può vedere come scenario nel quale non esiste la possibilità X, individuando i compiti proponibili. Si capisce quindi che si fissano scenari e, dentro a quello, si individuano un certo numero di azioni possibili.

Si definisce il *Principio del Minimo Privilegio/Principle of Least Privilege*, definendo le condizioni iniziali di un sistema; l'utente passa attraverso fasi, acquisendo gradualmente privilegi, con situazioni molto specifiche/controllate. *Non tutti possono fare tutto*, limitando il privilegio all'attore di quello scenario. Attraverso azioni, si acquistano ulteriori privilegi. Questo si determina tramite l'individuazione di scenari e le relative gerarchie di attori, cambiando progressivamente la prospettiva sul "what".

Design Pattern Strutturali (Cardin)

(Alcuni riferimenti di codice:
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/decorator
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns/proxy)

Oggi introduciamo i design pattern strutturali che cercano di:

- affrontare problemi che riguardano la *composizione* di classi e oggetti (creazione di oggetti per fornire nuove funzionalità)
- consentire il *ri-utilizzo* degli oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta (attraverso integrazioni fra librerie / componenti diverse)
- sfruttano *l'ereditarietà* e *l'aggregazione*
- identificare in modo chiaro le *relazioni* tra oggetti (al fine di formare strutture più grandi)

Partiamo considerando l'Adapter Pattern, il cui scopo è *convertire* l'interfaccia di una classe in un'altra (è anche nota come *wrapper*); questo permette anche di fornire funzionalità che la classe adattata non possiede. Viene spesso usata per permettere a classi esistenti di lavorare con altre senza modificare il codice sorgente (es. un convertitore del DOM di XML per altri scopi)
Si cerca di dipendere *il meno possibile* dalla libreria esterna, usando una classe che *adatti le interfacce* (per ereditarietà o per composizione), dato che spesso i *toolkit* non sono riusabili/modificabili.

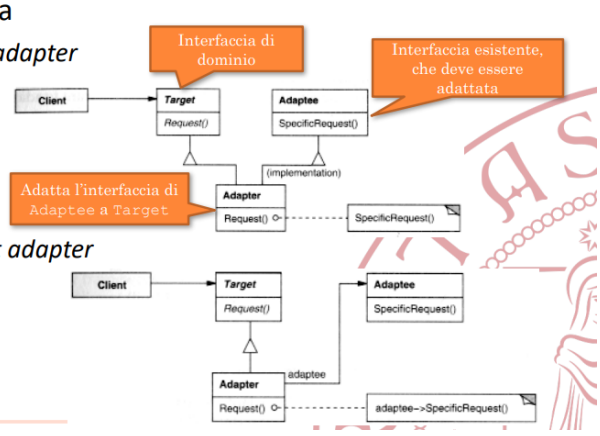
Il design pattern adapter risolve problemi come:
- Come si può riutilizzare una classe che non ha un'interfaccia richiesta dal cliente?
- Come possono lavorare insieme classi che hanno interfacce incompatibili?
- Come si può fornire un'interfaccia alternativa per una classe?

Spesso la classe adapter separata, che converta l'interfaccia (incompatibile) di una classe (adattata) in un'altra interfaccia (target) richiesta dai client; per fare ciò, deve supportare un comportamento polimorfico (converte un'interfaccia in un'altra affinché corrisponda a quanto si aspetta il client).

Strutturalmente, abbiamo due componenti:
- *Class adapter*, che usa l'ereditarietà per adattare un'interfaccia (*adaptee*) ad un'altra (*adapter*); può adattare una sola classe, ma diverse interfacce possono essere adattate usando l'ereditarietà multipla. Non può adattare classi che sono *final* o non abbiano un costruttore di default.
- *Object adapter*, adattando per composizione (usando l'interfaccia esposta da *target*, ma usando una reference all'adaptee passato in fase di costruzione). Può adattare classi multiple e qualsiasi oggetto, purché si abbia un'interfaccia compatibile.

Il suo scopo, da un punto di vista di applicabilità, è di:
- *Riutilizzo* di una classe esistente, non è conforme all'interfaccia target
- Creazione di classi *riusabili* anche con classi non ancora analizzate o viste
- Non è possibile adattare l'interfaccia attraverso ereditarietà (*Object adapter*)

- Struttura
  - *Class adapter*



  - *Object adapter*

- Il *Client* è una classe che contiene la business logic del programma
  - Non viene accoppiato alla classe concreta dell'Adapter, finché lavora con l'Adaptee tramite l'interfaccia del client

- Il *Target* è l'interfaccia che il client si aspetta, con una serie di protocolli attesi

- *L'Adaptee* è la classe che possiede la funzionalità che il cliente vuole utilizzare, ma non implementa l'interfaccia target

- *L'Adapter* è la classe che adatta l'Adaptee all'interfaccia Target, consentendo al client di utilizzare le funzionalità dell'adattatore/adaptee attraverso l'interfaccia di destinazione/target

- Nel caso di class adapter, estende Adaptee a Target
- Nel caso di object adapter, è la classe che ha un riferimento (reference) da Adaptee che implementa a Target

In questo modo, è possibile introdurre nuovi tipi di adattatori nel programma senza interrompere il codice client esistente. Questo può essere utile quando l'interfaccia della classe di servizio viene cambiata o sostituita: si può semplicemente creare una nuova classe di adattatori senza modificare il codice client.

Conseguenze:

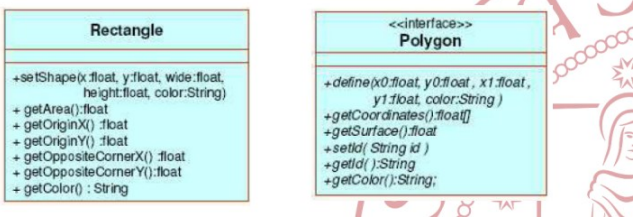
- Class adapter
  - *Non funziona* quando bisogna adattare una classe e le sue sottoclassi
  - Permette all'Adapter di modificare alcune caratteristiche dell'Adaptee
- Object adapter
  - Permette ad un Adapter di *adattare più tipi* (Adaptee e le sue sottoclassi)
  - Non permette di modificare le caratteristiche dell'Adaptee
  - Un oggetto adapter non è sottotipo dell'adaptee

Considerazioni:

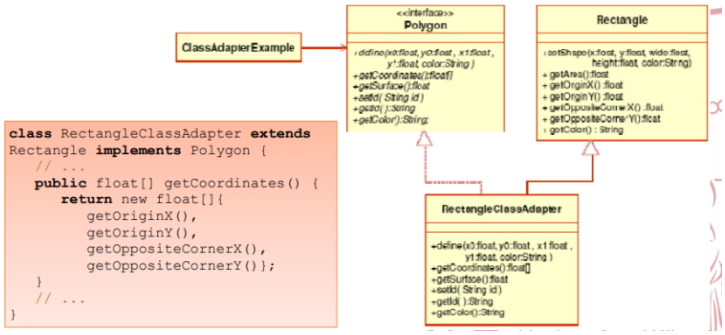
- Si usa la classe Adapter quando si vuole usare una classe esistente, ma la sua interfaccia non è compatibile con il resto del codice (anche per interfacce per sistemi complessi)
- Riduce i cambiamenti al sistema disaccoppiando il client dall'implementazione
- Può aggiungere un ulteriore livello di indirezione, che può aumentare la complessità del sistema e renderlo più difficile da capire
- Può essere più difficile da implementare rispetto ad altri design pattern, perché richiede la creazione di una nuova classe e la comprensione dell'interfaccia di destinazione e della classe di adattamento (a volte anche di classi multiple)

- Esempio

Convertire (adattare) una vecchia classe `Rectangle` ad una nuova interfaccia `Polygon`.

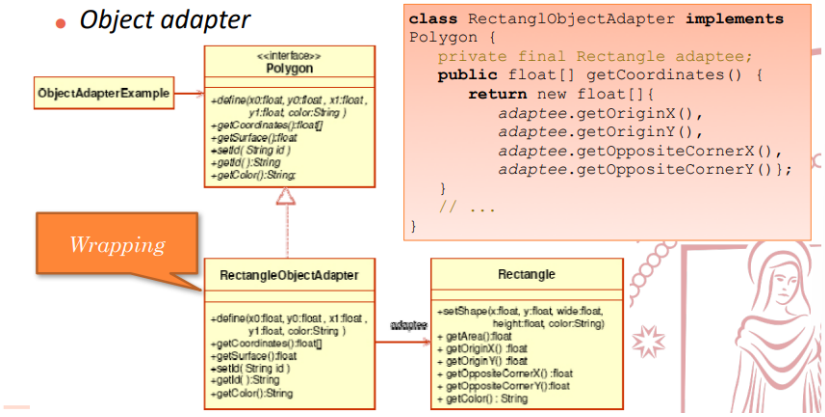


- Esempio
  - *Class adapter*



Qui *adattiamo* Rectangle a Polygon, usando i metodi comuni, ma adattando al contesto di Polygon.

- Esempio
  - *Object adapter*




```
class RectanglObjectAdapter implements
Polygon {
   private final Rectangle adaptee;
   public float[] getCoordinates() {
      return new float[]{
         adaptee.getOriginX(),
         adaptee.getOriginY(),
         adaptee.getOppositeCornerX(),
         adaptee.getOppositeCornerY()};
   }
   // ...
}
```

In questo contesto, invece, non è Rectangle, ma un generico *adaptee* a adattarsi al concetto di Polygon.

Implementazione:

- Si cerca di individuare *l'insieme minimo di funzioni* (narrow) da adattare, per renderlo più semplice da implementare e manutenere
  - Non occorre progettare l'adapter sulla base della libreria, ma l'opposto; si cerca idealmente di fare in modo di usare un solo adapter per contesto.
  - Questo si basa sull'utilizzo di operazioni astratte.
- Diverse varianti strutturali alternative (fondendo i pezzi sulla base della business logic):
  - (Client – Target) + Adapter
  - Client + Target + Adapter

Un esempio di Adapter in Java:

```
// The target interface that the client expects
interface Shape {
   void draw();
}

// The adaptee class that has the functionality that we want to adapt
class TextView {
   void getOrigin(int x, int y) {
      // implementation not relevant
   }
   void getExtent(int width, int height) {
      // implementation not relevant
   }
   boolean isEmpty() {
      // implementation not relevant
      return true;
   }
   void draw() {
      // implementation not relevant
   }
}

// The adapter class that adapts the adaptee to the target interface
class TextShape implements Shape {
   private TextView textView;

   TextShape(TextView textView) {
```

In questo esempio, la classe *TextView* ha la funzionalità che vogliamo utilizzare, ma non implementa l'interfaccia *Shape* che il client si aspetta.

La classe adattatore *TextShape* adatta la classe *TextView* all'interfaccia *Shape*, implementando l'interfaccia *Shape* e delegando a un oggetto *TextView*.

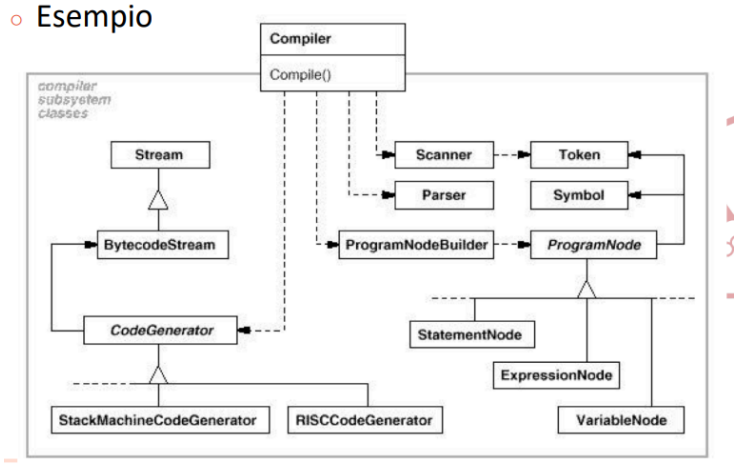
Il client può quindi utilizzare l'oggetto *TextShape* come se fosse uno *Shape*, anche se in realtà sta adattando un oggetto *TextView*.

```
        this.textView = textView;
    }

    @Override
    public void draw() {
        textView.draw();
    }
}
// Client code that uses the target interface
class DrawingEditor {
    public void draw(Shape shape) {
        shape.draw();
    }
}

// Example usage
TextView textView = new TextView();
TextShape textShape = new TextShape(textView);
DrawingEditor editor = new DrawingEditor();
editor.draw(textShape);
```

Consideriamo poi il Facade Pattern (cioè *façade*, letto come *"fasad"*) o classe facciata, che è un oggetto che funge da *interfaccia frontale* per mascherare un codice sottostante o strutturale più complesso tramite un'interfaccia unica.
Questo struttura un sistema in sottosistemi, diminuendo la complessità ma aumentando le dipendenze tra i sottosistemi. La gestione delle dipendenze è comunque semplificata, nascondendo la complessità del sistema (non nascondendo le funzionalità low-level).

- Esempio

Esempio utile: il compilare riunisce come facciata una serie di funzionalità (interpretazione dei flussi/stream I/O, scan, parse/elaborazione, lettura di token, connessioni nodi e variabili, etc.), *senza esserne dipendente*.



- La *Facade* fornisce un accesso comodo a una parte particolare della funzionalità del sottosistema. Sa dove indirizzare la richiesta del client e come far funzionare tutte le parti mobili.
    - È possibile creare una classe Additional Facade per evitare di inquinare una singola Facade con caratteristiche non correlate, che potrebbero renderla un'altra struttura complessa.

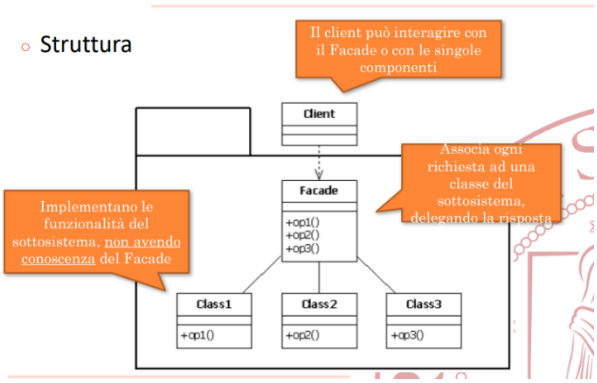
- Il sottosistema complesso è composto da decine di oggetti diversi. Per far sì che tutti facciano qualcosa di significativo, è necessario addentrarsi nei dettagli dell'implementazione del sottosistema, come inizializzare gli oggetti nell'ordine corretto e fornire loro i dati nel formato appropriato.
    - In questo caso, il compilatore riunisce tante componenti, sapendo l'ordine con il quale invocare le singole componenti

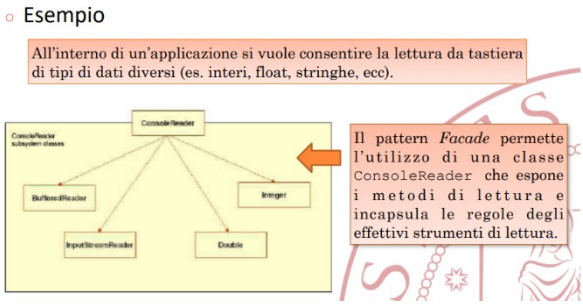
- Le classi del sottosistema non sono consapevoli dell'esistenza della Facade. Esse operano all'interno del sistema e lavorano direttamente con gli altri.

- Il client utilizza la Facade invece di chiamare direttamente gli oggetti del sottosistema.

Una facade può:
- migliorare la leggibilità e l'usabilità di una libreria software, mascherando l'interazione con componenti più complessi dietro un'unica (e spesso semplificata) API
- fornire un'interfaccia specifica per il contesto a funzionalità più generiche (completa di convalida degli input specifica per il contesto)
- servire come punto di partenza per un più ampio refactor di sistemi monolitici o strettamente accoppiati in favore di codice più sciolto
- diminuisce la complessità del sistema, ma aumenta le dipendenze tra sottosistemi
- funziona bene quando minimizza il codice utilizzato e quando si desidera un'interfaccia più semplice o più facile per un oggetto sottostante.

L'applicabilità risponde appunto a queste caratteristiche:
- Necessità di una *singola interfaccia semplice*
  - Design pattern tendono a generare tante piccole classi
  - Vista di default di un sottosistema
- *Disaccoppiamento* tra sottosistemi e client
  - Nasconde i livelli fra l'astrazione e l'implementazione
- *Stratificazione* di un sistema in layers
  - Architettura Three tier





Come si vede, la classe è proprio di facciata; non conosce dettagli dei client o delle sottoclassi, ma semplicemente comunica con loro "mascherando" il proprio codice.

Conseguenze:
- *Riduce* il numero di classi del sottosistema con cui il client deve *interagire*
- Realizza un *accoppiamento lasco* (*loose coupling*) tra i sottosistemi e i client
  - Eliminazione delle dipendenze circolari
  - Aiuta a *ridurre i tempi di compilazione* e di building
- Non nasconde completamente le componenti di un sottosistema
- Single point of failure (in quanto, dando troppe funzionalità ad un singolo oggetto facade, qualora fallisse, l'intero sistema viene compromesso)
- *Sovradimensionamento* della classe Facade (se diventa troppo grande, si ha il rischio che il software diventi troppo dipendente dall'interfaccia universale)

Implementazione:

- Classe Facade come classe astratta
  - Una classe concreta per ogni “vista” (implementazione) del sottosistema
- Gestione di classi da più sottosistemi
- Definizione di interfacce “pubbliche” e “private”
  - Facade nasconde l’interfaccia “privata”
- Singleton pattern: una sola istanza del Facade

Vediamo un esempio di Facade in Python:

```
class Computer:
    def get_electric_shock(self):
        return "Ouch!"

    def make_sound(self):
        return "Beep beep!"

    def show_loading_screen(self):
        return "Loading.."

    def bam(self):
        return "Ready to be used!"

    def close_everything(self):
        return "Bup bup bup buzzzz!"

    def sooth(self):
        return "Zzzzz"

    def pull_out_plugh(self):
        return "Hah!"

class ComputerFacade:
    def __init__(self, computer):
        self._computer = computer

    def turn_on(self):
        return (
            self._computer.get_electric_shock(),
            self._computer.make_sound(),
            self._computer.show_loading_screen(),
            self._computer.bam()
        )

    def turn_off(self):
        return (
            self._computer.close_everything(),
            self._computer.pull_out_plugh(),
            self._computer.sooth()
        )
computer = ComputerFacade(Computer())
print(computer.turn_on())  # prints ("Ouch!", "Beep beep!", "Loading..", "Ready to be used!")
print(computer.turn_off())  # prints ("Bup bup bup buzzzz!", "Hah!", "Zzzzz")
```

In questo esempio, la classe *Computer* rappresenta un sistema complesso con diversi metodi che eseguono varie operazioni relative all'accensione e allo spegnimento di un computer. La classe *ComputerFacade* è una facciata che semplifica l'interfaccia della classe *Computer*, fornendo un'interfaccia di livello superiore con due soli metodi: *turn_on* e *turn_off*.

Il codice client può usare la classe *ComputerFacade* per accendere e spegnere il computer, senza dover conoscere i dettagli delle complesse operazioni eseguite dalla classe *Computer*. Questo rende il codice client più semplice e comprensibile, poiché deve preoccuparsi solo dell'interfaccia di alto livello fornita dalla facciata.

Scritto da Gabriel

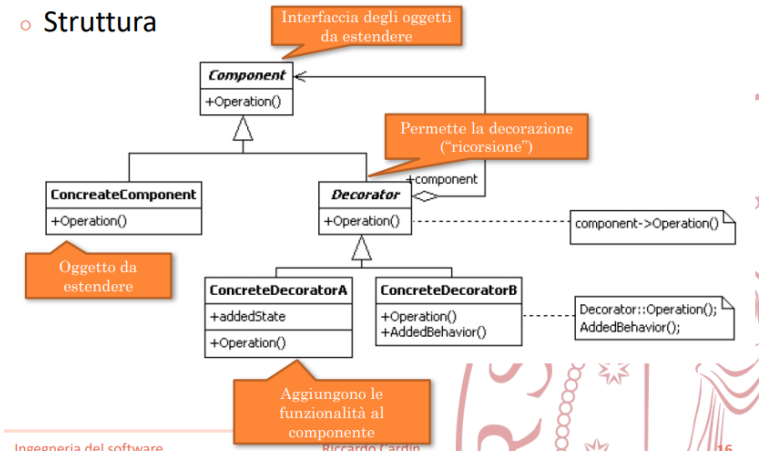
Il Decorator Pattern (anche questo noto come *wrapper*) aggiunge a runtime/dinamicamente caratteristiche (*non modificare, ma aggiungere*). Il Decorator ingloba un componente in un altro oggetto che aggiunge la funzionalità. Ha lo scopo di aggiungere nuovo comportamento o estendere la funzionalità di un oggetto aggiungendo nuovi Decorator a runtime. Ciò incapsula il comportamento in una classe Decorator piuttosto che aggiungerlo direttamente all'oggetto che viene decorato.

In particolare, è proprio come i vestiti; si pensi ad una felpa o a un impermeabile: essi estendono il comportamento normale, ma non sono parte della persona e si usano solo in un certo contesto, togliendoli quando ho finito di doverli usare.

Per quanto riguarda l'applicabilità:

- *Aggiungere* funzionalità *dinamicamente* ad un oggetto in modo *trasparente*
- Funzionalità che possono essere "circoscritte" (quindi, utili in quel contesto)
- Estensione via *subclassing* non è possibile
    - Esplosione del numero di sottoclassi
    - Non disponibilità della classe al subclassing

Identifichiamo la seguente struttura:



- Il *Component* definisce l'interfaccia per gli oggetti che possono essere decorati

- Il *Concrete Component* è una classe di oggetti da decorare. Definisce il comportamento di base, che può essere modificato dai decoratori.

- La classe *Decorator* è una classe astratta che definisce l'interfaccia per i decoratori. In genere ha un campo per memorizzare un riferimento all'oggetto da decorare e implementa l'interfaccia Component, inoltrando le chiamate all'oggetto decorato.

- I *Concrete Decorators* definiscono comportamenti aggiuntivi che possono essere aggiunti dinamicamente ai componenti, normalmente implementando i metodi di Decorator e modificando questi come voluto.

- Il *Client* usa la decorazione, purché funzioni con tutti gli oggetti tramite l'interfaccia del componente.

Conseguenze:

- Maggiore *flessibilità* della derivazione statica
- Combinazione di vari comportamenti con vari decorator ed estensione senza nuove sottoclassi
- *Evita* classi "*agglomerati* di funzionalità" in posizioni alte delle gerarchia
    - La classi componenti diventano più semplici

Scritto da Gabriel

  - Software as a Service (SaaS), specializzando il software al contesto se utile
- Il decoratore e le componenti non sono uguali
  - Non usare nel caso in cui la funzionalità si basi sull'identità
- Proliferazione di piccole classi simili e single-purpose
  - Facili da personalizzare, ma difficili da comprendere e testare
  - Difficili da rimuovere quando si stratificano
  - La configurazione iniziale degli strati potrebbe non essere il massimo
- Potrebbe non essere necessario o appropriato utilizzare il pattern nei casi in cui l'oggetto da decorare abbia un'interfaccia relativamente semplice e non sia necessaria la flessibilità di aggiungere nuovi comportamenti in fase di esecuzione

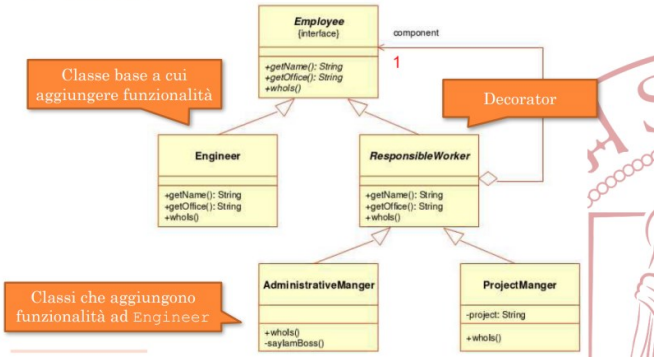
- Esempio

Si possiede un modello di gestione di oggetti che rappresentano gli impiegati (`Employee`) di una azienda. Il sistema vuole prevedere la possibilità di "promuovere" gli impiegati con delle responsabilità aggiuntive (e adeguato stipendio :P).

Ad esempio, da impiegato a capoufficio (`AdministrativeManager`) oppure da impiegato a capo progetto (`ProjectManager`).

Nota: Queste responsabilità non si escludono tra di loro.

- Esempio



Implementazione:

- Interfaccia del decoratore deve essere conforme a quella del componente
- Omissione della classe astratta del decoratore
  - Grandi gerarchie di classi già presenti
- Mantenere "*leggera*" (stateless) *l'implementazione* del Component
- Ha senso modificare direttamente i Decorator quando le componenti sono "leggere"

Vediamo un esempio di Decorator in Python (con *pass* che è una keyword che "non fa nulla", ma viene usato come placeholder per cambiarlo in fase di produzione e comunque far eseguire il programma):

```
from abc import ABC, abstractmethod

class Beverage(ABC):
    @abstractmethod
    def get_description(self):
        pass

    @abstractmethod
    def cost(self):
        pass

class Espresso(Beverage):
    def get_description(self):
        return "Espresso"

    def cost(self):
        return 1.99

class HouseBlend(Beverage):
    def get_description(self):
```

In questo esempio, la classe *Beverage* è una classe astratta che definisce l'interfaccia per gli oggetti che possono essere decorati. Le classi *Espresso* e *HouseBlend* sono implementazioni concrete della classe *Beverage*.

La classe *CondimentDecorator* è una classe astratta che estende la classe *Beverage* e definisce il metodo *get_description* che tutte le classi concrete di decoratori devono implementare.

Le classi *Mocha* e *Whip* sono implementazioni concrete della classe *CondimentDecorator* che aggiungono nuovi comportamenti all'oggetto da decorare, modificando il comportamento dei metodi *get_description* e *cost*.
Il codice del client crea un'istanza della classe *Espresso* e poi la decora con le classi *Mocha* e *Whip*.

Scritto da Gabriel

```
        return "House Blend Coffee"

    def cost(self):
        return 0.89

class CondimentDecorator(Beverage, ABC):
    @abstractmethod
    def get_description(self):
        pass

class Mocha(CondimentDecorator):
    def __init__(self, beverage):
        self._beverage = beverage

    def get_description(self):
        return f"{self._beverage.get_description()}, Mocha"

    def cost(self):
        return 0.20 + self._beverage.cost()

class Whip(CondimentDecorator):
    def __init__(self, beverage):
        self._beverage = beverage
    def get_description(self):
        return f"{self._beverage.get_description()}, Whip"

    def cost(self):
        return 0.10 + self._beverage.cost()

beverage = Espresso()
print(f"{beverage.get_description()} ${beverage.cost()}")  # prints "Espresso $1.99"

beverage = Mocha(beverage)
beverage = Mocha(beverage)
beverage = Whip(beverage)
print(f"{beverage.get_description()} ${beverage.cost()}")  # prints "Espresso, Mocha, Mocha, Whip $2.49"

beverage2 = HouseBlend()
beverage2 = Mocha(beverage2)
beverage2 = Mocha(beverage2)
beverage2 = Whip(beverage2)
print(f"{beverage2.get_description()} ${beverage2.cost()}")  # prints "House Blend Coffee, Mocha, Mocha,
Whip $1.49"
```

Proseguiamo con il Proxy pattern, pattern che fornisce un oggetto *surrogato* o *placeholder* per un altro oggetto, che controlla l'accesso a tale oggetto. Permette di creare una classe wrapper che controlla l'accesso a un oggetto sottostante, fornendo al contempo la stessa interfaccia dell'oggetto sottostante. Si pensi ad un caso molto semplice: app software con accesso a risorse remote alle quali occorre autenticazione per potervi accedere/poterne usufruire. Un proxy permette di gestire il processo di autenticazione e di inoltrare la richiesta alla risorsa remota, con la stessa interfaccia del client e senza modificare quest'ultimo.

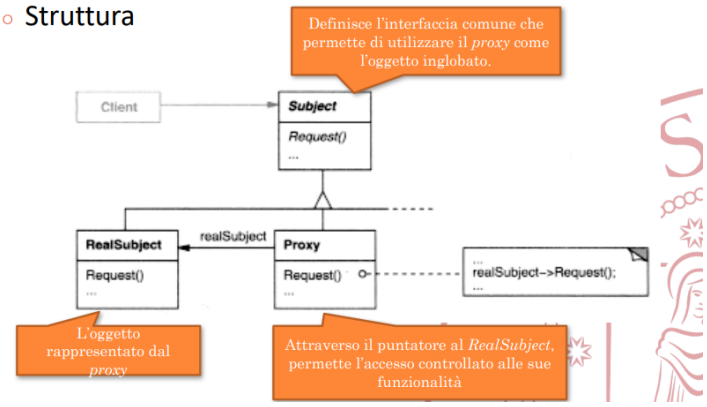
Infatti, intende:

- Creare oggetti in modo semplice quando la duplicazione di codice non basta
- Rinviare il costo di creazione di un oggetto all'effettivo utilizzo (on demand)
- Il proxy agisce come l'oggetto che ingloba
    - Stessa interfaccia
- Le funzionalità dell'oggetto "inglobato" vengono accedute attraverso il proxy
    - Senza l'accesso vero e proprio (virtual proxy
- Mantiene i riferimenti con l'oggetto reale
- Previene la duplicazione di elementi potenzialmente grandi e pesanti a livello di memoria, mantenendo anche la sicurezza in modo distribuito

In termini di applicabilità, ne esistono di diversi tipi:

- Remote proxy
    - *Rappresentazione locale* di un oggetto che si trova in uno spazio di indirizzi differente
    - Classi stub (cioè classi che simulano il comportamento di funzionalità software) in Java RMI
- Virtual proxy
    - Creazione di oggetti complessi on-demand
    - Questo accade quando si ha un oggetto di servizio pesante che spreca risorse di sistema rimanendo sempre attivo, anche se serve solo di tanto in tanto.
- Protection proxy
    - Controllo degli accessi (diritti) all'oggetto originale
    - Questo avviene quando si vuole che solo client specifici possano utilizzare l'oggetto del servizio; ad esempio, quando gli oggetti sono parti cruciali di un sistema operativo e i client sono varie applicazioni lanciate (comprese quelle dannose).
- Smart proxy
    - Esattamente come gli *smart pointer* in C++, conduce lavoro aggiuntivo ogni qual volta il client accede all'oggetto (es. uso di un lock controllando nessuno acceda)
- Logging proxy
    - Questo è il caso in cui si vuole mantenere una cronologia delle richieste all'oggetto del servizio.
- Caching proxy
    - In questo caso è necessario memorizzare nella cache i risultati delle richieste dei client e gestire il ciclo di vita di questa cache, soprattutto se i risultati sono piuttosto grandi.

- Struttura



Se è necessario eseguire qualcosa prima o dopo la logica primaria della classe, il proxy consente di farlo senza modificare la classe.

Poiché il proxy implementa la stessa interfaccia della classe originale, può essere passato a qualsiasi client che si aspetta un vero oggetto di servizio.

- La classe *RealSubject* è la classe che rappresenta l'oggetto a cui accede il codice client. Definisce i metodi che forniscono la funzionalità effettiva dell'oggetto.

- La classe *Subject* è un'interfaccia che definisce i metodi che la classe proxy e la classe oggetto reale devono implementare

- La classe Proxy è una classe che controlla l'accesso alla classe RealSubject. Implementa la stessa interfaccia della classe RealSubject e ha un campo per memorizzare un riferimento all'oggetto RealSubject. In genere inoltra le richieste all'oggetto RealSubject e può modificare la richiesta o la risposta prima o dopo l'inoltro.

Conseguenze:

- Introduce un *livello di indirezione* che può essere "farcito" (quindi, ramifica il fatto di essere intermediario specializzando le funzionalità in base al singolo contesto)
    - Remote proxy, nasconde dove un oggetto risiede
    - Virtual proxy, effettua delle ottimizzazioni
    - Protection proxy, definisce ruoli di accesso alle informazioni
-
- *Copy-on-write*
    - La copia di un oggetto viene eseguita unicamente quando la copia viene modificata
- Richiede l'introduzione di nuove classi, che in alcuni casi possono essere molte
- Realizza un'incapsulazione a microservizi, reindirizzando le singole richieste d'accesso nel modo opportuno
- Possibile *bottleneck*/collo di bottiglia (potrebbe essere in overload e non rispondere correttamente in modo scalabile al cambiamento)
- La risposta del servizio potrebbe essere ritardata
- Il proxy funziona anche se l'oggetto del servizio non è pronto o non è disponibile
- È possibile gestire il ciclo di vita dell'oggetto servizio quando i client non se ne preoccupano.

Implementazione:

- Implementazione "a puntatore" (quindi, uso riferimenti/reference e accedo agli attributi privati)
    - Overload operatore -> e * in C++
- Alcuni *proxy* agiscono in modo differente rispetto alle operazioni
    - In Java costruzione tramite *reflection* (Spring, H8...), che ci permette di ispezionare e/o modificare gli attributi di runtime di classi, interfacce, campi e metodi. Questo è particolarmente utile quando non conosciamo i loro nomi al momento della compilazione
    - Inoltre, possiamo istanziare nuovi oggetti, invocare metodi e ottenere o impostare i valori dei campi utilizzando la reflection
    - Proxy per più tipi, grazie al fatto che *subject* è una classe astratta
        - ma non se il proxy deve istanziare il tipo concreto
- Rappresentazione del subject nel proxy (ho accesso al proxy, ma non al real subject/oggetto)
- Finché non interagisco con l'oggetto, non mi serve metterlo in memoria (*lazy loading*); questo si realizza solo quando ho i metodi necessari
    - Può produrre side-effects indesiderati (non si sa più cosa e quando carica)

Ecco un esempio di Proxy in Python:

```
from abc import ABC, abstractmethod

class Subject(ABC):
    @abstractmethod
    def request(self):
        pass

class RealSubject(Subject):
    def request(self):
        return "Handling request.."

class Proxy(Subject):
```

```
    def __init__(self, real_subject):
        self._real_subject = real_subject

    def request(self):
        if self.check_access():
            return self._real_subject.request()
        else:
            return "Access denied!"

    def check_access(self):
        # check access permissions
        return True

real_subject = RealSubject()
print(real_subject.request())  # prints "Handling request.."

proxy = Proxy(real_subject)
print(proxy.request())  # prints "Handling request.."
```

In questo esempio, la classe *Subject* è un'interfaccia che definisce il metodo di richiesta che entrambe le classi *RealSubject* e *Proxy* devono implementare. La classe *RealSubject* è un'implementazione concreta della classe *Subject*, che fornisce l'effettiva implementazione del metodo di richiesta.

La classe *Proxy* è una classe che controlla l'accesso alla classe *RealSubject*. Ha un campo per memorizzare un riferimento all'oggetto *RealSubject* e implementa il metodo *request* inoltrando la richiesta all'oggetto *RealSubject* e controllando i permessi di accesso con il metodo *check_access*.

Il codice client crea un'istanza della classe *RealSubject* e la avvolge con un oggetto *Proxy* per controllare l'accesso all'oggetto *RealSubject*. Il codice client può quindi utilizzare l'oggetto *Proxy* nello stesso modo dell'oggetto *RealSubject*, senza dover conoscere i dettagli del meccanismo di controllo degli accessi utilizzato.

Diario di bordo: Verifica, Casi d'uso, sprint e pianificazione

Dubbi riscontrati e ordine di discussione:

- Impatto Temporale e Verifica

Per la RTB, ci sarà una finestra temporale con cui incontrarsi con Cardin ed esporre quanto è stato fatto; poi, ci si sottopone a Tullio per la fine della revisione (mostrando consapevolezza delle tecnologie e le ragioni con Cardin e modo di lavorare/metodo utilizzato con Tullio).
Per verificare la pianificazione, servono i contenuti, poi serve la data.
L'operazione di verifica rallenta l'opera e serve a valutare quanto fatto (nel senso di loop/iterazione).

Questa si basa su una checklist pronta (condizioni logiche/cose che devono essere vere per conseguenza dell'attività svolta), che deve esistere prima che inizino le attività.
La misura della fine è legata alla persona che fa; il verificatore serve come opinione terza, non avendo chi la esegue automatismo oggettivo. La tendenza è: più si va avanti, più la verifica si automatizza. Si intende che la checklist sia una *procedura* di controllo, perché dice *come si fa* a verificare (quindi, come procedura è calcolabile). Esiste una checklist per attività (nel senso di classi di compiti) e per istanza di attività.
Quindi → cosa devo fare – come devo controllarlo.

Il costo di automazione *deve* valerne la pena; ha senso metterlo a procedura, tale da automatizzarlo automisurandosi.
Quando l'esito di una verifica è negativo, la checklist aiuta a capire cosa è andato bene e, progressivamente, l'approccio converge e ragioniamo sul perché non sia corretto. Mai ipotizzare che la verifica abbia esito positivo, occorre ripensare tutto (si collassano i tempi). Il margine delle attività è da affinare (es. primo mese di progetto, è molto più preciso il metodo di lavoro per il gruppo).

Ogni singola attività ha un costo temporale orario e di risorse; il calendario deve basarsi sulla disponibilità reale delle persone, non sui giorni (la mancata disponibilità è un rischio da preventivare).
La checklist è un'attività procedurale (che aiuta ci ha e chi verifica); si pianifica nel concreto, automatizzando il ragionamento e responsabilizzando le persone a lavorare (sulla base del tempo garantito). Se la pianificazione ci porta troppo tempo, occorre riorganizzare i ruoli.

Scritto da Gabriel

- Casi d'uso

Il senso dei casi d'uso è capire il problema e aiuta a comprendere/pensare nozioni sul problema che inizialmente non erano state pensate, capendo se serve a realizzare il compito.
Per poter avere una soluzione, serve comprendere a fondo il problema e quindi iterare, correggendo progressivamente sulla distinzione tra requisiti lato utente (senso del capitolato è esplorativo, capire cosa si può fare) e soluzione (prodotto finito).

- Progettazione e Diagrammi di attività

Ha senso concentrarsi sul design e sui pattern: per realizzare una soluzione, serve vedere come risolvere il problema individuando bene i requisiti. In particolare: i diagrammi dei casi d'uso servono a fare l'analisi dei requisiti (con lo scenario dell'utente, cioè non vedendo la realizzazione e capendo che l'attore X fa la cosa Y), i diagrammi delle attività guidano la realizzazione.

- Sprint monolitico

Sapendo che l'organizzazione di molti è asincrona, ha senso che tutti si vedano in base al contesto. Ha senso non vedersi mai quando si ha molta competenza e si sa sempre cosa fare (board virtuale e massimo asincronismo), lavorando a regola d'arte con professionalità. Si impara dagli errori, ragionandoci sopra. Se ha senso vedersi tutti insieme bene, altrimenti si usano cerimonie brevi a cerimonia blindata, rimandando il resto a sottogruppi. Ovviamente, non vogliamo parallelismo asincrono, ma coordinazione agendo da ponte (tutti devono sapere tutto, costruendo un'informazione globale).

- Pianificazione a lungo termine

Non fissarsi per una revisione attendendo una data, ma si ragioni sulle attività imparando per errore (riducendo nel corso del tempo la distanza tra cose fatte e cose da fare), dando un margine di cautela.

Qualità di prodotto (qualità del software) (Vardanega)

Il concetto di qualità è un riscontro oggettivo di misurazione (correlata con l'idea di *valutazione*, nel senso di poter fare confronti) del grado di conformità alle attese del prodotto software. Essa va misurata in modo oggettivo e non retrospettivo e rappresenta un indice importante per chi realizza il prodotto usandola come indice per attuare miglioramenti, sia per chi utilizza il prodotto, che definisce la qualità come indice di garanzia del suo funzionamento, infine anche per chi lo valuta.

Distinguiamo quindi vari destinatari con punti di vista diversi:

- Chi fa (chi lo produce, indicando un resoconto delle attività svolte)
- Chi usa (chi valuta come soggetto finale la qualità)
- Chi valuta come terza parte (il prodotto abbia qualità sufficiente rispetto alla competizione/mercato/contesto)

Un'idea sbagliata di valutazione del software è considerare che il prodotto esista solo se è stato pagato: questo blocca l'innovazione, che non aspetta la domanda, ma crea anche senza risorse. Inoltre, l'innovazione continua sempre a compararsi sondando il mercato.

Fissiamo quindi le voci a glossario (riferito allo specifico dominio):

- Qualità: "Insieme delle caratteristiche di un'entità, che ne determinano la capacità di soddisfare esigenze sia espresse che implicite"
  (ISO 8402:1994, glossario dei termini, confluito in ISO 9000:2005)
  - Visioni della qualità che producono risultati osservabili tramite metriche
    - Intrinseca: conformità ai requisiti, idoneità all'uso
    - Relativa: soddisfazione del cliente
    - Quantitativa: misurazione oggettiva, per confronto
  - Si adotta una checklist, si capisce con un insieme di punti se siamo effettivamente a posto
  - Come si fa ad avere qualità?
    - Prima era una visione in retrospettiva (si agisce vedendo poi i risultati)
    - Ora è un processo continuo (arrivare al collaudo certi del prodotto)
  - In carico al Sistema Qualità del fornitore

- Sistema Qualità: "Struttura organizzativa, responsabilità, procedure e risorse messe in atto al perseguimento della qualità" (ISO 8402:1994 → ISO 9000:2005)
  - Si riassume in tre elementi (tutti compresi nel nostro Piano di Qualifica)

    - Piano della Qualità
      - "Le attività del Sistema Qualità mirate a fissare gli obiettivi, nonché processi e risorse necessarie per conseguirli" (ISO 9000)
        - Visione orizzontale, per fissare le politiche aziendali
        - Visione verticale, specifica di prodotto/servizio
          - Fissare gli standard del prodotto finale e i tipi di testing
        - Basate sull'adozione di uno specifico way of working
          - Grado di conformità riflesso nel corrispondente cruscotto
        - La parola "piano" intende "previsione impegnativa", quindi necessario a raggiungere gli obiettivi prefissati

    - Controllo di Qualità
      - "Le attività del Sistema Qualità pianificate e attuate per assicurare che il prodotto soddisfi le attese" (ISO 9000)
        - Prevenire è meglio che curare (si concentra sull'ispezione)
          - Assicurare conformità passo-passo invece che solo a fine corsa
          - Attuare il way of working
          - Controllarne gli effetti tramite il cruscotto di controllo (modo non invasivo sulle attività)
        - Questo è *Quality Assurance* (accertamento/garanzia di qualità), cioè fornire garanzia che le azioni fatte siano progressivamente miglioranti e portino a zero il tasso d'errore
          - Questa attività si occupa di determinare in modo sistematico se un prodotto o un servizio soddisfino o meno una serie di requisiti specifici. Si focalizza sui processi e sulle procedure
          - In questo modo, sappiamo facilmente individuare cosa va male e mitigare facilmente i rischi, sapendo che l'attività potrebbe non andare bene

- Miglioramento continuo
  - Con questo si intende che la qualità vada ricercata, adottando un way of working che si autovaluta e compia miglioramenti progressivi. L'innovazione si crea sbagliando, correggendo e sperimentando.
  - Il principio di miglioramento semplifica il processo d'azione e di verifica, nonché di manutenzione
  - Significa concentrarsi sulle cose che facciamo più volte e cercare di realizzarle meglio, con meno risorse (efficienza) e ottenendo un buon risultato (efficacia)

Il Sistema di Qualità si basa su 7 principi fondamentali, cose che insieme concorrono ad un obiettivo.
A noi interessa in particolare che la qualità sia *basata sull'evidenza* (*evidence-based*). Il quality management è fare in modo che le cose vadano secondo i piani e tutto funzioni secondo quanto prestabilito. Ecco perché *i piani sono la sostanza del management e dicono che traccia scavare*.

ISO 9000 Quality Management Principles

Listiamo quindi i vari principi:

- Attenzione al cliente (Customer focus)
  - Comprenderne i bisogni e il punto di vista
  - Allineare gli obiettivi a quelli del cliente
- Puntare in alto (Aim high)
  - Avere obiettivi ambiziosi
  - Mirare a superare le aspettative
  - Dare fiducia, responsabilità, riconoscimenti
- Guida illuminata (Leadership)
  - Valorizzare le competenze
  - Assegnare responsabilità e valutare prestazioni
  - Discutere apertamente di problemi e vincoli
  - Condividere la conoscenza
  - Coinvolgere tutti nel miglioramento continuo
- Gestione delle relazioni (Relationship management)
  - Entro il team
  - Con gli stakeholder
- Lavorare a processi (Process approach)
  - Istanziare processi alle necessità di progetto
  - Assegnare risorse congruenti con le necessità
  - Comprendere le dipendenze tra attività
  - Cercare sempre il miglioramento continuo
- Auto-miglioramento (Improvement)
  - Stabilizzare i miglioramenti conseguiti
  - Attuare attività di auto-miglioramento
  - Motivare tutti all'auto-miglioramento
- Decidere sulla base dei fatti (Evidence based decision making)
  - Alimentare il cruscotto di controllo con dati accurati e affidabili
  - Usare i dati per indirizzare le decisioni gestionali

L'evidenza si basa su *fatti che noi comprendiamo per decidere*. Questa è data dal cruscotto, che è la *base delle informazioni che voglio sapere* (non necessariamente *sapere tutto*, ma sapere *quello che serve*).

I modelli (visti come insieme di specifiche che descrivono un fenomeno di interesse aiutando a studiarlo, comprenderlo, misurarlo, intendo per noi il SW stesso) della qualità SW servono per uniformare punti di vista diversi e per favorire valutazione oggettiva:

- *Lato sviluppo*: prospettiva del progetto
- *Lato uso*: prospettiva utente
- *Lato direzione*: costi/benefici del way of working

Un insieme di modelli utili riguardo la qualità del SW segue:

- Cosa significa "qualità" in un prodotto SW
  - *ISO/IEC 9126:2001 SWE Product Quality*
    - Ha tre tipi di visioni
      - Qualità *esterna* (comportamento durante l'esecuzione del SW e rilevata dai test prestabiliti)
      - Qualità *interna* (scrittura del codice sorgente e della documentazione)
      - Qualità *in uso* (punto di vista dell'utente finale)
    - Qualità interna ed esterna hanno varie caratteristiche per il prodotto software:
      - Funzionalità → soddisfa le esigenze stabilite
      - Affidabilità → mantiene un certo livello di prestazioni
      - Efficienza → è compreso ed apprezzato dall'utente
      - Manutenibilità → può essere modificato/migliorato/corretto nel tempo
      - Portabilità → può essere facilmente trasportato da un ambiente di lavoro ad un altro
    - Visione ottima di questo standard:
      - www.colonese.it/00-Manuali_Pubblicatii/07-ISO-IEC9126_v2.pdf
- Metriche per la *valutazione quantitativa* della qualità
  - *ISO/IEC 14598:1999 SW Product Evaluation*
    - Fornisce linee guida per l'associazione di una misurazione quantitativa (metrica) ad una data sotto-caratteristica individuata nello standard ISO 9126
    - In poche parole, dice che esiste sempre almeno una relazione tra gli oggetti della misurazione e una particolare misurazione formale (metrica), cercando di misurare per quanto possibile attributi interni ed esterni di qualità
    - Può essere difficile farlo in modo formale; cerca di essere il più possibile oggettivo
    - Possibile preview:
      - https://cdn.standards.iteh.ai/samples/24902/fca297c50e12418caea8569938ea8d82/ISO-IEC-14598-1-1999.pdf

Oggi queste due dimensioni sono *unificate*:

  - *ISO/IEC 25000:2005 SQuaRE: Systems and software Quality Requirements and Evaluation*
    - Parte dalla persecuzione della qualità dei requisiti per definire un modello di qualità su cui avere una gestione di qualità con misurazioni della qualità che portano ad una corretta valutazione della qualità.
    - Possibile preview:
      https://cdn.standards.iteh.ai/samples/35683/917c6ad92a6e4c6c9326547e53f2dd7a/ISO-IEC-25000-2005.pdf

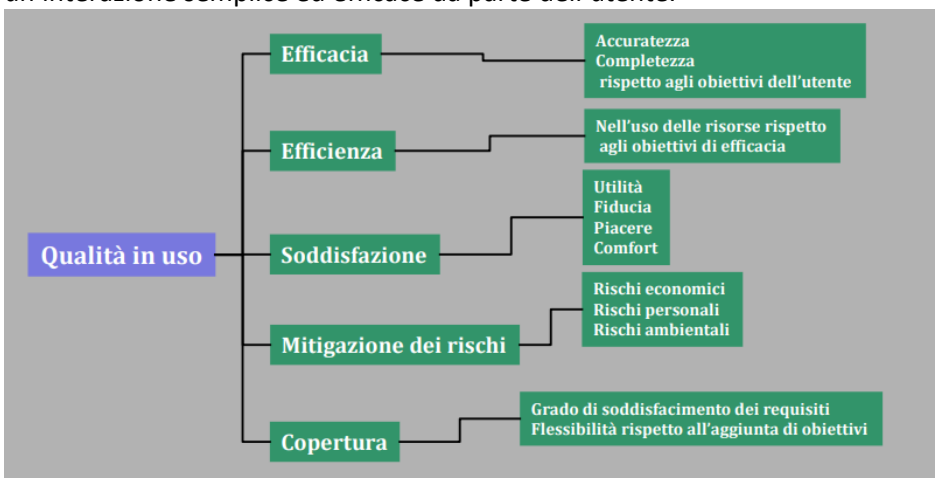
Altri modelli di qualità SW sono invece riferiti a requisiti più impliciti:

- 25010:2011 *Quality model*
  - Cosa significa «qualità SW»
- 25020:2019 *Quality measurement framework*
  - Come si misura la qualità SW
- 25030:2007 *Quality requirements*
  - Come si specificano i requisiti di qualità SW
- 25040:2011 *Quality evaluation*
  - Come si conduce la valutazione della qualità SW

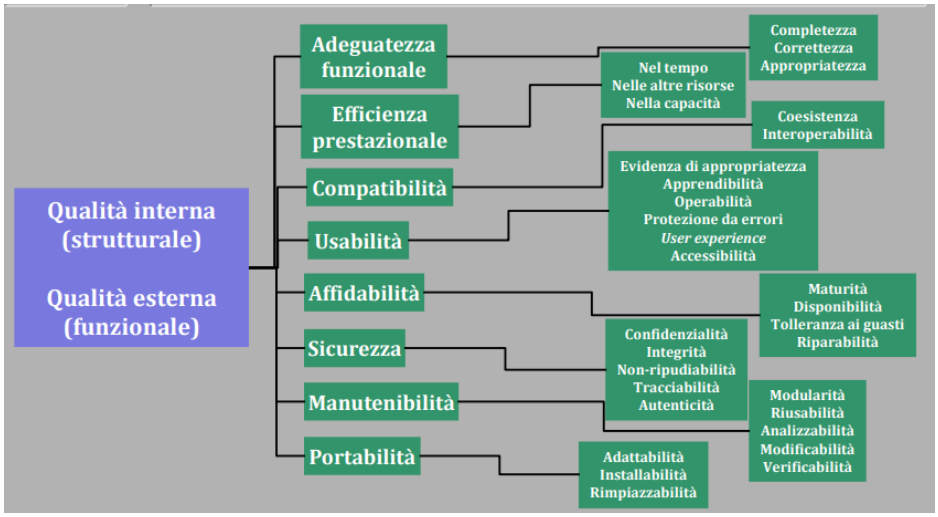
<u>Misurazione quantitativa</u>

- Il processo con cui assegnare simboli o numeri ad attributi di una entità, secondo regole definite (con sistemi che danno valori significativi; non è facile stabilirle a priori, ma si affinano facendo)

Individuiamo in questo modo le dimensioni di qualità (lato uso del prodotto) nell'immagine sottostante. Si qualifica un prodotto come "*utile*" quando i benefici superano di molto i costi che servono per realizzare un prodotto: non solo deve fare quello che ci aspettiamo, ma farlo anche nel modo più semplice possibile. Il concetto di essere utile si accompagna al concetto di essere *utilizzabile*: significa quindi che possiede un'interazione semplice ed efficace da parte dell'utente.



Successivamente le qualità intrinseche del prodotto, da un punto di vista interno/strutturale nell'immagine sottostante. Il concetto, come si vede, è molto ramificato e complesso da realizzare. Come si vede, distinguiamo chiaramente tra una qualità interna ed esterna, fatta di caratteristiche funzionali e caratteristiche di usabilità.



Scritto da Gabriel

La definizione di *metriche software/software metrics*, data da Sommerville (Software Engineering, 8th ed.):

- "Qualsiasi tipo di misurazione relativa a un software, a un processo o a una documentazione. Può consentire di quantificare gli attributi del prodotto e del processo e può essere utilizzata per prevedere gli attributi del prodotto o per controllare il processo software. Le metriche di prodotto possono essere utilizzate per predizioni generali o per identificare componenti anomali".

Qui qualche esempio di metrica relativo a varie componenti.

Entity	Metric
Program	SLOC
Effort	Person/days
Text	Gunning's Fog index

Fog index = [(average # words / sentence) + (# words of 3 syllables or more)] * 0.4

Ciascuna metrica si basa su *assunzioni sul software/metrics assumptions*, che hanno questa definizione:

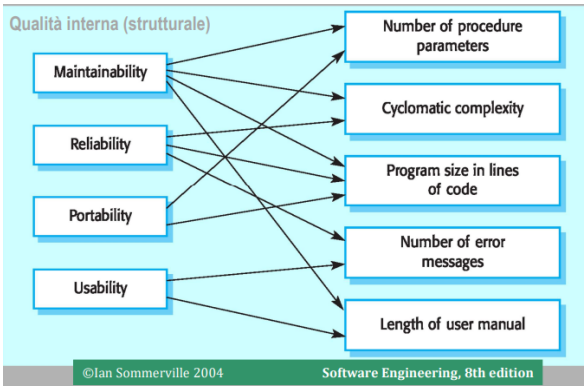
- Una proprietà o un attributo del software può essere misurato.
- In generale, esiste una relazione tra ciò che si può misurare e ciò che vogliamo sapere
  - Sappiamo solo come misurare gli attributi *interni*
    - Qualità del prodotto
  - Ma spesso siamo più interessati ad attributi *esterni*
    - Qualità nell'uso

Concretamente, avremo che:

- Una metrica misura in maniera appropriata un aspetto del sistema che viene usato come riscontro oggettivo di una particolarità del prodotto
- Una metrica viene misurata precisamente (con standard) e in modo accurato
- Una metrica viene usata nel contesto preciso
- Una metrica è guida per l'evidence-based decision making

Le metriche osservano valori su cose non misurabili ma che ci interessano, questo è il succo.

Per esempio, ogni forma a destra è misurabile, ma non univocamente; abbiamo infatti spesso a che fare con dei contesti con complessità molto alta (e.g. automatismi di controllo nel repo di prodotto).



Il processo di valutazione è basato su metriche predefinite e criteri di accettazione previsti.
In particolare, una volta determinate le metriche, si esegue un'attività di indagine (misurazione), interpretando le misure presenti (valutazione) e determinando come eseguire la validazione (accettazione), determinando quindi un giudizio.

Si intende, dunque, che il prodotto passi attraverso queste fasi come processo integrativo. I requisiti e i costi determinano metriche/misure/criteri.

Ciclicamente, il SW segue un ciclo di vita e, come tale, è soggetto a cambiamenti ed iterazioni (specie per quanto riguarda la qualità); il processo di definizione di quest'ultima, sia interna che esterna, condiziona gli attributi della qualità in uso, commisurati al contesto.

In questo momento, confrontiamo alcuni tipi di qualità:

- Qualità obiettivo → Soddisfacimento requisiti espliciti/impliciti
- Qualità richiesta → Soddisfacimento requisiti contrattuali

Prospettiva committente

- Qualità progettata → Design soddisfa i requisiti contrattuali
- Qualità stimata → Scelte realizzative soddisfano i requisiti contrattuali
- Qualità consegnata → Esito di collaudo e accettazione

Prospettiva del fornitore

Diario di bordo: PoC & RTB

Dubbi riscontrati:

- Proof of Concept – Cosa deve contenere

Il PoC è utile se, nello svilupparlo, deve contenere notizie su prodotto e tecnologie. Il PoC rappresenta il prodotto finito e quali tecnologie individuare per realizzare le tecnologie utili per realizzare il prodotto, per fare cose che assomigliano (selezione, anche abbastanza limitata, delle tecnologie utili per realizzare il progetto). Serve a capire come usare le tecnologie, in un contesto controllabile/realizzabile, se riesco/non riesco a fare determinate cose. Attività dimostrativa che serve a capire se usando le tecnologie di progetto possa essere ragionevole realizzare determinate cose con le tecnologie.
Alcuni proponenti chiedono un PoC significativo (anche condizionato dall'avere più PoC, in alcuni casi; non tutto in uno, ma più PoC per affrontare le singole parti).
Il primo PoC serve a realizzare le tecnologie dimostrative; similmente, il prodotto può avere più percorsi realizzativi (un PoC per il frontend, un PoC per il backend, ad esempio, etc.).

- Condizioni per RTB (previsioni di candidatura)

Sarà esposto un calendario dove candidarsi in termini di periodi; si discute con noi quale può essere l'orizzonte di periodo di candidatura. L'idea di massima è di metà/fine gennaio, sulla base di una serie di precondizioni (checklist di cose che devono essere pronte). Per ogni obiettivo che abbiamo, riusciamo a controllare in percentuale le cose fare rispetto alle cose da fare

Appendice T08: Qualità del Software & Qualità di processo (Vardanega)

(Eventuali approfondimenti:
https://www.math.unipd.it/~tullio/IS-1/2006/Approfondimenti/ISO-serie9000.pdf
https://www.praxiom.com/iso-90003.htm)

Nel software non esiste la perfezione, ma utile individuare quando il software ha dei limiti. Attenzione a questi ultimi, però; qualora scegliessimo una soluzione facile ma limitata, ci sarebbe un costo aggiuntivo dato dall'aver scelto un approccio non ottimale: questo è il concetto di *technical debt*.
Per ogni scelta occorre essere lucidi sugli:

- Obiettivi (della scelta)
- Vincoli (nella scelta)
- Alternative (alla scelta)
- Come la soluzione corrisponde al problema

Qualità cardine della progettazione sono la *fattibilità* e *verificabilità*.

Scritto da Gabriel

Buone tecniche di progettazione sono:

- *Decomposizione modulare*
    - Produce componenti tra loro indipendenti, con minimo accoppiamento e autosufficienza (coesione funzionale), intesi come *moduli*, per cui ognuna ha un compito preciso ed univoco che non prescinde dalle altre presenti
    - La ripartizione dei contenuti ha senso qualora le relazioni tra le componenti siano legate in modo semplice e separato correttamente e quando i legami siano creati in modo tale da essere autosufficienti
        - *Coesione* → Indicatore dell'intensità di relazione all'interno di una singola parte
            - Forte coesione → Buona modularità
    - Importante evitare che le parti all'interno di un modulo modifichino le parti esterne ad esso; in altre parole, si cerca di evitare legame indesiderato, indicando l'intensità della relazione all'interno di una singola parte
        - *Accoppiamento* → Indicatore dell'intensità di relazione tra parti distinte
            - Forte accoppiamento → cattiva modularità

- Incapsulazione (information hiding)
    - L'encapsulation consiste nel nascondere i dettagli di implementazione di un modulo.
        - Meno un modulo è a conoscenza di altri moduli, meno dipenderà dai loro cambiamenti
    - Gli altri moduli possono accedere alle funzionalità e ai dati solo attraverso l'interfaccia fornita (che è l'unica parte pubblica ed è il mezzo per accedere e comunicare).
    - Il dettaglio realizzativo è noto solo all'interno

- Uso dell'astrazione
    - Processo di rappresentazione delle caratteristiche importanti di un sistema o di un concetto senza includere dettagli non necessari
        - Ciò che è caratteristico dell'intera gerarchia è fissato in radice
        - Ciò che differenzia si aggiunge per specializzazione allontanandosi dalla radice
    - L'astrazione è usabile quando si ha una concretizzazione (dunque, caratteristiche utilizzabili in base al contesto), realizzata in vari modi
        - Per parametrizzazione (e.g. template a entità in C++)
        - Per specializzazione (e.g. da interfaccia a classe in Java e C++)
        - Per valorizzazione (e.g. da classe a oggetto tramite costruttore)

Ogni sistema è un aggregato complesso di componenti con compiti e flussi di esecuzione, in cui ciascun processo deve essere indipendente e funzionare bene.

- La *concorrenza* di buona qualità aiuta a decomporre il sistema in più entità garantendo:
    - Efficienza di esecuzione
    - Atomicità di azione
    - Consistenza e integrità dei dati condivisi
    - Semantica precisa di comunicazione e serializzazione
    - Predicibilità di ordinamento temporale e di comportamento
    - Organizzazione con moduli separati funzionalmente, ottimizzando i flussi I/O
- La *distribuzione* di buona qualità ripartisce il sistema in programmi su nodi distinti garantendo:
    - Bilanciamento di carico
    - Indipendenza nelle comunicazioni (con frugalità, cioè limitandosi allo stretto necessario)

Alcune problematiche critiche si presentano in casi di concorrenza e distribuzione, in relazione:

- al flusso di dati (determinando quando un certo dato sia disponibile);
- al flusso di controllo (determinando l'input in un particolare stato, del sistema o di una sua parte);
- al trascorrere del tempo (saper determinare l'arrivo di certo istante temporale).

Fondamentale non fare valutazioni ottimistiche; la progettazione deve sapere *cosa* fare in situazioni erronee, in relazione ai dati/tempo/controllo.

- L'obiettivo è ricercare *integrità concettuale*, cioè avere coerenza di informazioni e con uno stile uniforme tra tutte le parti del sistema e le loro interazioni
- Il design dice che "guardando il codice nel suo complesso, riconosco la struttura, capendo perché ogni modulo sta insieme e che funzionalità ha"; ciò capita spesso nelle API
    - Ciò richiede osservanza e vigilanza nel complesso (per facilitare parallelismo)

Consigli (*enforce intentions* – "assicurare che un programma si comporti nel modo in cui è stato concepito"):

- Il passaggio da progettazione a codifica è basato sul controllo con checklist automatizzabili
- In generale, questo può comportare il controllo che i dati di ingresso soddisfino determinati requisiti, la convalida che le uscite siano prodotte come previsto e la verifica che il programma segua il corretto flusso di controllo.
- Occorre rendere chiaro il confine tra esterno ed interno dei moduli
- Decidere in modo chiaro e codificare chiaramente ciò che si può specializzare, rendendo il resto immodificabile (*final, const,* etc.)
- Proteggere tutto ciò che non deve essere visto all'esterno (*private, protected*, etc.)
- Decidere quali classi possono produrre e quali istanze no (se classi a istanza singola, uso Singleton)

Un modo per fare quanto sopra è tramite il *defensive programming* – "tecnica che prevede la scrittura di codice resistente agli errori e sia in grado di gestire input inattesi o condizioni eccezionali"):

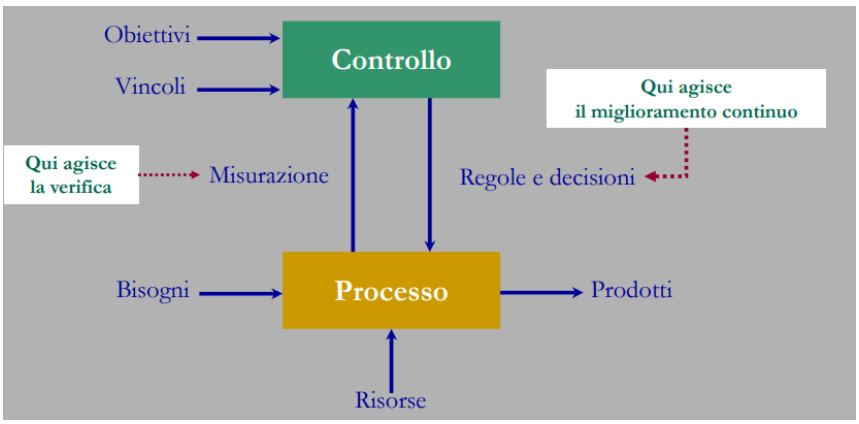
- Programmare esplicitamente il trattamento dei possibili errori nei dati di input (verificando la legalità prima di usarli) o sulle funzioni (fissando invarianti/pre-post condizioni)
- Definire la strategia di trattamento degli errori (*error handling*), con alcuni accorgimenti:
    - Attendere fino all'arrivo di un valore legale e validare i dati di input
    - Assegnare un valore predefinito (default)
    - Usare le asserzioni
    - Registrare gli errori in log persistenti
    - Gestire eccezioni ed errori (error handling)
    - Usare strumenti di analisi statica e automatizzare i test

Lo sviluppo è considerato processo (e non attività singola, ma aggregato di varie attività), sapendo che la qualità di processo è esigenza *primaria*, che richiede:

- Adozione sistematica piuttosto che occasionale
- Verifica costante, preventiva prima che reattiva
- Valutazione riproducibile e quindi automatizzata
- Disposizione costante al miglioramento

Un progetto è tale se ha almeno un processo primario; nel caso nostro abbiamo la fornitura (needs del capitolato) e lo sviluppo (soddisfazione dei needs tramite la realizzazione del prodotto). Ci si arriva tramite un insieme di attività ordinate:

- Analisi dei requisiti
- Progettazione
- Codifica
- Verifica



L'attività di *controllo* acquisisce informazioni sullo stato delle cose e usa queste informazioni per decidere cosa fare (in modo iterativo, tramite *control loop*).

Scritto da Gabriel

Perseguire qualità di processo è possibile, considerando che questo agisce secondo un tempo limitato:

- Si definisce il processo secondo il way of working, applicandolo e valutandone gli effetti
- Si controlla il processo al fine di migliorarlo (secondo buone metriche/strumenti di valutazione):
    - In efficacia: prodotti conformi alle attese
    - In efficienza: minori costi a pari qualità di prodotto
    - In esperienza: apprendere dall'esperienza (anche di altri)

Il processo di qualità ha due piani:

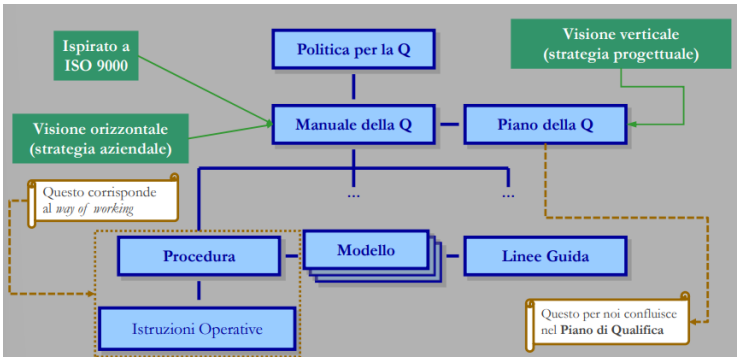
1) Azione e attuazione delle attività, sulla base di verifica e del miglioramento continuo sul way of working, immaginando come poter migliorare i processi
2) Controllo su come sto lavorando attraverso un "cruscotto"

La qualità di un *deck* (in altri termini, sistema, definito così da Tullio) si basa su una serie di norme:

- ISO 9000:2015 (fondamenti e glossario)
    - Modello di qualità neutro rispetto al dominio
- ISO 9001:2015 (sistema qualità – requisiti)
    - La visione ISO 9000 calata nei sistemi produttivi
    - ISO/IEC/IEEE 90003:2018 (ISO 9001:2015 applicato a prodotti SW)
- ISO 9004:2018 (qualità organizzativa - autovalutazione)

Il way of working determina in modo fattuale il Sistema Qualità, per cui la qualità si persegue come ideale (parte alta dello schema, "vogliamo lavorare bene", cioè se ho bilancio positivo sulle attività fatte) che viene realizzato modularmente dai compiti svolti.

Lavorare bene costa di più inizialmente (*ramp up*), ma significa pagare meno poi, producendo una cosa buona/migliore.



La qualità si misura e si valuta, mirando a proprietà specifiche. Esistono vari modelli:

- SW Process Assessment & Improvement (SPY per Tullio, SPAI per il resto del mondo)
    - Approccio sistematico alla valutazione e al miglioramento dei processi utilizzati per sviluppare, mantenere e distribuire il software.
    - L'obiettivo di SPAI è identificare le aree di miglioramento e implementare i cambiamenti che aumenteranno l'efficienza e l'efficacia del processo di sviluppo del software.

Ogni processo è sottoposto ad alcune procedure:

1) Identificare i processi da valutare
2) Valutare lo stato attuale dei processi
3) Identificare le aree di miglioramento
4) Implementare le modifiche
5) Valutare l'impatto delle modifiche

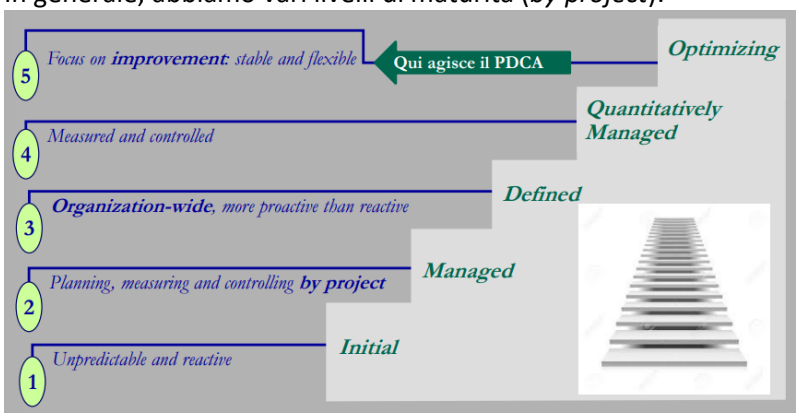
- CMM (Capability Maturity Model, 1987)
    - Un metodo per migliorare la maturità del processo di sviluppo del software di un'organizzazione sviluppato dalla Carnegie Mellon University come metodo per misurare e migliorare i processi utilizzati per sviluppare il software

    - Si basa su:
        - *Capability*: si occupa di capire quanto un processo sia capace di raggiungere un risultato atteso, ragionando in termini di efficienza ed efficacia. Si vuole che il suo livello sia il più alto possibile (garantisce che il processo venga eseguito da tutti in modo disciplinato, sistematico e quantificabile)
        - *Maturity*: coinvolge invece un insieme di processi e misura quanto è governato il loro sistema ed è il risultato della combinazione della capability dei singoli processi
        - *Governance*: misura della reattività dei processi di un'organizzazione ed essa ha come fine l'innalzamento del livello di maturity

Venne poi esteso al service management come CMMI (Capability Maturity Model Integration):
- Il punto focale di questo approccio per il miglioramento dei processi è l'integrazione delle funzioni organizzative, prima divise, al fine di fornire linee guida per la definizione di obiettivi, per le loro priorità e per la valutazione dei processi attuali.

- CMMI integra molteplici modelli CMM sviluppati per settori distinti. I livelli individuati riguardano la maturità dei processi aziendali, da quello iniziale (livello 1) a quello di ottimizzazione (livello 5):
    1) Initial: I processi dell'organizzazione sono ad hoc e caotici
    2) Managed: L'organizzazione ha stabilito processi di base che sono ripetibili, ma non sono ben controllati.
    3) Defined: I processi dell'organizzazione sono documentati e standardizzati e vengono gestiti utilizzando dati quantitativi.
    4) Quantitatively managed: I processi dell'organizzazione sono misurati, controllati e ottimizzati utilizzando tecniche statistiche e altre tecniche quantitative.
    5) Optimized: I processi dell'organizzazione vengono continuamente migliorati sulla base di un feedback basato sui dati.

In generale, abbiamo vari livelli di maturità (*by project*):



Di fatto, all'inizio il progetto è imprevedibile e reattivo, poi viene gestito e pianificato in modo collaborativo dall'azienda, successivamente misurato e viene attuato un miglioramento, sulla base del PDCA.

Per analogia, diamo un esempio:

Orientarsi in terreno sconosciuto		
5	***Focus on improvement*** : ho anche informazioni dinamiche sulle congestioni (ottimizzare il percorso scegliendolo a seconda della situazione)	 
4	***Measured and controlled*** : la cartina stradale è arricchita di indicazioni numeriche precise sulle distanze (gestire il viaggio quantitativamente)	 
3	***Organization-wide*** : dispongo di una cartina stradale (corrispondente alla mappa dei processi condivisa a livello di organizzazione)	
2	***By project*** : la persona cui chiedo potrebbe fornirmi indicazioni precise con riferimenti (sapendo mentre avanzo se sono sulla strada giusta), ma anche no	 
1	***Unpredictable and reactive*** : chiedo a qualcuno, che mi fornisce indicazioni approssimative (magari arrivo; più probabilmente mi perdo)	 

Si cita la norma ISO/IEC 33020:2019 – Process Assessment (https://cdn.standards.iteh.ai/samples/78526/e84f5951f904440092d79e0e881c1122/ISO-IEC-33020-2019.pdf)

Esso è uno standard per la valutazione dei processi nel campo dell'ingegneria del software. Fornisce indicazioni su come condurre una valutazione dei processi di sviluppo del software di un'organizzazione, con l'obiettivo di identificare le aree di miglioramento e implementare le modifiche che aumenteranno l'efficienza e l'efficacia dei processi.

Lo standard copre le seguenti aree di valutazione dei processi:

- Pianificazione e preparazione
    - Questo include la determinazione dell'ambito della valutazione, la selezione di un modello di valutazione dei processi adeguato e l'identificazione delle risorse e del personale necessari per condurre la valutazione.
- Conduzione della valutazione
    - Si tratta di raccogliere dati sui processi, analizzarli per identificare i punti di forza e di debolezza e confrontarli con le best practice del settore.
- Riferimento dei risultati
    - Si tratta di documentare i risultati della valutazione e di fornire raccomandazioni per il miglioramento.
- Utilizzo dei risultati della valutazione
    - Si tratta di implementare le modifiche basate sui risultati della valutazione e di valutare l'impatto delle modifiche sull'efficienza e l'efficacia dei processi.

Lo standard è destinato alle organizzazioni che desiderano migliorare la qualità e l'efficienza dei processi di sviluppo del software. Può essere utilizzato insieme ad altri modelli di valutazione dei processi, come il Capability Maturity Model.

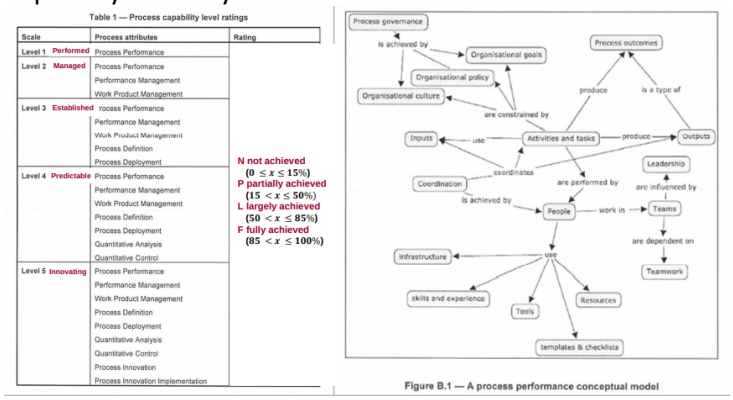


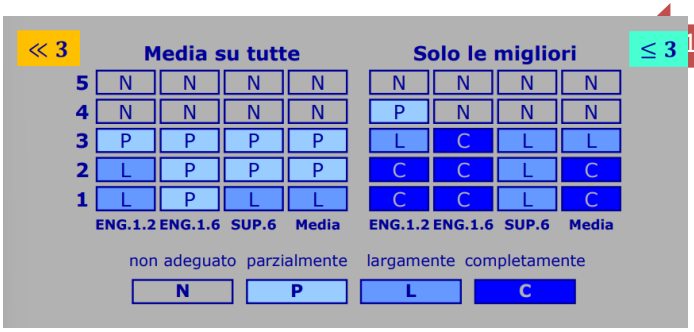
Table 1 — Process capability level ratings

Scale	Process attributes	Rating
Level 1 Performed	Process Performance	
Level 2 Managed	Process Performance Performance Management Work Product Management	
Level 3 Established	rocess Performance Performance Management Work Product Management Process Definition Process Deployment	
Level 4 Predictable	Process Performance Performance Management Work Product Management Process Definition Process Deployment Quantitative Analysis Quantitative Control	
Level 5 Innovating	Process Performance Performance Management Work Product Management Process Definition Process Deployment Quantitative Analysis Quantitative Control Process Innovation Process Innovation Implementation	

N not achieved ($0 \leq x \leq 15\%$)
P partially achieved ($15 < x \leq 50\%$)
L largely achieved ($50 < x \leq 85\%$)
F fully achieved ($85 < x \leq 100\%$)

Figure B.1 — A process performance conceptual model

In media, le indagini sulle aziende (2005), riporta che la metà sia tra livello 1 e 2, il 30% sia a livello 3 e una minoranza tra livello 4 e 5. Una significativa riduzione di tempo deriva dall'usare asset ed esperienza esistente.



- *SPICE (Software Process Improvement Capability dEtermination, 1992)*
    - Modello che esclude la maturità e valuta a grana fine la capability dei processi
    - Si trattano varie aree di processo: rapporto cliente-fornitore, ingegneristica, di supporto, gestione e organizzazione
    - Vi sono in particolare 6 livelli di valutazione (riferimento Sweky):
        - Incomplete process
            - Livello in cui non vi è nessun tipo di indicatore, indica che un processo non è implementato oppure è fallito, cioè non ha prodotto nessun risultato.
        - Performed process
            - Livello in cui il processo è stato attuato e adempie all'obiettivo prefissato.
        - Managed process
            - Livello in cui il processo che già adempie ai suoi obiettivi presenta dei prodotti controllati e mantenuti, attività pianificate e controllate, documentate nello svolgimento.
        - Established process
            - Livello in cui il processo è ora ad un livello tale da garantire la produzione di prodotti adatti.
        - Predictable process
            - Livello in cui il processo viene eseguito con limiti e obiettivi di produzione definiti.
        - Optimizing process
            - Livello in cui il processo è continuamente migliorato per soddisfare gli obiettivi di business attuali e pianificati.

Garantire l'idoneità a tutte le applicazioni e a tutte le dimensioni delle organizzazioni.
Anche in Italia si è condotta questa ricerca; di fatto, per aziende di perlopiù piccole dimensioni, si è evidenziata una visione limitata della qualità (con pochi certificati standard, e la crescita/qualità viste come obiettivi secondo o come risposta a clienti/alla concorrenza).

Verifica e validazione: Introduzione (Vardanega)

(Eventuali approfondimenti:
https://martinfowler.com/articles/mocksArentStubs.html
https://www.math.unipd.it/~tullio/IS-1/2004/Approfondimenti/Inspections-Walkthroughs.pdf
http://www.testingstandards.co.uk/Component%20Testing.pdf)

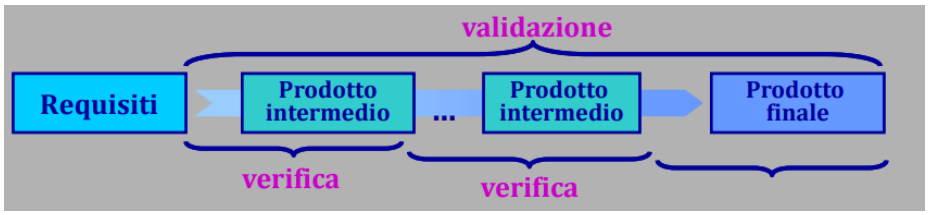
Il fine della verifica è aiutare la validazione e quest'ultima ha esito definitivo (guarda *solo al prodotto*, cioè guarda all'origine/needs, vedendo quanto corrisponde a questo), dando una risposta binaria (sì-no); secondo lo standard ISO/IEC 12207 le definizioni sono le seguenti:

- Verifica software (*Software verification*)
  - Fornisce prove oggettive del fatto che i risultati di *un particolare segmento* del ciclo di vita (fase) dello sviluppo del software soddisfa tutti i requisiti specificati
  - Cerca consistenza (prove concrete della validità), completezza (un requisito utente è soddisfatto da *n* requisiti software) e correttezza di tali risultati (quindi, pur cambiando le baseline, il risultato rimane corretto)
  - *Fornisce supporto per la successiva conclusione che il software è convalidato, accertandosi non siano stati introdotti errori*

- Validazione software (*Software validation*)
  - Conferma mediante esame e presentazione di prove oggettive che:
    - le specifiche SW sono conformi alle esigenze dell'utente e agli usi previsti
    - i requisiti implementati tramite SW possono essere costantemente (e sufficientemente) soddisfatti
      - Questo si dimostra con il *tracciamento* (il requisito X dimostra la fattibilità di uno scenario Y)
  - La validazione non si applica ad un particolare segmento, ma dovrebbe essere una *self-fulfilling prophecy/profezia che si autoavvera*.

La dimostrazione viene fatta *consistentemente*, al fine di *rendere usabile* il prodotto.

1) La *verifica* è un'azione fatta più volte che permette di dare certezza e preparare il successo della validazione (senza affacciarsi alla validazione quando non si ha sicurezza). Essa accerta che l'esecuzione delle attività attuate nel periodo in esame non abbia introdotto errori, approvando le *baseline* associate alle *milestone* di progetto.
2) La *validazione* effettua una consegna *che fa le cose attese* (sulla base di quello che vuole l'utente).

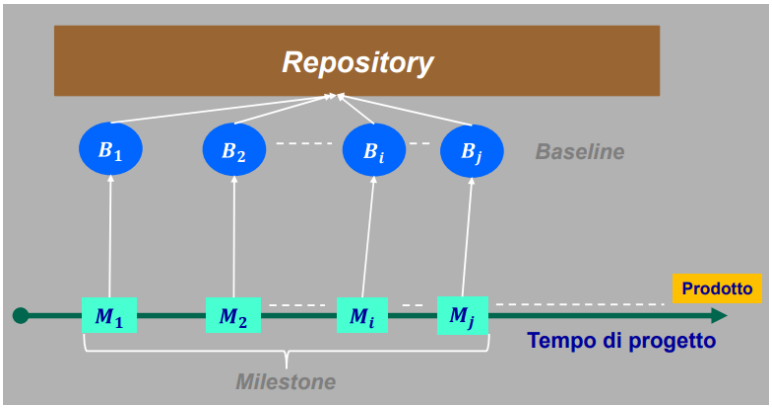


A glossario, individuiamo che:

- Milestone
  - Una data del calendario di progetto, che denota un punto di avanzamento atteso, sostanziato da una baseline corrispondente
  - Il loro numero è inizialmente abbastanza grande, in quanto non si hanno certezze, e capisco progressivamente come posizionarle. Non esiste un numero fisso; alcuni ne piazzano pochi, altri molti. Tutto dipende dal tipo di attività progettuale; ha senso averne tante se ci sono tante date che testimoniano avanzamenti *importanti* nel progetto.
  - A livello di tempo, arrivano prima le milestone (cioè le attese)

- Baseline
  - Un insieme di parti poste sotto configurazione (CI – Configuration Item), che realizzano uno specifico avanzamento di prodotto
  - Esse sostanziano le milestone, in quanto sono oggettive e materiale utile per la verifica
  - La compilazione ha senso se sostanzia dei requisiti utili
  - Possono essere tante a piacere; ogni qual volta cambia la baseline, si ha la verification (i cambiamenti avvengono solo a verifica con esito positivo del lavoro svolto, con evidenza oggettiva di aver fatto bene)
    - Quando tutti gli user needs sono soddisfatti, si effettua il collaudo
  - La verification viene fatta ogni volta prima di una baseline; in questo modo, sappiamo che viene fatta nel modo giusto. Questa non necessariamente *richiede esecuzione*, perché arriverebbe solo nella fase di implementation.

Dunque, il prodotto di progetto è un aggregato di SW e di documentazione. Nel progetto, la baseline è solida quando sostanziata da una repo. La riproducibilità viene garantita dalla repo, in cui sono tutte le parti e i modi per montarle (oggetti riproducibili). È molto più facile prendere tutto per piccole parti piuttosto che in modo monolitico. Quindi: milestone → baseline → repository.



- Analisi statica
  - Non richiede esecuzione dell'oggetto di verifica; per questo motivo, è applicabile a ogni prodotto di processo (e quindi a tuti i processi attivati nel progetto)
    - Per fare ciò occorrono strumenti automatici che lo facciano in modo ripetibile e proceduralizzabile
  - Studia documentazione e codice (sorgente, oggetto)
    - Tende ad assicurare *comportamenti predicibili*, affinché documentazione e codice siano il meno ambigui possibile
    - Ciò dipende anche dalle scelte di strutturazione del codice, tenendo conto che maggiore la dimensione del contesto d'uso, minore la verificabilità
  - Può usare metodi di lettura (desk check), impiegati solo per prodotti semplici
    - Letteralmente analizza il codice linea per linea identificando problemi, bug, errori nel codice; viene fatto da un singolo o da un insieme di sviluppatori
  - Può usare metodi più formali (basati su prove di proprietà, quando la dimostrazione vera costa troppo)
  - Accerta conformità a regole, assenza di difetti, presenza di proprietà desiderate

I metodi di lettura dell'analisi statica sono:

- <u>walkthrough</u>
  - ricerca a pettine scovando errori; essa è costosa e lavora in modo incerto, a grana grossa;
- <u>inspection</u>
  - procedura automatizzabile, tramite una checklist, svolte tramite studio dell'oggetto di verifica, con lettura umana o automatizzata.

L'efficacia di entrambi i metodi è dipendente dall'esperienza dei verificatori, in particolare nell'organizzare le attività da svolgere e nel documentare le risultanze (usano modalità tra di loro complementari).

Possiamo dire che:

- La *walkthrough* ha l'obiettivo di rilevare la presenza di difetti attraverso lettura critica ad ampio spettro del prodotto in esame. Essa usa gruppi misti di verificatori/sviluppatori. In particolare, il codice viene esaminato percorrendolo e simulandone possibili esecuzioni, mentre per i documenti occorre studiarne ogni parte. Prima di eseguire la lettura il Walkthrough va pianificato e inseguito alla lettura va discusso quanto individuato per correggere i difetti, documentando le attività svolte.

- La *inspection* rileva la presenza di difetti, eseguendo lettura mirata dell'oggetto di verifica ed è basata sul controllo di punti critici. La sua ricerca è focalizzata su presupposti (*error guessing*) e, anche qui, ogni passo documenta attività svolte e risultanze. Prima di procedere con la lettura, viene definita una checklist che può derivare anche dalla walkthrough, oltre che da altre conoscenza, risultando complementare con il precedente.

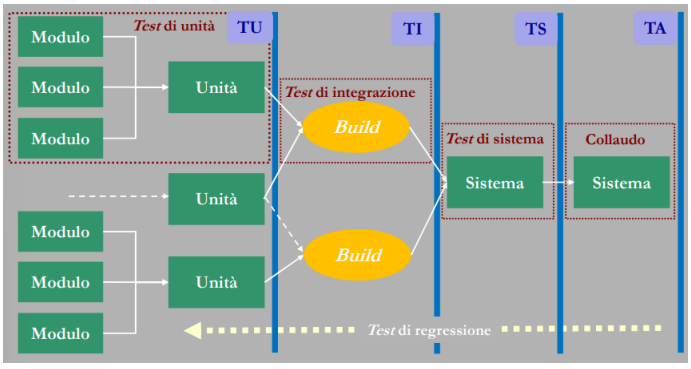
Le due metodologie sono entrambe dei controlli di tipo desk check, documentati formalmente e che coinvolgono programmatori e verificatori. Per contro, walkthrough richiede maggiore attenzione e un lavoro collaborativo, mentre inspection risulta più rapido ma si basa su errori presupposti e dunque può rilevare solo alcuni errori.

Nell'analisi dinamica, i test sono domande fissate, devono essere molti e vengono fatti ogni volta che si ha un upload sulla repo e in particolare:

- I test devono essere ripetibili (*pongono la stessa domanda nelle stesse identiche condizioni e sono considerate affidabili*) e per questo specificano:
  - Ambiente d'esecuzione: HW/SW, stato iniziale
  - Attese: ingressi richiesti, uscite ed effetti attesi
  - Procedure: esecuzione, analisi dei risultati

- I test vanno automatizzati e, per questo motivo, usano strumentazione:
  - *Driver*
    - Componente attiva fittizia per pilotare il test (essa chiama una procedura) ai livelli inferiori del codice, quando il livello superiore non è sviluppato
  - *Stub (calco)*
    - Componente passiva fittizia per simulare parti del sistema utili al test ma non oggetto di test; viene intesa come stampo, essendo routine di test su un sistema remoto e intesa come implementazione predefinita di un'interfaccia o di un'implementazione
    - Differisce dal cosiddetto mock per il fatto che quest'ultimo simula il comportamento di un singolo oggetto, ma lo stub è più un "template", dunque si adatta coma calco a base per altri test
  - *Logger*
    - Componente non intrusivo di registrazione dei dati di esecuzione per analisi

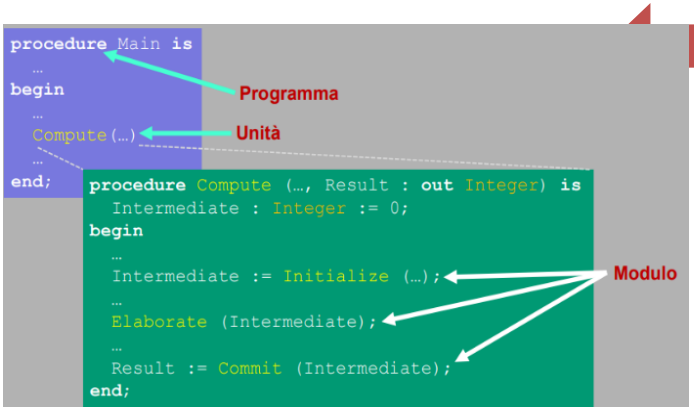
La verifica si occupa di montare cose buone e valide e agisce con i test a tutti i livelli: la parte più piccola è quello di *unità*, la parte più grande è quello di *sistema*, ciò che sta in mezzo è di *integrazione*.
Ad alto livello tutto ciò viene rappresentato come a lato.



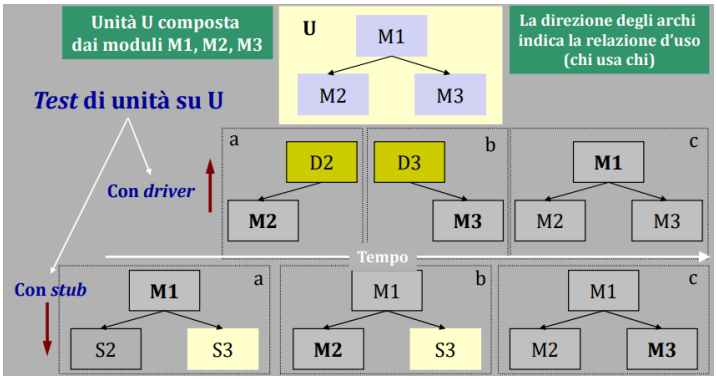
I test devono essere *solidamente* buoni, mettendo insieme test su parti piccole (che hanno passato il loro test). Essi vengono decisi nella fase di design.

A livello di glossario, parliamo di unità, quindi la più piccola quantità di SW che sia utilmente sottoponibile a verifica individuale, tipicamente prodotta da un singolo programmatore.

Questa viene intesa in senso architetturale: non linee di codice ma entità di organizzazione logica (procedura/classe/aggregato-package).



In particolare, vediamo il *modulo* come frazione di unità (come normalmente accade nei linguaggi di programmazione) e il *componente* è una parte che svolge una specifica funzione, fatta di unità, a loro volta fatte da moduli.
Per agganciarci a quanto detto sopra, abbiamo che le unità sono composte da alcuni moduli, con relazioni d'uso. Il driver testa i livelli dal basso verso l'alto (bottom-up), mentre gli stub testano varie funzionalità all'opposto, dall'alto verso il basso (top-down).



Design Pattern Comportamentali (Cardin)

(Riferimenti di codice:
https://github.com/rcardin/swe/tree/master/src/main/java/it/unipd/math/swe/patterns)

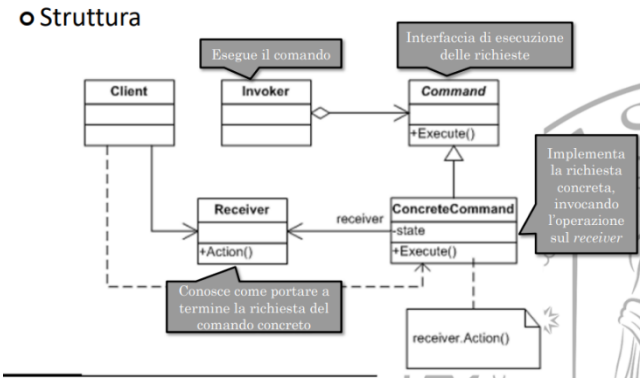
I design pattern comportamentali sono un tipo di pattern di progettazione che si concentra sulla *comunicazione* e *sull'interazione* tra gli oggetti, piuttosto che sulla struttura e sul comportamento dei singoli oggetti. Questi pattern mirano a migliorare la *flessibilità* e il riutilizzo di un progetto orientato agli oggetti, definendo il modo in cui gli oggetti devono comunicare e interagire tra loro per realizzare un compito specifico. Essi rispondono a due domande specifiche:

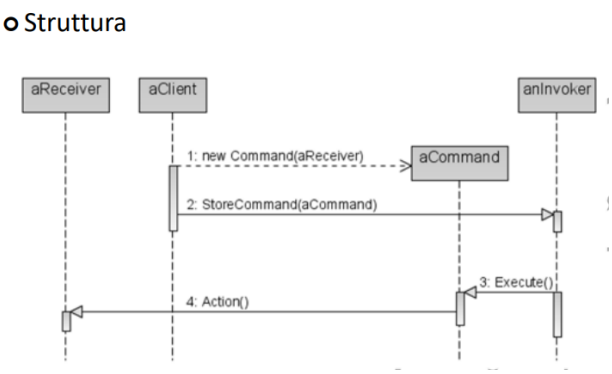
- In che modo un oggetto svolge la sua funzione?
- In che modo diversi oggetti collaborano tra loro?

Il primo pattern di questi che vediamo è il Command Pattern, che è un modello di progettazione comportamentale che incapsula una richiesta come un oggetto, consentendo di parametrizzare i client con richieste diverse (così i client sono indipendenti dalle richieste), di accodare o registrare le richieste e di supportare operazioni di annullamento/ripristino.

Questa permette di chiamare delle sottoclassi senza averci a che fare (realizzando disaccoppiamento); ogni istanza delle sottoclassi viene *ricevuta* (si chiamano Receiver) ed è tutto ciò di cui voglio fornire accesso tramite il comando (*Command*), realizzando un comando utile all'implementazione (*ConcreteCommand*).
Di fatto, è un'interfaccia utilizzata per eseguire una richiesta:

Scritto da Gabriel





Descriviamo i singoli pezzi:

- La classe *Invoker* (alias *Sender*) è responsabile dell'avvio delle richieste. Questa classe deve avere un campo per memorizzare un riferimento a un oggetto *Command*. Il mittente attiva il *Command*, invece di inviare la richiesta direttamente al destinatario. Si noti che il mittente non è responsabile della creazione dell'oggetto *Command* (infatti, riceve un comando precostituito dal client tramite il costruttore).

- L'interfaccia *Command* definisce il metodo per eseguire una richiesta. Ha un singolo metodo *execute*, che viene chiamato dall'oggetto *Invoker* per eseguire la richiesta.

- Gli oggetti *ConcreteCommand* sono implementazioni concrete dell'interfaccia *Command* che definiscono la richiesta specifica che rappresentano. Hanno un campo per memorizzare un riferimento all'oggetto *Receiver*, che è l'oggetto che eseguirà la richiesta, e implementano il metodo execute richiamando il metodo appropriato sull'oggetto *Receiver*.

- La classe *Receiver* è l'oggetto che esegue la richiesta. Dispone delle conoscenze e delle risorse necessarie per eseguire la richiesta e viene chiamato dall'oggetto *ConcreteCommand* quando la richiesta viene eseguita.

- Il *Client* è l'oggetto che avvia una richiesta. Crea un oggetto *ConcreteCommand* e ne imposta il destinatario, che è l'oggetto che eseguirà la richiesta. L'oggetto Client passa quindi l'oggetto *ConcreteCommand* all'oggetto *Invoker*, che esegue la richiesta.

Dal punto di vista di applicabilità:

- Parametrizzazione di oggetti sull'azione da eseguire
  - *Callback function* (funzione passata in un'altra funzione come argomento, che viene poi invocata all'interno della funzione esterna per completare un qualche tipo di routine o azione)
- Composizione di comandi (*composite command*)
  - Questo permette l'esecuzione atomica (*transazionale*) o su pezzi (*partizionate*), per poterne prevedere l'esecuzione (*schedulazione*)
- Specificare, accodare ed eseguire richieste molteplici volte senza conoscere a priori le operazioni che può fare (disaccoppiamento)
  - Può anche permettere l'esecuzione differita delle richieste in questo modo
- Implementazione di operazioni reversibili
- Implementazione di macro che eseguono una serie di comandi semplici
- Binding tra Receiver e azioni da eseguire, con comandi autoconsistenti
  - Utilizzo di Template e/o Generics
- Supporto ad operazioni di *Undo* e *Redo* (facendo attenzione allo stato del sistema da mantenere)

Scritto da Gabriel

- Supporto a transazione
    - Un comando equivale ad una operazione atomica

Conseguenze:

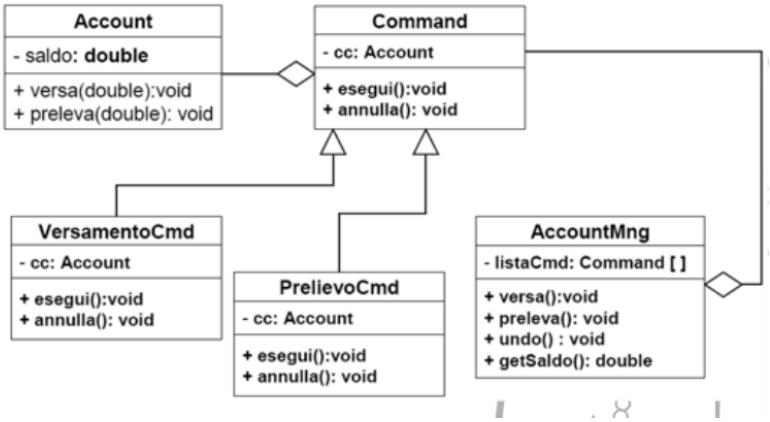
- Accoppiamento “lasco” tra oggetto invocante e quello che porta a termine l’operazione
- Accumulo di errori durante l’esecuzione di più comandi
- I comandi possono essere estesi
- I comandi possono essere composti (*composite command*) e innestati
- È facile aggiungere nuovi comandi
- Le classi esistenti non devono essere modificate
- Può rendere difficile la comprensione del programma, avendo un flusso di controllo distribuito tra oggetti

Esempio

Una classe Account modella conti correnti. Le funzionalità che si vogliono realizzare sono:

- Prelievo
- Versamento
- Undo

Questa operazione consente di annullare una delle precedenti, ma con il vincolo che l’annullamento deve avvenire con ordine cronologico inverso.



Vediamo un’implementazione di Command con Python:

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class Light:
    def turn_on(self):
        print("Turning on the light")

    def turn_off(self):
        print("Turning off the light")

class LightOnCommand(Command):
    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self._light = light
```

In questo esempio, la classe *Light* è una classe semplice che rappresenta una luce e dispone di metodi per accenderla e spegnerla. Le classi *LightOnCommand* e *LightOffCommand* sono implementazioni concrete dell'interfaccia Command, che definiscono la richiesta specifica che rappresentano.

Hanno un campo per memorizzare un riferimento all'oggetto *Light* e implementano il metodo execute richiamando il metodo appropriato sull'oggetto *Light*.

La classe *RemoteControl* è un oggetto *Invoker* che contiene un riferimento a un oggetto *Command* e invoca il metodo *execute* sull'oggetto Command quando viene chiamato il metodo *press_button*.

L'oggetto *Client* crea gli oggetti *LightOnCommand* e *LightOffCommand* e li passa all'oggetto *RemoteControl* per eseguire le richieste. L'oggetto *RemoteControl* non deve sapere come vengono eseguite le richieste, in quanto è astratto dall'interfaccia *Command*.

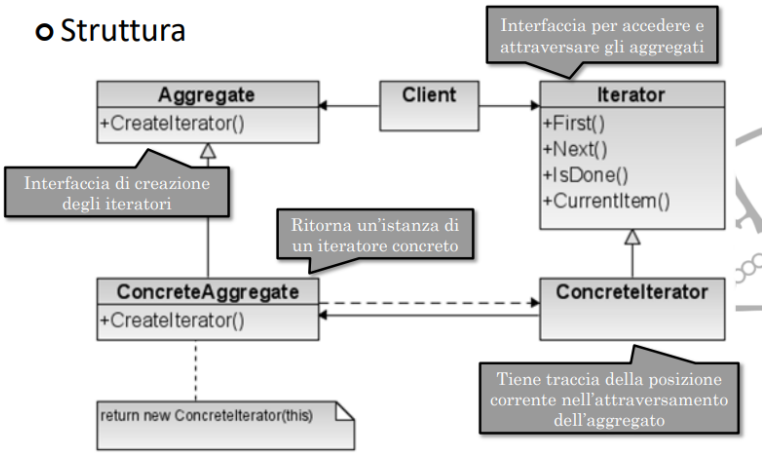
Scritto da Gabriel

```
    def execute(self):
        self._light.turn_off()

class RemoteControl:
    def set_command(self, command):
        self._command = command

    def press_button(self):
        self._command.execute()

light = Light()
light_on_command = LightOnCommand(light)
light_off_command = LightOffCommand(light)

remote = RemoteControl()
remote.set_command(light_on_command)
remote.press_button()  # prints "Turning on the light"

remote.set_command(light_off_command)
remote.press_button()  # prints "Turning off the light"
```

Introduciamo invece l'Iterator Pattern quando vogliamo inserire funzionalità/politiche di scorrimento senza modificare l'aggregato/struttura stesso/a. Devono essere disponibili diverse politiche di attraversamento e sposta la responsabilità di attraversamento in un oggetto iteratore (tiene traccia dell'elemento corrente). In particolare, si vuole avere un modo per attraversare ciascun elemento senza ripetere ogni volta gli stessi elementi, soprattutto quando si ha a che fare con strutture dati complesse (e questo peggiora l'efficienza).

Dal punto di vista di applicabilità:

- Accedere il contenuto di un aggregato senza esporre la rappresentazione interna (non per forza in modo sequenziale, ma usando un oggetto *next*)
- Supportare diverse politiche di attraversamento
- Fornire un'interfaccia unica di attraversamento su diversi aggregati
    - Polymorphic iteration



Per aggiungere funzionalità di iterazione, avremo vari *ConcreteIterator*; di fatto ogni iteratore dovrà accedere alle proprietà interne dell'aggregato (implementando varie politiche di aggregazione).

Descrivendo la composizione:

- L'interfaccia *Iterator* dichiara le operazioni necessarie per l'attraversamento di un insieme: recupero dell'elemento successivo, recupero della posizione corrente, riavvio dell'iterazione, ecc.

- I *Concrete Iterator* implementano algoritmi specifici per l'attraversamento di un insieme. L'oggetto iteratore deve tenere traccia del progresso dell'attraversamento per conto proprio. Ciò consente a diversi iteratori di attraversare lo stesso insieme indipendentemente l'uno dall'altro.

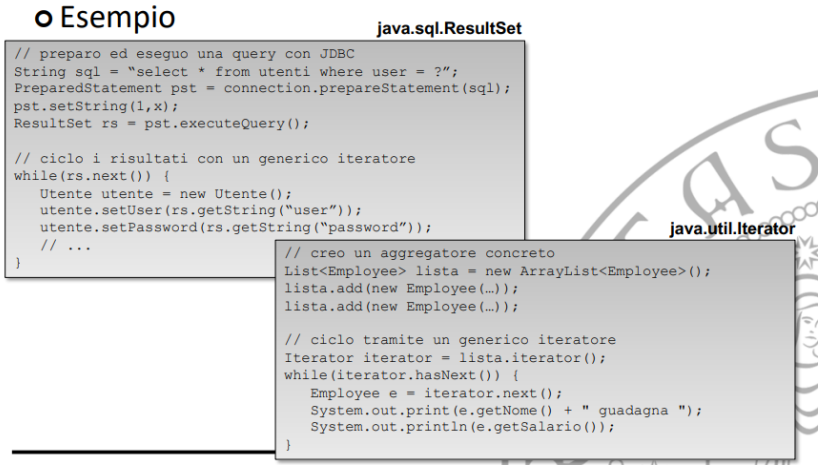
- L'interfaccia *Collection* dichiara uno o più metodi per ottenere iteratori compatibili con la collezione. Si noti che il tipo di ritorno dei metodi deve essere dichiarato come interfaccia iteratore, in modo che le collezioni concrete possano restituire vari tipi di iteratori.

- Le *Concrete Collection* restituiscono nuove istanze di una particolare classe di iteratori concreti ogni volta che il client ne richiede uno. Il resto del codice della collezione, normalmente, si trova nella classe stessa.

- Il *Client* lavora sia con le collezioni che con gli iteratori tramite le loro interfacce. In questo modo, il client non è legato a classi concrete, consentendo di utilizzare diverse collezioni e iteratori con lo stesso codice client. In genere, i *client* non creano iteratori per conto proprio, ma li ottengono dalle collezioni (in alcuni casi, possono crearne di propri, nel caso di iteratori speciali)

Conseguenze:

- Supporto a variazioni nelle politiche di attraversamento di un aggregato
- Coesione nell'uso dell'aggregato e dell'iteratore (dipendenza buona)
- Semplificazione dell'interfaccia dell'aggregato
- Riduzione della duplicazione del codice di attraversamento
- Attraversamento contemporaneo di più iteratori sul medesimo aggregato
- Attraversamento di una struttura di dati complessa, ma che si vuole nascondere ai client (per comodità o per motivi di sicurezza)



Nell'esempio qui a lato realizzato con J2SE di Java (per applicazioni portable), preparo ed eseguo una query usando un iteratore che, genericamente, attraversa una lista, a prescindere dai suoi dettagli.

Implementazione:

- Chi controlla l'iterazione?
  - *External (active) iterator*: il client controlla l'iterazione
  - *Internal (passive) iterator*: l'iteratore controlla l'iterazione
    - Programmazione dichiarativa (non stiamo dicendo *in che modo* dobbiamo scorrere, ma *cosa* dobbiamo scorrere); se fosse imperativa, direi *come* e anche *che cosa*
    - Esempio di passive iterator → gli stream

- Chi definisce l'algoritmo di attraversamento?
  - Aggregato/Struttura dati: iteratore viene definito "cursore"
    - Il client invoca *Next* sull'aggregato, fornendo il cursore
  - Iteratore: viene violata *l'encapsulation* dell'aggregato (come detto, deve per forza accedere allo stato interno dell'aggregato)
- Iteratori robusti
  - Assicurarsi che l'inserimento e la cancellazione di elementi dall'aggregato non creino interferenze/alias (nel caso di Java ad esempio, modificare la collezione durante l'iterazione può sollevare eccezione)
- *Polymorphic iterator*
  - Sfruttando le capacità polimorfiche di alcuni linguaggi di programmazione è possibile definire un'unica interfaccia uniforme per tutti i tipi di oggetti iteratore. Questo permette di scrivere programmi in cui l'accesso ai dati contenuti da un aggregato è completamente indipendente dalla struttura dell'aggregato stesso, purché esista una concreta implementazione dell'oggetto iteratore per una data.
  - Utilizzo del *Proxy Pattern* per deallocazione dell'iteratore
- Accoppiamento stretto tra iteratore e aggregato
  - Questo accade quando non si ha polimorfismo, quindi necessariamente, l'iteratore deve conoscere vari dettagli implementativi del client
  - In C++, un modo è dichiarare *friend* l'iteratore
- *Null iterator*
  - Iteratore degenere che implementa *IsDone* con il ritorno di *true*
  - Utile per scorrere strutture ricorsive

Vediamo un esempio di Iterator in Python:

```
from abc import ABC, abstractmethod

class Iterator(ABC):
    @abstractmethod
    def has_next(self) -> bool:
        pass

    @abstractmethod
    def next(self) -> object:
        pass

class ConcreteIterator(Iterator):
    def __init__(self, aggregate):
        self._index = 0
        self._aggregate = aggregate

    def has_next(self) -> bool:
        return self._index < len(self._aggregate)

    def next(self) -> object:
        result = self._aggregate[self._index]
        self._index += 1
        return result

class Aggregate(ABC):
    @abstractmethod
    def create_iterator(self) -> Iterator:
```

In questo esempio, l'interfaccia Iterator definisce i metodi per accedere agli elementi di un oggetto aggregato in modo sequenziale. La classe *ConcreteIterator* è un'implementazione concreta dell'interfaccia Iterator che mantiene una posizione corrente e fornisce accesso agli elementi di un oggetto *ConcreteAggregate*.

L'interfaccia *Aggregate* definisce il metodo per creare un oggetto Iterator. La classe *ConcreteAggregate* è un'implementazione concreta dell'interfaccia *Aggregate* che dispone di un elenco di elementi e fornisce un metodo per creare un oggetto Iterator per accedere a tali elementi.

Il codice client crea un oggetto *ConcreteAggregate* e poi usa il suo metodo *create_iterator* per creare un oggetto Iterator per accedere agli elementi dell'aggregato. Quindi utilizza i metodi *has_next* e *next* dell'oggetto Iterator per accedere agli elementi dell'aggregato.

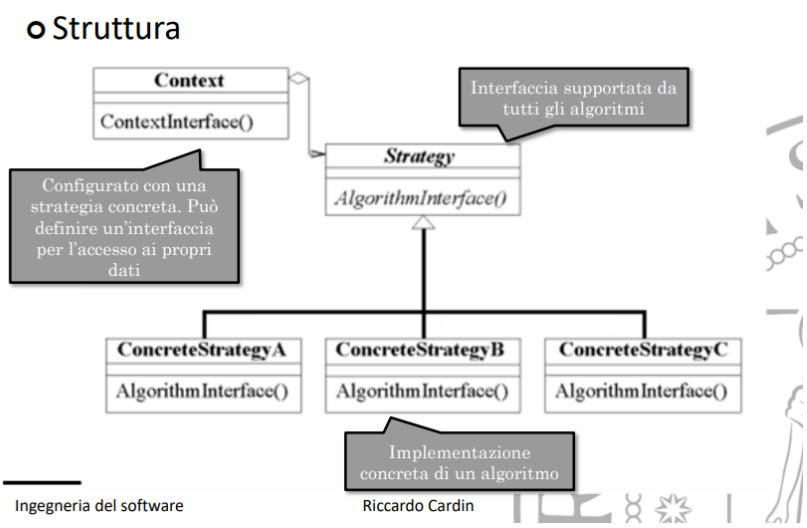
```
        pass

class ConcreteAggregate(Aggregate):
    def __init__(self):
        self._items = [1, 2, 3, 4, 5]

    def create_iterator(self) -> Iterator:
        return ConcreteIterator(self._items)

aggregate = ConcreteAggregate()
iterator = aggregate.create_iterator()

while iterator.has_next():
    item = iterator.next()
    print(item)  # prints 1 2 3 4 5
```

Successivamente vediamo lo Strategy Pattern, che è un modello di progettazione comportamentale che definisce una famiglia di algoritmi, incapsula ciascuno di essi e li rende intercambiabili. Consente di selezionare un algoritmo in fase di esecuzione e dà al codice la flessibilità necessaria per essere utilizzato con qualsiasi algoritmo della famiglia.

Prendiamo l'esempio del gioco degli scacchi: avremo vari livelli di difficoltà, ma "esternamente", il gioco rimane lo stesso. Esistono differenti algoritmi (strategie) che non possono essere inserite direttamente nel client, ma richiedono un'implementazione coerente in quanto devono farne parte. Questo può generare client ragionevolmente complessi (dato che richiede di aggiungere nuovi algoritmi e modificare quelli esistenti).

Il pattern Strategy suggerisce di prendere una classe che fa qualcosa di specifico in molti modi diversi e di estrarre tutti questi algoritmi in classi separate, chiamate *strategie/strategies* (e strategy al singolare).
La classe originale, chiamata *context*, deve avere un campo per memorizzare un riferimento a una delle strategie e delega il lavoro a un oggetto strategy collegato.

Il context non è responsabile della selezione di un algoritmo, ma è il client che passa al context la strategia desiderata. Lavora con tutte le strategie attraverso la stessa interfaccia generica, che espone solo un singolo metodo per attivare l'algoritmo incapsulato nella strategia selezionata.
In questo modo il context diventa indipendente dalle strategie concrete, per cui è possibile aggiungere nuovi algoritmi o modificare quelli esistenti senza modificare il proprio codice o quello di altre strategie.



L'implementazione concreta è utile per semplificare tanti *if-switch case* tale che, *con un'interfaccia unica*, si lascia una "scelta" su *quale strategia adottare*.

Scritto da Gabriel

- Il *Context* mantiene un riferimento a una delle *Strategy* concrete e comunica con questo oggetto solo attraverso l'interfaccia della *Strategy*.

- L'interfaccia *Strategy* è comune a tutte le strategie concrete. Dichiara un metodo che il *Context* utilizza per eseguire una strategia.
- Le *Concrete Strategies* implementano diverse varianti di un algoritmo utilizzato dal contesto.

- Il *Context* chiama il metodo di esecuzione sull'oggetto *Strategy* collegato ogni volta che deve eseguire l'algoritmo. Il contesto non sa con quale tipo di strategia lavora o come viene eseguito l'algoritmo.

- Il *Client* crea un oggetto strategia specifico e lo passa al *Context*. Il *Context* espone un setter che consente ai client di sostituire la *Strategy* associata al contesto in fase di esecuzione.

Applicabilità:
- Diverse classi differiscono solo per il loro comportamento
    - Molto spesso, per varie strategie vi sono input diversi; questi possono essere passati in fase di costruzione. Questo è possibile creando un tipo astratto come input da concretizzare
- Si necessita di diverse varianti dello stesso algoritmo e spesso si vuole passare agilmente da un algoritmo ad un altro durante l'esecuzione
- Un algoritmo utilizza dati di cui i client non devono occuparsi
- Una classe definisce differenti comportamenti, tradotti in un serie di statement condizionali
- È possibile utilizzare diverse varianti di un algoritmo all'interno di un oggetto e passare da un algoritmo all'altro durante l'esecuzione
- Isolare la logica aziendale di una classe dai dettagli di implementazione degli algoritmi che potrebbero non essere così importanti nel contesto di tale logica

Conseguenze:
- Definizione di famiglie di algoritmi per il riuso del contesto
- Alternativa all'ereditarietà dei client
    - Evita di effettuare subclassing direttamente dei contesti
- Eliminazione degli statement condizionali
- Differenti implementazioni dello stesso comportamento
- I client a volte devono conoscere dettagli implementativi per poter selezionare il corretto algoritmo
- Comunicazione tra contesto e algoritmo
    - Alcuni algoritmi non utilizzano tutti gli input
- Incremento del numero di oggetti nell'applicazione
- Se si ha solamente una coppia di algoritmi che cambiano raramente, inutile complicare il programma con le Strategy
- Tanti linguaggi moderni permettono di evitare le classi extra/interfacce di Strategy

Nell'esempio dell'utilizzo del link detto (motori di ricerca), vogliamo *separare i contesti di motori di ricerca*; in questo modo, separiamo le famiglie differenziando la costruzione di oggetti *diversi per contesto*.
(nelle classi, *SearchEngine2016* non viene usato da *SearchContext2017*).

Implementazione:
- Definire le interfacce di strategie e contesti
    - Fornisce singolarmente i dati alle strategie
    - Fornire l'intero contesto alle strategie
    - Inserire un puntamento al contesto nelle strategie
- Implementazione strategie
    - C++: Template, Java: Generics

  - Solo se l'algoritmo può essere determinato a compile time e non può variare dinamicamente
- Utilizzo strategia opzionali
  - Definisce una strategia di default

Esempio

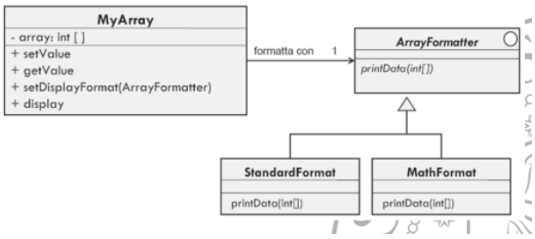
Esempio

Si vuole realizzare una classe `MyArray` per disporre di tutte le funzioni utili per lavorare con vettori di numeri. Si prevedono 2 funzioni di stampa:

- Formato matematico { 67, -9, 0, 4, ...}
- Formato standard `Arr[0] = 67 Arr[1] = -9 Arr[2] = 0 ...`

Questi formati potrebbero, in futuro, essere sostituiti o incrementati

Esempio



Diamo un esempio di Strategy in Python:

```
from abc import ABC, abstractmethod

class Strategy(ABC):
    @abstractmethod
    def do_algorithm(self, data: list) -> list:
        pass

class ConcreteStrategyA(Strategy):
    def do_algorithm(self, data: list) -> list:
        return sorted(data)

class ConcreteStrategyB(Strategy):
    def do_algorithm(self, data: list) -> list:
        return reversed(data)

class Context:
    def __init__(self, strategy: Strategy):
        self._strategy = strategy

    def set_strategy(self, strategy: Strategy):
        self._strategy = strategy

    def do_algorithm(self, data: list) -> list:
        return self._strategy.do_algorithm(data)

context = Context(ConcreteStrategyA())
print(context.do_algorithm([3, 2, 1]))  # prints [1, 2, 3]

context.set_strategy(ConcreteStrategyB())
print(context.do_algorithm([3, 2, 1]))  # prints [1, 2, 3][::-1]
```

In questo esempio, l'interfaccia *Strategy* definisce il metodo per eseguire un algoritmo. Le classi *ConcreteStrategyA* e *ConcreteStrategyB* sono implementazioni concrete dell'interfaccia *Strategy* che forniscono implementazioni specifiche dell'algoritmo.

L'oggetto *Context* contiene un riferimento a un oggetto *Strategy* e delega l'esecuzione dell'algoritmo all'oggetto *Strategy*. Ha un metodo set_strategy per cambiare l'oggetto *Strategy* in fase di esecuzione e un metodo *do_algorithm* per eseguire l'algoritmo.

Il codice *client* crea un oggetto *Context* con un oggetto *ConcreteStrategyA* e chiama il metodo *do_algorithm* per ordinare i dati. Successivamente, cambia l'oggetto *Strategy* in un oggetto *ConcreteStrategyB* e richiama il metodo *do_algorithm* per invertire i dati.

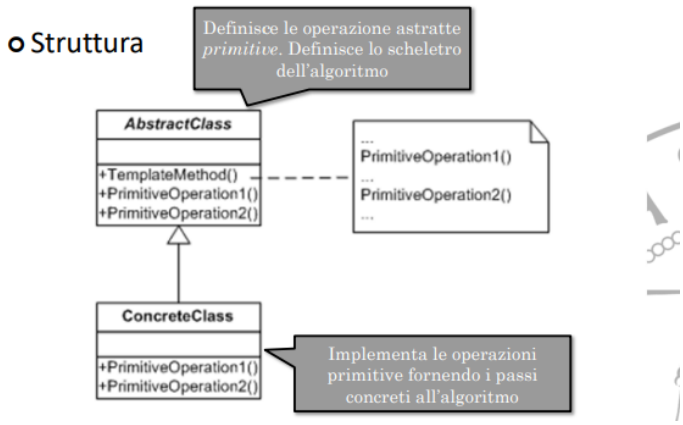
Abbiamo poi il Template Method, pattern di progettazione comportamentale che definisce lo scheletro di un algoritmo nella superclasse, ma consente alle sottoclassi di sovrascrivere passi specifici dell'algoritmo senza modificarne la struttura. L'obiettivo di definire un algoritmo in termini di operazioni astratte (viene fissato l'ordine delle operazioni); le sottoclassi forniscono il comportamento concreto.

Lo schema Template Method suggerisce di scomporre un algoritmo in una serie di passi, trasformare questi passi in metodi e inserire una serie di chiamate a questi metodi all'interno di un singolo metodo template. I passi possono essere astratti o avere un'implementazione predefinita. Per utilizzare l'algoritmo, il client deve fornire la propria sottoclasse, implementare tutti i passi astratti e sovrascrivere alcuni di quelli opzionali, se necessario (ma non il metodo template stesso).

Abbiamo due tipi di passi:
- i passi astratti devono essere implementati da ogni sottoclasse;
- i passi opzionali hanno già un'implementazione predefinita, ma possono essere sovrascritti se necessario.

Esiste un altro tipo di passo, chiamato *hook*. Un hook è un passo opzionale con un corpo vuoto. Di solito, gli hook sono collocati prima e dopo i passi cruciali degli algoritmi, fornendo alle sottoclassi ulteriori punti di estensione per un algoritmo.

Come per il Singleton, Template può diventare un *antipattern*, in quanto rompe l'encapsulation e, a sua volta, accoppia strettamente l'implementazione di un metodo ai metodi della classe figlia.
Applicabilità:
- Implementare le parti invarianti di un algoritmo una volta sola (definendo le parti astratte come *primitive*)
- Evitare la duplicazione del codice
    - Principio "refactoring to generalize"
- Controllare le possibili estensioni di una classe
    - Fornire sia operazioni astratte sia operazioni hook (wrapper)
- Per permettere ai client di estendere solo particolari passi di un algoritmo, ma non l'intero algoritmo o la sua struttura



- La *Abstract Class* dichiara i metodi che agiscono come passi di un algoritmo, nonché il metodo Template vero e proprio che chiama questi metodi in un ordine specifico. I passi possono essere dichiarati astratti o avere un'implementazione predefinita

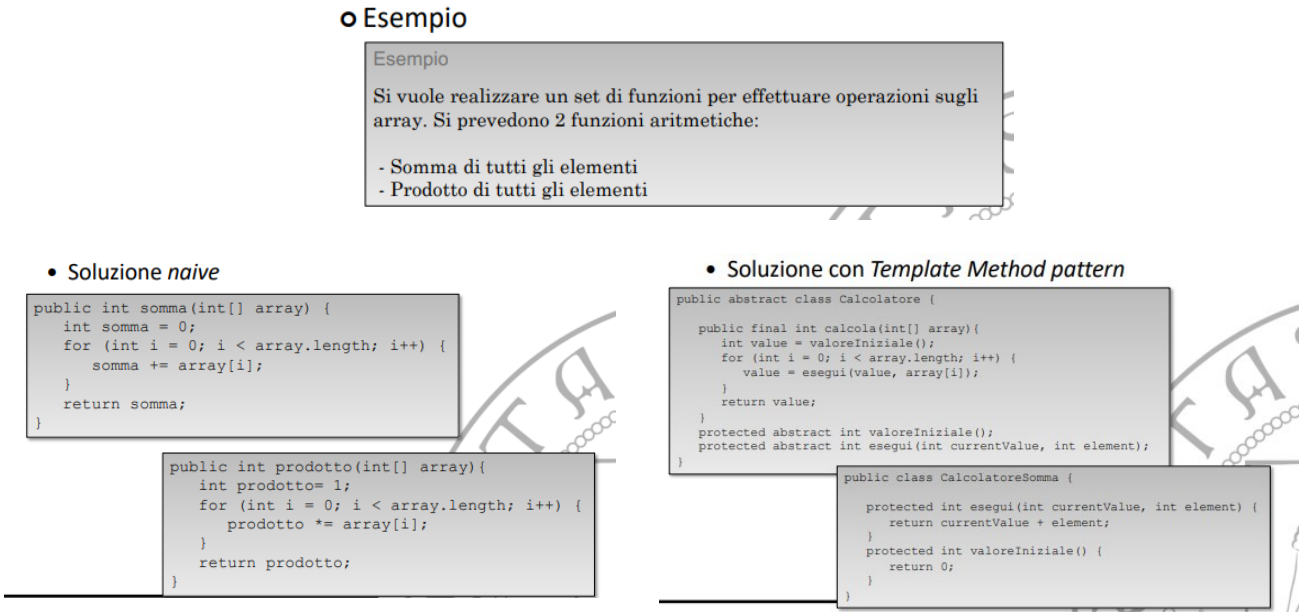
- Le *Concrete Class* possono sovrascrivere tutti i passi, ma non il metodo Template stesso

Conseguenze:

- Tecnica per il riuso del codice
  - Fattorizzazione delle responsabilità (uso quello che mi serve adattandolo ad uno specifico compito)
- "The Hollywood principle"
  - Afferma che un oggetto che dipende da un altro non deve chiamarlo, ma deve aspettare che sia l'altro a chiamarlo; quindi, il template viene chiamato se serve
- Tipi di operazioni possibili
  - Operazioni concrete della classe astratta
  - Operazioni primitive (astratte)
  - Operazioni hook
    - Forniscono operazioni che di default non fanno nulla, ma rappresentano punti di estensione
- Documentare bene quali sono operazioni primitive e quali hook

Spesso, infatti, si consiglia di utilizzare la composizione al posto dell'ereditarietà per Template Pattern.

Nell'esempio sottostante, le operazioni primitive di esecuzione calcolo e valore iniziale sono definite protette, mentre il metodo Template viene definito *final*. Di fianco, l'implementazione naif che scorre tutto l'array ed esegue somma e prodotto con un banale ciclo, ripetendo la stessa operazione; come accade in C++, l'operazione richiede qualcosa che permetta di "prevenire l'operazione ripetuta", appunto il template.



Implementazione:

- Le operazioni primitive dovrebbero essere membri protetti
- Il codice duplicato si può mettere in una superclasse
- Il template method non dovrebbe essere ridefinito
  - Java: dichiarazione "final"
- Minimizzare il numero di operazioni primitive
  - Resta poco nel Template method
- Definire una *naming convention* per i nomi delle operazioni di cui effettuare *override*
- È possibile consentire ai client di sovrascrivere solo alcune parti di un algoritmo di grandi dimensioni, rendendoli meno influenzati dalle modifiche apportate ad altre parti dell'algoritmo

Esempio del pattern Template in Python:

```
from abc import ABC, abstractmethod

class Game(ABC):
    def play(self):
        self.initialize()
        self.start_play()
        self.end_play()

    @abstractmethod
    def initialize(self):
        pass

    @abstractmethod
    def start_play(self):
        pass

    @abstractmethod
    def end_play(self):
        pass

class Cricket(Game):
    def initialize(self):
        print("Cricket Game Initialized! Start playing.")

    def start_play(self):
        print("Cricket Game Started. Enjoy the game!")

    def end_play(self):
        print("Cricket Game Finished!")

class Football(Game):
    def initialize(self):
        print("Football Game Initialized! Start playing.")

    def start_play(self):
        print("Football Game Started. Enjoy the game!")

    def end_play(self):
        print("Football Game Finished!")

game = Cricket()
game.play()  # prints "Cricket Game Initialized! Start playing.", "Cricket Game Started. Enjoy the game!",
"Cricket Game
```

In questo esempio, la classe *Game* definisce il metodo template *play*, che consiste in una serie di passi comuni a tutti i giochi. I metodi *initialize*, *start_play* ed *end_play* sono contrassegnati come astratti e devono essere implementati da sottoclassi concrete.

Le classi *Cricket* e *Football* sono sottoclassi concrete della classe *Game* che forniscono implementazioni specifiche dei metodi astratti.

Il codice *client* crea un oggetto *Cricket* e un oggetto Football e chiama il metodo play su ciascuno di essi. Ciò comporta l'esecuzione dei passi comuni dell'algoritmo e dei passi specifici implementati nelle sottoclassi concrete.

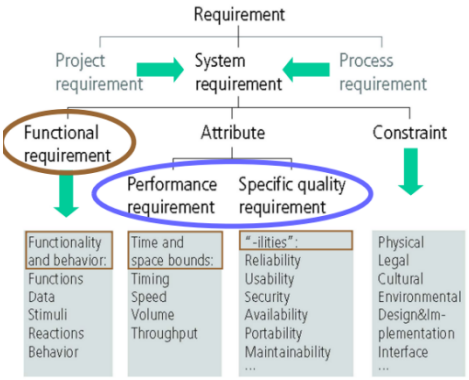
Verifica e Validazione: Analisi Statica (Vardanega)

Un SW di qualità deve possedere:

- Tutte le capacità *funzionali* attese (qualità esterna), che specificano *cosa il sistema debba fare*
- Tutte le caratteristiche *non-funzionali* necessarie (qualità interna/performance) affinché il sistema lavori sempre *come previsto*

Dimostrarlo richiede accertare il possesso di svariate proprietà:

- *Di costruzione*: architettura, codifica, integrazione
- *D'uso*: esperienza utente, precisione, affidabilità
- *Di funzionamento*: prestazioni, robustezza, sicurezza



Costruiamo la verifica secondo una matrice impostata gerarchicamente e accettando o meno i requisiti, approvandoli a seconda del caso.
Come dice Dijkstra, deve essere chiaro come un prodotto soddisfi i requisiti, in particolare:

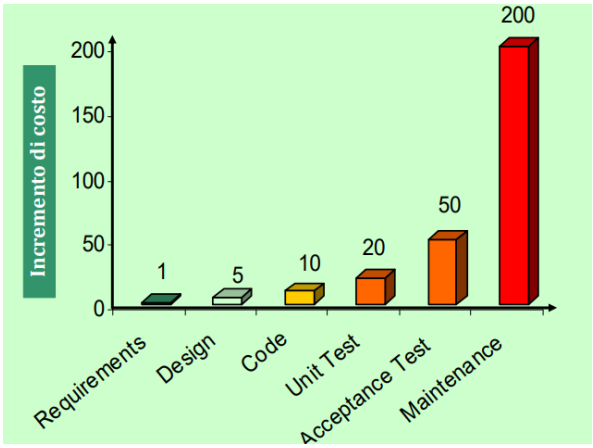
1) Indicare le proprietà di una cosa, in virtù delle quali essa soddisferebbe i nostri bisogni
    - Analisi e specifica dei requisiti
2) Fare in modo che un oggetto abbia le proprietà volute
    - Progettazione e codifica della soluzione

La verifica accerta che la seconda proprietà soddisfi la prima.

Ogni progetto ha delle norme di codifica, tali da rendere la verifica più agevole. Infatti, la codifica deve aiutare la verifica, non ostacolarla. Pochi linguaggi la facilitano attivamente e, per questo, serve imporre disciplina di programmazione. La verifica deve continuare a garantire l'usabilità e la mancanza di errori nel codice, essendo sempre a conoscenza di ciò che fa e delle sue parti.

Per scrivere *programmi verificabili* è fondamentale dotarsi di norme di codifica coerenti con le esigenze di verifica, promuovendo buone prassi di codifica e vincoli sui costrutti di programmazione, verificandone attivamente il rispetto.

La verifica retrospettiva è inutile; normalmente, il costo di rilevazione e correzione di errori cresce con l'avanzare dello sviluppo.
Come si vede qui, il costo di manutenzione è quello più grosso.



- L'approccio reattivo alla verifica è ingenuo ed ottimistico (*hoping for correctness by correction*), che agisce solo quando c'è un problema.
- È più saggio sostenere lo sviluppo con la verifica sistematica: approccio costruttivo (*pursuing correctness by construction*), facendo passi che sono giusti per loro natura.

Occorre regolamentare l'uso del linguaggio di programmazione tramite principi da riflettere nelle Norme di Progetto in modo documentale:

1. Per assicurare *comportamento predicibile*
    a. Ciò intende dire prima *cosa* succederà, *senza* effettivamente *eseguire*. Per essere senza ambiguità, occorre prevedere:
        i. Effetti laterali – side effects (p.es. di sottoprogrammi), intese come invocazioni della stessa azione che producano effetti diversi. Questi sono dovuti alla condivisione di memoria; utile se serve, altrimenti occorre evitarla
        ii. Ordine di elaborazione e inizializzazione, sapendo che l'elaborazione prevede che l'eseguibile, una volta compilato, interagisca con un ambiente di esecuzione (runtime) creando le condizioni utili per l'utilizzo del codice.
            1. Esempio: i *thread* sono impredicibili a livello di attivazione
        iii. Modalità di passaggio dei parametri, per cui la scelta di una modalità di passaggio (per valore, per riferimento) può influenzare l'esito dell'esecuzione

Eseguiamo analisi statica di questo codice, capendo se inverte oppure no:

```
class Swapper{
  public static void swap(int Left, int Right)
  {
    int tmp = Left;
    Left = Right;
    Right = Left;
  }

  public static void main(String args[])
  {
    int Source = 1;
    int Destination = 3;
    swap(Source, Destination);
  }
}
```

In Java, i nomi sono riferimenti, ma le chiamate sono per valore!

Questo codice non funziona come previsto. Il metodo *swap* prende come argomenti due numeri interi, ma questi argomenti non vengono modificati all'interno del metodo.

Al contrario, il metodo crea nuove variabili locali con gli stessi nomi degli argomenti e assegna loro nuovi valori, ma queste modifiche non influiscono sugli argomenti originali.

Per far sì che il metodo *swap* funzioni come previsto, si potrebbero passare gli argomenti come riferimenti ai valori, come si vede a lato.

```
class Swapper{
public static void swap(int[] arr)
{
int tmp = arr[0];
arr[0] = arr[1];
arr[1] = tmp;
}
```

2. Per usare buoni principi di programmazione

A prescindere dal linguaggio di programmazione, tutto questo deve esistere nel codice:

- Riflettere l'architettura (*design*) nel codice
    - Usare programmazione strutturata per esprimere componenti, moduli, unità come da progettazione, e facilitare l'integrazione
- Separare le interfacce dall'implementazione
    - Fissare bene le interfacce a partire dall'architettura logica
    - Esporre le interfacce (ciò che il chiamante deve conoscere)
    - Nascondere l'implementazione
- Massimizzare l'incapsulazione (*information hiding*)
    - Usare membri privati e metodi pubblici per l'accesso ai dati
- Usare tipi specializzati per specificare dati
    - Composizione e specializzazione aumentano il potere espressivo dei tipi del programma
    - Soprattutto, grazie ai tipi definiti dall'utente, riusciamo a specializzare un dato al contesto di lavoro (e personalizzarlo in base alle esigenze)

3. Per ragioni pragmatiche (cioè, concrete/pratiche)

L'efficacia dei metodi di verifica è funzione della qualità di strutturazione del codice (e.g. una procedura con un solo punto di uscita facilita l'analisi del suo effetto sullo stato del sistema; se ha più punti di uscita, amplifico i possibili effetti del codice, perché ho più possibilità).
La verifica di un programma relaziona frazioni di codice con frazioni di specifica (cioè, più modulare è la struttura più facile è determinare la correlazione con i requisiti):

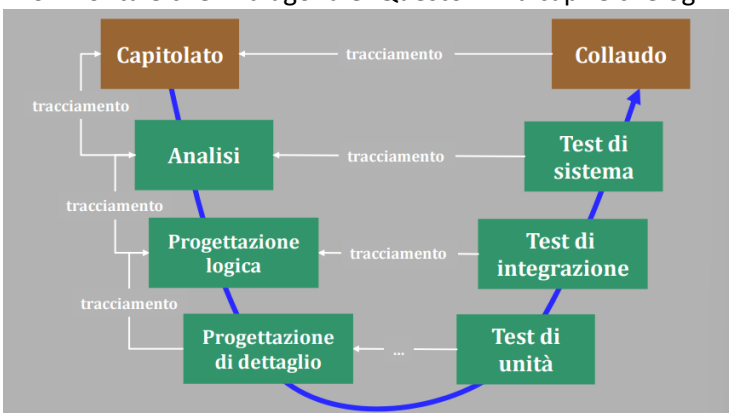
- La verificabilità è funzione inversa dell'ampiezza del contesto oggetto di verifica
    - Più cresce il secondo, più diminuisce la prima: confinare *scope* e visibilità
- Una buona architettura facilita la verifica
    - P.es. tramite incapsulazione dello stato e controllo di accesso

Il tracciamento aiuta in quanto dimostra che tutto ciò che è stato fatto sia lì per una ragione specifica. Esso dimostra completezza ed economicità del prodotto garantendo che nessun requisito sia stato dimenticato e non vi sia nessuna funzionalità superflua. Va applicato:

- Su ogni passaggio della specifica (ramo discendente)
- Su ogni passaggio dell'implementazione (ramo ascendente)
- Va automatizzato il più possibile per limitarne il costo all'aumentare della sua intensità

Conclusione Analisi Statica (Vardanega)

Come si vede dal modello a V, il tracciamento permette di definire le relazioni causa-effetto tra le parti, sia in orizzontale che in diagonale. Questo mi fa capire che ogni volta che calendarizzo, penso alla verifica.



Ogni attività deve essere piccola per intensità e durata, individuando periodi brevi di azione. La codifica è un'attività singola, tale che ogni unità sia facilmente verificabile.

Il repo cresce incrementalmente a piccoli pezzi e più volte. Integriamo cose buone e con esito controllato.

Il codice presenta una struttura precisa: un'intestazione per ogni unità, in cui si presenta una breve descrizione e la soddisfazione di certi requisiti (creato automaticamente e verificato correttamente dal repo), marcandolo chiaramente.

Quanti requisiti soddisfare?

- Questa risposta viene dalla verifica, perché intendo risalire il V Model avendo già in mano tutti gli oggetti di verifica, che da requisiti sono diventati implementazione e quindi oggetto di test
- Tradotto: dipende dal contesto e dal tipo specifico
- Occorre parallelizzare quelle del ramo ascendente, massimizzando l'esecuzione asincrona, evitando di andare nel ramo discendente

Tracciare i requisiti su progettazione di dettaglio e codifica aiuta a valutare il costo di verifica:

- Assegnare N requisiti elementari a 1 singolo modulo SW richiede N procedure di prova per quel modulo
    - (1 prova per 1 requisito aiuta a rendere le prove decidibili)
- Al crescere di N crescono la criticità e il costo di quel modulo

Chi scrive codice deve saperne l'obiettivo e i suoi costrutti devono essere *espressivi*, pena complessità di esecuzione e maggiore costo nella dimostrazione di correttezza.
Maggiore il potere espressivo di un costrutto, maggiore la sua complessità di esecuzione ed il costo di dimostrarlo corretto (dipende sempre dal tipo di costrutto e dal suo contesto).
Le unità devono rispondere in modo specifico ai requisiti, in modo organizzato e modulare.

Esistono vari tipi di analisi statica del codice (di queste siamo noi a decidere cosa usare nel progetto):

1) Analisi di flusso di controllo
    a. Essa accerta la logica (cioè, l'esecuzione da parte del PC avverrà nella sequenza specificata) e la visibilità del codice (buona struttura), verificandone la propagazione. Si localizza inoltre codice non raggiungibile (mai eseguito in nessun caso ma sintatticamente corretto; questo può non essere mai testato e non sappiamo cosa quel pezzo faccia), identificando rischi di non terminazione
    b. Si deve dimostrare che il pezzo di codice funziona in ogni condizione. La dimostrazione di correttezza della ricorsione è pericolosa per il rischio di non terminazione; per fare questo, si usa l'albero delle chiamate/*call-tree analysis*, mostrando l'ordine di chiamata e se è presente ricorsione diretta/indiretta). Questo viene tracciato dal verificatore

2) Analisi di flusso dei dati
    a. Essa accerta che nessun cammino d'esecuzione del programma acceda a variabili non valorizzate (analisi sulle variabili e le modalità di accesso). Rileva inoltre variabili globali e possibili anomalie di scrittura/lettura (scritture sovrascritte, letture fatte nel punto sbagliato, scritture con letture mai fatte)
    b. Si vuole evitare di violare il principio di incapsulazione, ad esempio con gli alias (zone condivise in memoria tenute per i puntatori)

3) Analisi di flusso d'informazione
    a. Essa calcola le dipendenze tra ingressi e uscite determinate dall'esecuzione di singole unità di codice, identificando side-effects e dipendenze inammissibili
    b. L'informazione è il prodotto di elaborazione dati che deve essere conforme a certe attese, trovando i rapporti tra dati di I/O. Vorremmo evitare dipendenze inattese tra input e output, facendo in modo che il flusso informativo ci dica per certo cosa fare

4) Analisi di limite
    a. Essa verifica che i valori del programma restino sempre entro i limiti del loro tipo e della precisione desiderata, per esempio:
        1. Overflow, valori maggiori al massimo rappresentabile
        2. Underflow, valori minori al minimo rappresentabile
        3. Range checking, rispetto dei limiti nell'accesso alle strutture dati
    b. Alcuni linguaggi permettono di assegnare limiti statici a tipi discreti per facilitare verifica sulle corrispondenti variabili; più difficile farlo con tipi enumerati e reali.
        1. Similmente, con i linguaggi di programmazione possiamo superare anche limiti matematici (es. intendere float come reali, etc.).

5) Analisi d'uso di stack
    a. Per determinare la massima domanda di stack richiesta a tempo d'esecuzione in relazione con la dimensione della memoria assegnata all'esecuzione del programma, facendo in modo non vi sia rischio di collisione tra *stack* ed *heap*.

- Lo *stack/pila* è la memoria usata per ospitare dati locali e indirizzi di ritorno generati dal compilatore alla chiamata di sottoprogrammi. Ogni flusso di controllo ha il suo stack, la cui dimensione cresce con l'annidamento delle chiamate/uso di funzioni e i dati hanno chiare regole di visibilità e ciclo di vita. Esso lavora sui parametri visibili alla chiamata, che non sono visibili all'OS

- Lo *heap/mucchio* è la memoria usata per tutto il resto, con dimensione fissata prima dell'inizio dell'esecuzione del programma e contenuto determinato dagli oggetti globali creati nell'esecuzione del programma. Quando oltrepassiamo il confine delle chiamate, si ha un segmentation fault (in questo, la ricorsione è pericolosa perché non si ferma mai)

6) Analisi temporale
    a. Esso permette di studiare le dipendenze temporali (latenza) tra le uscite del programma e i suoi ingressi e verifica che il valore giusto sia prodotto al momento giusto (cercando di saperlo *by design*)
    b. Limiti espressivi dei linguaggi e delle tecniche di programmazione complicano questa analisi, ad esempio:
        1. Iterazioni prive di limite statico (while)
        2. Creazione dinamica di variabili (new)

7) Analisi d'interferenza
    a. Essa permette di mostrare l'assenza di effetti di interferenza tra parti isolate ("partizioni") del sistema (non per forza limitate a componenti SW). Tipici problemi di interferenza derivano da memoria virtuale condivisa (parti separate di programma lasciano traccia di dati abbandonati ma non distrutti, *memory leak,* mitigato dall'azzeramento dei page frame rilasciati dal programma) o I/O programmabile/registri di periferiche (variabili *volatili*)

SOLID Principles of Object-Oriented Design (Cardin)

I principi SOLID sono stati introdotti per la prima volta da Robert Martin nel 2000. Servono tutti allo stesso scopo: "Creare codice comprensibile, leggibile e testabile su cui molti sviluppatori possano lavorare in modo collaborativo, aiutando a gestire le dipendenze, in modo flessibile e robusto".

Analizziamo ogni principio uno per uno. Seguendo l'acronimo SOLID, essi sono:
- *Single Responsibility Principle*
    - Una classe dovrebbe avere una, e una sola, ragione per cambiare
- *Open-Close Principle*
    - Si dovrebbe essere in grado di estendere il comportamento di una classe, senza modificarlo, ma aggiungendo codice
- *Liskov Substitution Principle*
    - Le classi derivate devono essere sostituibili con le loro classi base
- *Interface Segregation Principle*
    - Creare interfacce a grana fine che siano specifiche per il client; esso non deve essere forzato ad implementare un'interfaccia che non deve usare
- *Dependency Inversion Principle*
    - Si dovrebbe dipendere dalle astrazioni, non dalle concretizzazioni

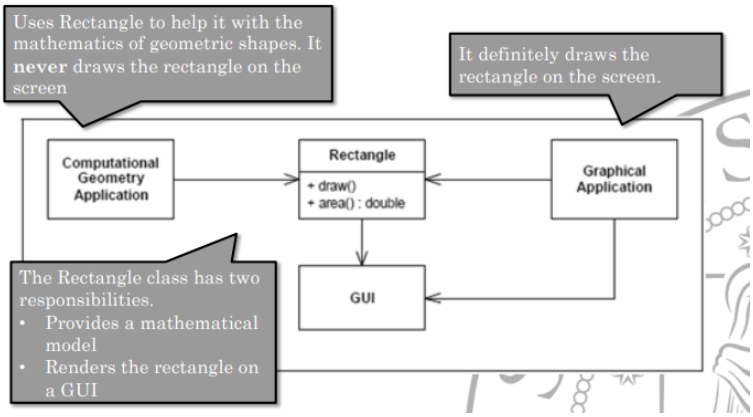
Cosa da ricordare: se avessi solo la classe senza sapere quali sono le sue implementazioni, non potrei sapere a priori se soddisfa i principi SOLID.

Il Single Responsibility Principle (SRP), noto anche come *coesione*, afferma che un modulo dovrebbe avere un solo motivo per cambiare (vi deve essere correlazione funzionale degli elementi di un modulo), tale da "avere sempre una motivazione di cambio responsabilità".
Le responsabilità sono accoppiate (in quanto le modifiche a una responsabilità possono compromettere o inibire la capacità della classe di soddisfare le altre), portando ad un design più fragile che può rompersi in modi inaspettati (ricompilazione, test, deploy, etc.).

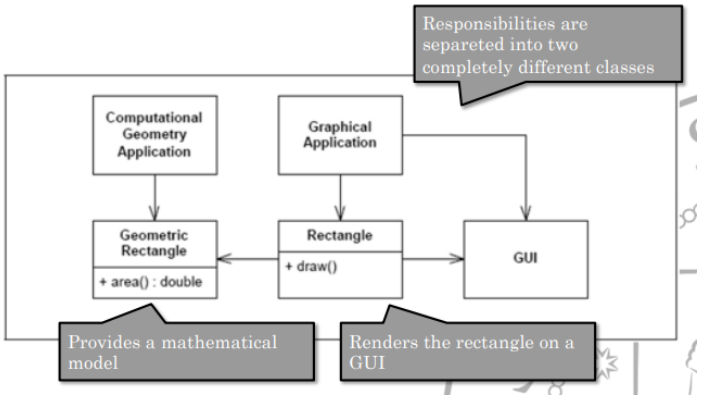
Bisogna analizzare tutto il contesto e non la singola classe per capire se si aderisce al principio SRP.

Vediamo un esempio di una classe che non rispetta il principio precedente, in quanto ha più responsabilità.

- Il metodo *draw()* viene usato solo dalla GUI
- Il metodo *area()* è usato dall'applicazione



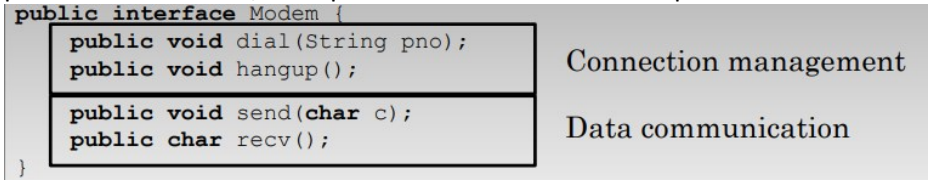
L'accoppiamento fa in modo che, se cambio i metodi, potrei potenzialmente modificare la GUI o l'applicazione (in quanto la parte di calcolo sarebbe coinvolta nei cambiamenti grafici).
Nell'esempio seguente, invece, realizziamo correttamente la separazione diminuendo l'accoppiamento; la parte grafica viene del tutto separata dalla parte di calcoli.



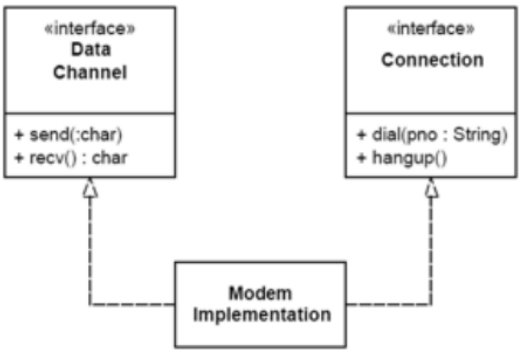
Che cos'è davvero una responsabilità?

- Un asse di cambiamento è tale solo se i cambiamenti si verificano effettivamente; anche il contesto dell'applicazione è importante (si potrebbe aggiungere complessità inutile)

L'esempio è di una interfaccia Modem, in cui i metodi dati e comunicazione sono presenti per come viene pensata inizialmente; le responsabilità devono essere separate a seconda di come cambia l'applicazione.



Un'eventuale separazione delle responsabilità rispetto al caso precedente è come segue (evitando rigidità); sono ancora accoppiati in *ModemImplementation*, ma i client non devono preoccuparsi delle implementazioni delle interfacce.



Ripensando all'Hexagonal Architecture, rafforza SRP separando chiaramente business logic (che ricordiamo può essere testata indipendentemente con l'uso di porte e adattatori) rispetto ad esempio alla layered architecture.

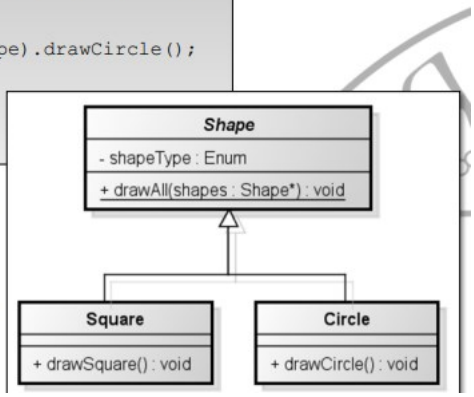
Per l'Open-Close Principle (OCP), le entità software devono essere aperte all'estensione, ma chiuse alle modifiche (si estende il comportamento aggiungendo nuovo codice, non modificando il vecchio).
Ci sono varie euristiche da seguire (variabili di membro private, variabili globali vanno evitate, usare L'RTTI (Run Time Type Identification) e l'OCP le sottolinea tutte.
L'astrazione è la chiave; i tipi astratti sono la parte fissa, le classi derivate sono i punti di estensione.

L'esempio successivo dimostra che:

- Qualsiasi volta voglia aggiungere una nuova funzionalità, voglio aggiungere codice senza modificare il codice esistente (nell'esempio successivo, aggiungendo nuove forme, bisogna modificare il codice)
- Occorre risolvere il problema che si ha *in quel momento*, senza sovraingegnerizzare il codice
- Si ha ereditarietà ma non polimorfismo, in quanto il metodo *drawAll* non considera le implementazioni di *Square* e *Circle*; inoltre, *shapeType* ha poco senso, in quanto occorre astrarre proprio per evitare dipendenza dalle sottoclassi/sottotipi, come avviene nello *switch* sotto

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        switch (shape.shapeType) {
            case Square:
                ((Square) shape).drawSquare();
                break;
            case Circle:
                ((Circle) shape).drawCircle();
                break;
        }
    }
}
```

Does not conform to the open-closed principle because it cannot be closed against new kinds of shapes. If I wanted to extend this function, I would have to modify the function

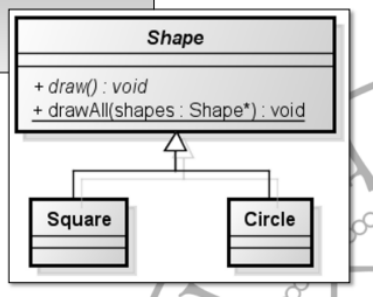


Non è conforme a OCP perché non può essere chiuso a nuovi tipi di forme. Se volessi estenderla, dovrei modificare la funzione.

È aperta alla modifica ma non al cambiamento. Se non modifico, non introduco errori.

L'uso del polimorfismo mi garantisce di essere open-close; la soluzione presente sotto è conforme al principio di apertura e chiusura. Per estendere il comportamento di *drawAll* per disegnare un nuovo tipo di forma, tutto ciò che si deve fare è aggiungere un nuovo derivato della classe *Shape*.
I programmi conformi all'OCP non subiscono "cambiamenti a cascata"; le modifiche si ottengono aggiungendo nuovo codice.

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```



Solution that conforms to open-close principle. To extend the behavior of the `drawAll` to draw a new kind of shape, all we need do is add a new derivative of the `Shape` class.

Le modifiche possono sempre introdurre probabilità di apertura, quindi attenzione.

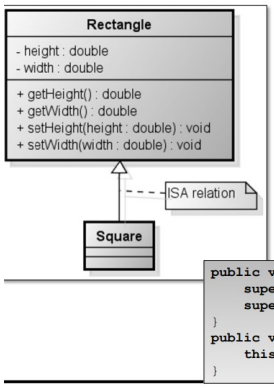
Il succo è: estendo le funzionalità di quello che già esiste in modo corretto, quindi *astraggo* dal contesto e aggiungo codice che mi permette di realizzare questo.
Nessun programma sarà mai chiuso al 100% e la *closure*/chiusura va raggiunta in modo strategico, appunto con l'astrazione (uso di interfacce e polimorfismo) secondo una dipendenza *data-driven* (quindi, usando i dati e astraendo a seconda dell'ordine per i quali i tipi si presentano).

Alcune convenzioni:

- Rendere private tutte le variabili membro
    - Quando le variabili membro di una classe cambiano, ogni funzione che dipende da esse deve che dipende da esse deve essere modificata
    - Incapsulamento
- Evitare variabili globali
    - Nessun modulo che dipenda da una variabile globale può essere chiuso contro qualsiasi altro modulo che potrebbe scrivere su quella variabile
        - Ci sono pochissimi casi che possono disobbedire (es. *cin, cout*)
- La RTTI è pericolosa
    - L'esempio di *Shape* mostra il modo sbagliato di usare l'RTTI (Runtime Type Checking), in quanto implicitamente useremmo il polimorfismo cercando di semplificare il programma, ma lo blocchiamo vincolandolo ad una specifica implementazione (caso delle forme)
    - Altri problemi sono le performance (determinazione dinamica dei tipi costa) e la sicurezza (un attaccante potrebbe usare RTTI per determinare il tipo di un oggetto e chiamare metodi associati a questo in modi non intesi in origine)
    - Attenzione ai linguaggi ad oggetti non tipizzati (es. JavaScript, Groovy); possono violare il principio OCP (perché non siamo interessati allo stato interno degli oggetti, ma al comportamento/behaviour delle classi)

Il Liskov Substitution Principle (LSP) afferma che "Le funzioni che utilizzano puntatori o riferimenti a classi base devono essere in grado di utilizzare oggetti di classi derivate senza saperlo e non rompendo l'applicazione" e ha senso fare l'override di metodi concreti, dato che aggiungo delle condizioni per vincolare l'implementazione a contratto della classe senza modificare il comportamento base della classe. Violare questo principio significa violare l'OCP (funzione che utilizza un puntatore o un riferimento a una classe base, ma deve conoscere tutti i derivati di quella classe base).

Questo sta alla base di gerarchie di ereditarietà fatte bene: sono ben organizzate, facili da mantenere, flessibili ed efficienti, evitando di creare complessità non organizzando la gerarchia chiaramente e non usando l'ereditarietà quando si dovrebbe.
L'idea nell'esempio successivo è vedere il quadrato come rettangolo con tutti i lati uguali; in questo modo, eseguo un override, essendo sicuro che altezza e larghezza siano della stessa dimensione.



A `Square` does not need both `height` and `width` member variables. Yet it will inherit them anyway. Clearly this is wasteful.

`Square` will inherit the `setWidth` and `setHeight` functions. These functions are utterly inappropriate for a `Square`.
But, we could override them...

```
public void setWidth(double width) {
    super.setWidth(width);
    super.setHeight(width);
}
public void setHeight(double height) {
    this.setWidth(height);
}
```

Nell'esempio successivo il test fallisce; il quadrato ha la precondizione di avere altezza e larghezza sempre uguali. Mentre imposto l'altezza, non ho vincoli sulla larghezza. Qui viene violato il principio LSP, perché il rettangolo non avrà altezza per i lati uguale e non avrò altezza diversa per il quadrato, andando in contrasto con quanto detto sopra.

If we pass a reference to a `Square` object into this function, and the height will be changed too.

This is a clear violation of LSP. The `f` function does not work for derivatives of its arguments.

```
public void f(Rectangle r) {
    r.setWidth(42);
}
@Test
public void testF() {
    Rectangle r = new Square();
    r.setHeight(15);
    f(r);
    // This test will not pass!!!
    assertEquals(15, r.getHeight());
}
```

Cosa è andato storto?

- Ciò che conta è il comportamento pubblico estrinseco (cioè, il comportamento mostrato all'esterno)
    - È il comportamento da cui dipendono i client
    - La relazione tra Quadrato e Rettangolo non è una relazione IS-A in OOD

Il principio LSP è anche noto come programmazione per contratto/design by contract è un paradigma di progettazione del software che utilizza contratti formali per specificare le aspettative e gli obblighi dei vari componenti di un sistema software. Qui, lo sviluppatore scrive *contratti* per ogni classe o funzione che specificano gli input (*precondizioni*), gli output e altre aspettative per quella classe o funzione (*postcondizioni*).

Questi contratti vengono poi applicati in fase di esecuzione e qualsiasi violazione dei contratti viene rilevata e segnalata come errore. Il suo utilizzo migliora la correttezza e l'affidabilità del programma, rendendolo più facile da mantenere (anche da leggere e da capire); tuttavia, può richiedere un significativo overhead dovuto dal controllo dei contratti a runtime. Usi reali si ritrovano in Spring di Java, .NET, Ada, etc.

"Quando si ridefinisce una routine [in una derivata], si può solo sostituire la sua precondizione con una più debole e la sua postcondizione con una più forte".
Esempio delle asserzioni in Java e nei linguaggi JVM (Kotlin, Scala, Groovy, etc.):

```
// Rectangle.setWidth(double w) postconditions
assert((width == w) && (height == old.height));
```

Riguardo al "design by contract":

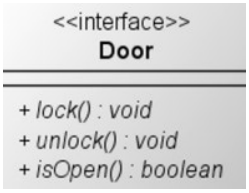
- In una classe derivata le precondizioni non devono essere più forti rispetto alla classe base
    - Utilizzando l'interfaccia della classe base, il cliente conosce solo le precondizioni della classe base precondizioni della classe base
- In una classe derivata le postcondizioni devono essere più forti rispetto alla classe base
    - La classe derivata deve essere conforme a tutte le condizioni della classe base
    - I comportamenti e gli output non devono violare vincoli stabiliti per la classe base

Altro principio utile è l'Interface Segregation Principle, per cui i client non devono essere costretti a dipendere da interfacce che non utilizzano:

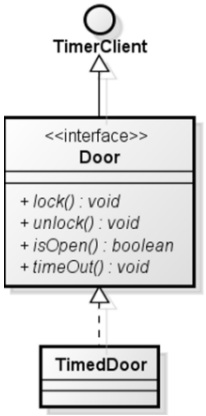
- Dichiarare in un'interfaccia metodi di cui il cliente non ha bisogno inquina l'interfaccia e porta a un'interfaccia "ingombrante/bulky" o "grassa/fat".
- Le interfacce "fat" non sono coesive (ricordiamo che questo può succedere con *Facade*)
    - I metodi possono essere suddivisi in gruppi di funzioni
    - I client devono visualizzare solo la parte che gli interessa
- I client non devono essere costretti a dipendere da interfacce che non utilizzano

Il succo del discorso è: creare delle interfacce con pochi metodi e molto specifiche. Mettere tanti metodi in una interfaccia può essere controproducente perché si rischia di avere classi che non dispongono di tutte quelle funzionalità e addirittura in alcuni casi portare alle stesse rogne che si possono avere con l'ereditarietà da classi concrete (obbligo di utilizzo di metodi che nel contesto non servono).

In questo sistema ci sono oggetti *Door* che possono essere bloccati e sbloccati, e che sanno se sono aperti o chiusi. I client utilizzano questa interfaccia per gestire le porte. Consideriamo ora una di queste implementazioni.
TimedDoor ha bisogno di suonare un allarme quando la porta è stata lasciata aperta per troppo tempo. Per farlo, l'oggetto *TimedDoor* comunica con un altro oggetto, chiamato *Timer*. Il metodo *TimeClient* rappresenta la funzione chiamata quando il timeout scade.



Prima soluzione:

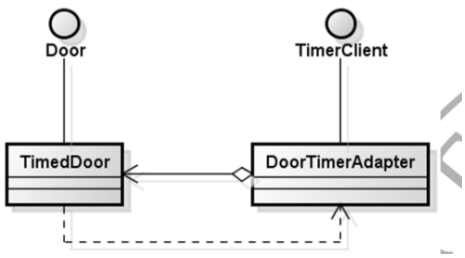


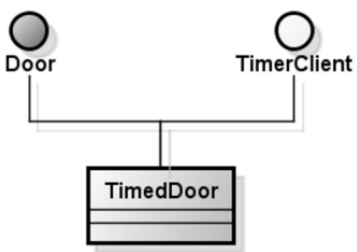
La classe *Door* ora dipende da *TimerClient*. Non tutte le varietà di porte necessitano di di temporizzazione. Inoltre, le applicazioni che utilizzano derivati dovranno importare la definizione della classe *TimerClient*, anche se non viene utilizzata.

L'interfaccia di *Door* è stata inquinata con un'interfaccia che non è necessaria. Ogni volta che viene aggiunta una nuova interfaccia alla classe di base, quella interfaccia deve essere implementata nelle classi derivate. Le implementazioni di default violano LSP.

- I client di *Door* e di *TimerClient* sono diversi
- Anche le interfacce dovrebbero rimanere separate
- Anche l'interfaccia *Door* deve essere modificata
    - Anche i client che non hanno bisogno di porte temporizzate ne saranno influenzati
- Il risultato è un accoppiamento involontario tra tutti i client

Un possibile disaccoppiamento è dato della separazione per delegazione, usando il pattern Adapter, permettendo di disaccoppiare *Door* e *TimerClient* (immagine di sx), con un'unica interfaccia.
Altra possibile separazione è sfruttando l'ereditarietà multipla (immagine di dx), in cui il client può utilizzare lo stesso oggetto tramite interfacce diverse e separate (possibile se l'ereditarietà multipla è supportata e si usano meno tipi rispetto alla soluzione precedente).





Scritto da Gabriel

Una cattiva progettazione spesso deriva dal degrado dovuto a nuovi requisiti e alla manutenzione:

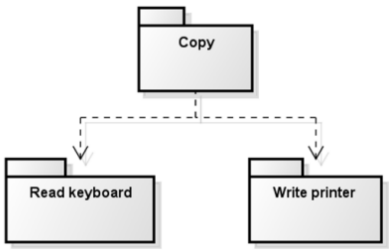
- Rigidità - difficile da modificare perché ogni cambiamento riguarda su molte parti del sistema
- Fragilità - quando si effettua un cambiamento, parti inaspettate del sistema si rompono del sistema si rompono
- Immobilità - è difficile da riutilizzare in un'altra applicazione perché non può essere facilmente separato

Secondo il Dependency Inversion Principle, i moduli di alto livello non devono dipendere dai moduli di basso livello; entrambi devono dipendere dalle astrazioni, per poter limitare la dipendenza. Ulteriormente "le astrazioni non devono dipendere dai dettagli; i dettagli dovrebbero dipendere dalle astrazioni".

Molto semplicemente indica il fatto che conviene usare spesso interfacce e classi astratte per definire la forma (e poi implementare o ereditare da queste) invece che usare subito una classe concreta che definisce il comportamento. Seguendo questo principio, è possibile progettare il sistema software in modo da favorire la flessibilità e la manutenibilità.

Il principio dell'inversione delle dipendenze viene spesso implementato utilizzando la tecnica dell'iniezione delle dipendenze (*dependency injection*), in cui le dipendenze di un modulo vengono iniettate nel modulo in fase di esecuzione alla bisogna, anziché essere codificate nel modulo stesso. In questo modo è facile modificare le dipendenze di un modulo senza modificare il modulo stesso.

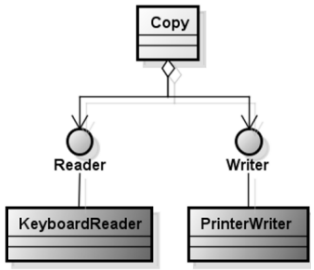
Consideriamo un semplice programma che ha il compito di copiare i caratteri digitati su una tastiera a una stampante. I comandi "Read keyboard" e "Write printer" sono abbastanza riutilizzabili. Tuttavia il modulo "Copy" non è non è riutilizzabile in nessun contesto che non coinvolga tastiera e stampante.

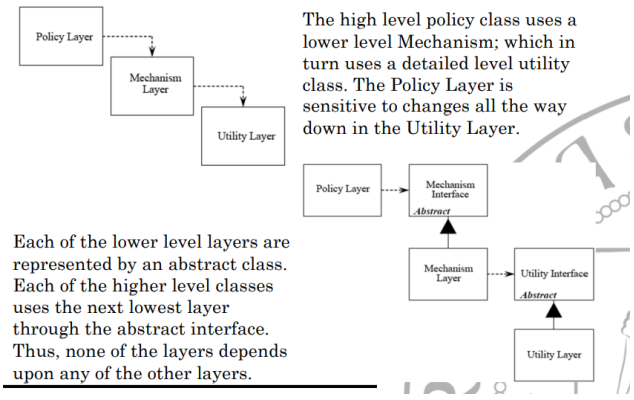



```
public void copy(OutputDevice dev) {
    int c;
    while ((c = readKeyboard()) != EOF)
        if (dev == PRINTER)
            writePrinter(c);
        else
            writeDisk(c);
}
```

Violates OCP

Le dipendenze sono state invertite: la classe "Copia" dipende dalle astrazioni (due interfacce apposite di lettura e scrittura), con lettori da tastiera/scrittori da stampante che le usano senza dipendenza. Ora possiamo riutilizzare la classe "Copy "indipendentemente dalla classe "Read keyboard" e dal "Write printer".





La classe di politiche di alto livello utilizza un *Mechanism* di livello inferiore, che a sua volta utilizza una classe di utilità dettagliato. Il Policy Layer è sensibile ai cambiamenti che avvengono nello Utility Layer. Ciascuno dei livelli inferiori è rappresentato da una classe astratta. Ogni classe di livello superiore utilizza il livello immediatamente inferiore attraverso l'interfaccia astratta. In questo modo, nessuno dei livelli dipende nessuno dei layer dipende dagli altri.

Le decisioni importanti sono contenute in moduli di alto livello, che vogliamo quindi poter riutilizzare (*template method pattern*). In una layered application, ogni livello deve esporre un adeguato livello di astrazione (interfaccia). Un'implementazione ingenua può forzare una dipendenza errata fra moduli. Quindi: *le parti core dell'applicazione (business) non devono dipendere dalle altre parti*. Per fare questo, usiamo le interfacce (perché on dipendendo dall'implementazione).

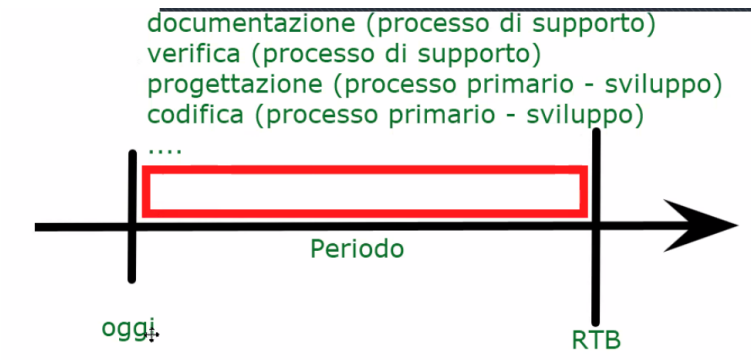
Diario di bordo: Verifica, ruoli, processi e periodi (Vardanega)

- Verifica e Validazione

Non sono sinonimi; entrambi hanno un loro processo specifico e ciò è posto a glossario. Viene fatta la verifica per far avanzare la baseline, tale che si dia una conferma certa e una successiva validazione.

- Processi di supporto

Sono chiamati da processi primari (fornitura, sviluppo, etc.), secondo obblighi contrattuali. Questo è evidente dal V-Model; né i documenti né il PoC sono degli oggetti validabili. In particolare sono attività che aiutano ad abilitare e migliorare i processi primari, che sono i processi fondamentali coinvolti nello sviluppo del software. I processi di supporto forniscono infrastrutture, indicazioni e risorse per i processi primari (es documentazione, garanzia di qualità, documentazione, formazione, etc.).



- Fase vs Periodo

È meglio usare il termine "periodo" per parlare di attività a lungo termine (si può parlare anche di "sprint" nel caso dei modelli agili, se sequenziale si parla di una singola attività di sviluppo e di supporto). Quando si adotta un tipo di progettazione non sequenziale, normalmente nei periodi eseguiamo più cose/azioni. Simultaneamente, si svolge attività di analisi di requisiti, progettazione, documentazione e codifica. Quindi, non è ragionevole dare un nome specifico alla fase, in quanto si eseguono più attività in modo sovrapposto.

L'obiettivo RTB possedendo i requisiti e le tecnologie, richiede di avere una comprensione empirica dei requisiti lato soluzione, grazie all'uso delle tecnologie. Gli obiettivi della RTB sono i seguenti:

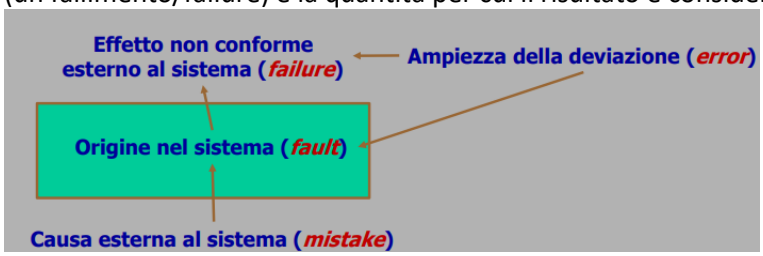
Baseline di requisiti e tecnologie
-->
comprensione dei requisiti di lato soluzione
approfondimento e finalizzazione dei requisiti di lato utente

Dunque, l'analista è fondamentale anche nel periodo pre-RTB, in quanto serve per determinare che l'analisi capita fino a quel momento concordi con quanto "scoperto" dal PoC. Finché il PoC, l'analisi dei requisiti non si può chiudere prima della chiusura totale.

Verifica e validazione: analisi dinamica (Vardanega)

(Eventuali approfondimenti:
https://www.math.unipd.it/~tullio/IS-1/2008/Approfondimenti/IEEE_Computer_41-8.pdf
https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf
http://selab.netlab.uky.edu/homepage/musa-quantify-sw-test.pdf
https://www.cin.ufpe.br/~in953/lectures/papers/TheCostOfErrorsInSoftwareDevelopmentEvidenceFromIndustry.pdf)

La verifica cerca di ricostruire la catena causale delle azioni, distinguendo in modo disciplinato tra l'azione umana (errore/mistake), la sua manifestazione nell'hardware software (difetto/fault), il risultato successivo (un fallimento/failure) e la quantità per cui il risultato è considerato scorretto (errore/error).



L'analisi dinamica consiste nell'esecuzione di varie prove sul programma riguardanti le sue esecuzioni. Ognuna di questa cerca di studiare il comportamento di singole parti di codice su un insieme finito di casi (considerando che il dominio di tutte le esecuzioni possibili è spesso infinito; occorre quindi ridurlo senza omettere possibilità di "fare prove").
Dato che il dominio di tutte le esecuzioni possibili tende all'infinito i test non garantiscono l'esaustività e infatti un test cerca di produrre un esito decidibile (oracolo) da verificare rispetto al comportamento atteso.

Dunque l'analisi dinamica può rilevare la presenza di malfunzionamenti ma non può dimostrarne l'assenza.
Ciascun caso di prova specifica:

- I valori di ingresso
- Lo stato iniziale del sistema
- L'effetto atteso (oracolo) che permette di decidere l'esito dell'esecuzione

Il test è un oggetto decidibile; l'oggetto della prova può essere:

- Il sistema nel suo complesso
  - (TS – Test di Sistema)
- Parti di esso, in relazione funzionale, d'uso, di comportamento, di struttura, tra loro
  - (TI – Test di Integrazione)
- Singole unità, considerate individualmente
  - (TU – Test di Unità)

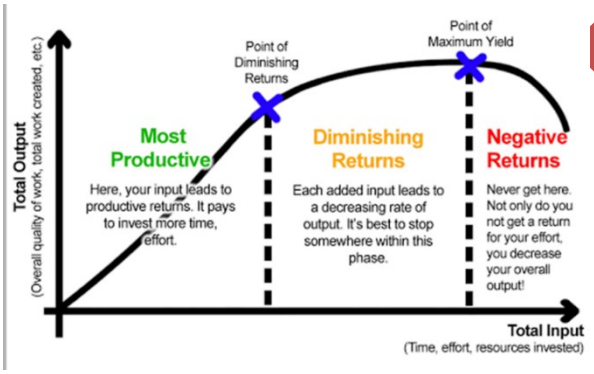
Per essere decidibile, deve essere una procedura totalmente automatizzabile. Nei nostri documenti:

- il Piano di Qualifica fissa gli obiettivi *minimi* di qualità per le prove da effettuare (con checklist opportuna di risultati, con obiettivi *quantitativi* di qualità di prodotto e di processo). Esso fissa la strategia di prova test e la correla con il piano delle attività da svolgere;
- il Piano di Progetto specifica la quantità *massima* di risorse per la verifica e per i test (in termini di tempo e di risorse effettive).

L'andamento del piano si capisce dal cruscotto, essendo strumento oggettivo ed imparziale.

La strategia di prova deve bilanciare costi e benefici, determinando la quantità minima di casi sufficiente a garantire la qualità attesa e facendo attenzione alla legge del rendimento decrescente,

Detta anche "law of diminishing returns", afferma che se una variabile di input viene aumentata mentre gli altri input vengono mantenuti costanti, alla fine si arriverà a un punto in cui il rendimento marginale dell'input aggiuntivo inizierà a diminuire.



Nell'ambito del testing del software, viene spesso utilizzata per descrivere la relazione tra la quantità di test eseguiti e il numero di bug trovati. Man mano che si esegue un numero sempre maggiore di test, la probabilità di trovare nuovi bug diminuisce e il costo per trovarli e risolverli aumenta. A un certo punto, il costo aggiuntivo della ricerca e della correzione dei bug inizia a superare i benefici che ne derivano.
Ciò significa che esiste un limite al valore di ulteriori test ed è importante determinare quando tale limite è stato raggiunto, in modo da concentrare gli sforzi di test sulle aree più critiche del sistema, dove il rapporto costi-benefici è favorevole.

L'obiettivo dei test è eseguire programmi con l'intento di trovarvi difetti, quindi potenzialmente pensarli per far fallire il programma (*failure*/malfunzionamenti). Essi vanno eseguiti in parallelo alla codifica, essendo parte essenziale della verifica e misura della qualità; la loro realizzazione ed esecuzione deve essere facile fin da subito.

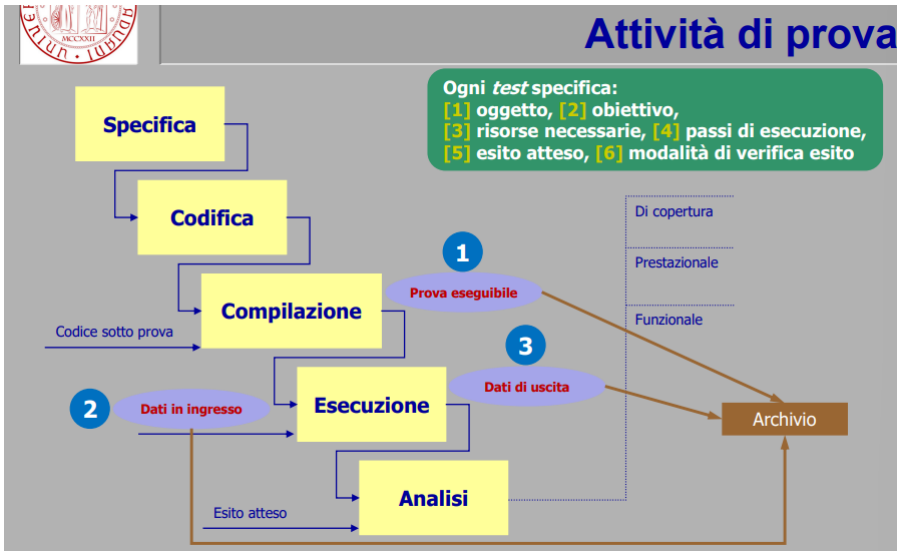
Le prove devono essere *riproducibili* per accertare:
- Buon esito di correzione dei malfunzionamenti osservati
- Funzionamento non perturbato dall'avanzare della codifica

Le prove sono *costose*:
- Richiedono molte risorse (tempo, persone, infrastrutture)
- Vanno governate con efficienza ed efficacia
- Richiedono cicli di analisi, progettazione, codifica, correzione

Si tengano in mente varie affermazioni:
- Dijkstra
  - Il test di un programma può rilevare la presenza di malfunzionamenti, ma non può dimostrarne l'assenza
- Teorema di Howden
  - Non esiste un algoritmo che, dato un programma P, generi per esso un test finito ideale (definito da criteri affidabili e validi)
- Teorema di Weyuker
  - Dato un programma P, sono indecidibili i seguenti problemi λ
    - ∃ ingresso che causi l'esecuzione di un dato comando di P?
    - ∃ ingresso che causi l'esecuzione di una data condizione di P?
    - ∃ ingresso che causi l'esecuzione di ogni comando / condizione / cammino di P?

Alcuni principi alla base delle attività di testing (Bertrand Meyer):
- Testare un programma significa provare a farlo fallire
- I test non sostituiscono le specifiche
- Ogni esecuzione fallita deve produrre un caso di test, sempre incluso nella suite di test del progetto
- Qualsiasi strategia di test dovrebbe includere un processo di test riproducibile e riproducibile ed essere valutata oggettivamente con criteri espliciti
- La qualità più importante di una strategia di test è il numero di difetti che scopre nel tempo

Scritto da Gabriel



Ogni test ha una serie di elementi:

- Caso di prova (test case)
  - Oggetto di prova, input richiesto, output atteso, ambiente di esecuzione e stato iniziale, step di esecuzione
- Batteria di prove (test suite)
  - Insieme di casi di prova
- Procedura di prova
  - Procedimento automatizzabile per eseguire prove e analizzarne i risultati
- Prova
  - Esecuzione automatica di procedura di prova

Nel contesto del testing del software, un oracolo è un meccanismo o metodo utilizzato per determinare e convalidare l'output di un system under test (SUT). Esso viene applicato da agenti automatici, per velocizzare la convalida e renderla oggettiva. Il problema dell'oracolo si riferisce alla difficoltà di determinare se l'uscita di un sistema è corretta, soprattutto quando l'uscita corretta non è nota o non può essere determinata facilmente.

Gli oracoli dovrebbero essere parte del testo del programma, come contratti. Possono essere prodotti in vari modi:

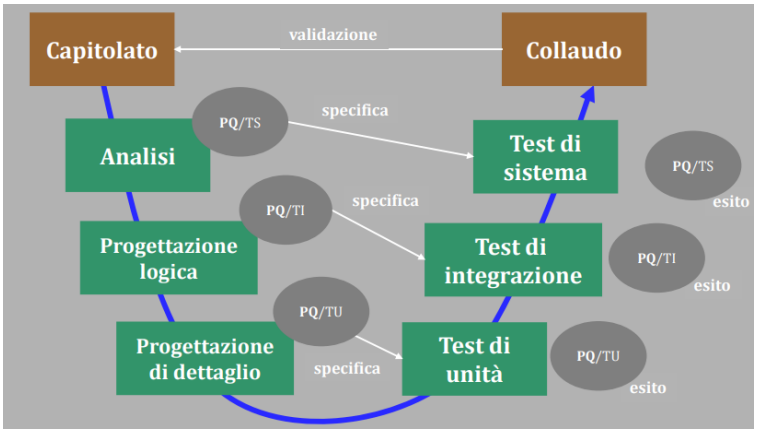
- Sulla base delle specifiche funzionali
- Entro prove semplici (facilmente decidibili)
- Tramite l'uso di componenti terze e fidate

A supporto dell'analisi dinamica vengono usati tre strumenti già visti sopra (qui riportati per comodità):

- un logger, che è un componente non intrusivo utilizzato per registrare le informazioni sull'esecuzione di un sistema (confronto l'esito del test e va salvato all'interno di un repo);
- un driver, che è un componente attivo utilizzato per interagire con il sistema in fase di test (non esegue da sola, ma viene chiamata con appositi input);
- uno stub, che è un componente passivo utilizzato per simulare il comportamento di una specifica componente.

Il V Model a fianco specifica che:

- i requisiti utente esplodono i requisiti lato soluzione (collaudo = requisiti del capitolato/sistema = tutti i requisiti);
- i test di unità sono decisi dalla progettazione di dettaglio, definiti in un'appendice del PdQ ("questi sono i test e questi sono gli output");
- i test di sistema sono decisi dopo l'Analisi dei Requisiti, definiti in un'appendice del PdQ.



I TU/Test di Unità si occupano di testare le unità software e i relativi moduli e vanno definiti durante la progettazione di dettaglio. Essi non vanno tagliati perché non dobbiamo essere ottimisti.
Il compito del programmatore è fare in tempo limitato e in modo decidibile ed eseguire su moduli (classi, spesso) senza rischio di incomprensioni/ritardi (il confine dell'unità si capisce dal design).

I test di unità sono diversi:

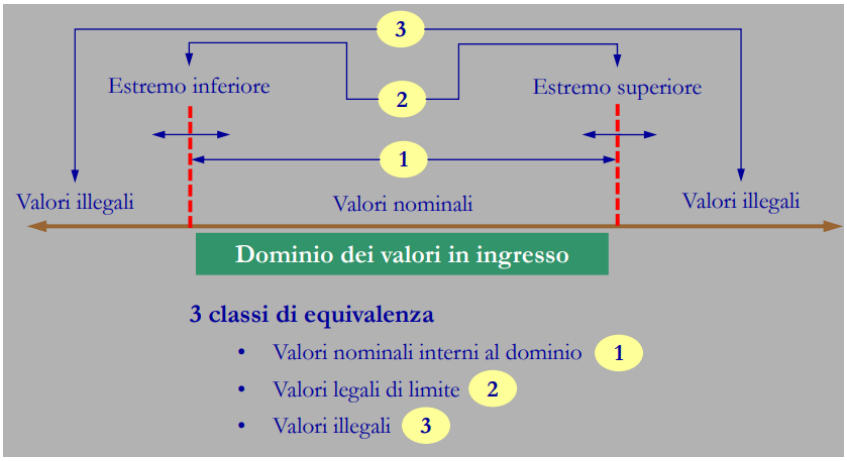
- Test funzionali (black box) → Non guardiamo dentro
    - Fanno riferimento solo alla specifica dell'unità
        - Lo fa la progettazione rispetto ai requisiti
    - Chi testa conosce solo gli input/output attesi e non la struttura interna o implementazione
    - Utilizza dati di ingresso che provochino specifici esiti
    - Dati di ingresso che producano stesso comportamento funzionale formano un caso di prova
        - Tali dati formano *classi di equivalenza*, da cui si estraggono campioni
    - Contribuiscono cumulativamente al *requirements coverage*, in crescendo fino al 100%
        - Misura di quanti requisiti funzionali siano soddisfatti dal codice prodotto
    - Non valutano la logica interna dell'unità
        - Questo tipo di difetti va scovato con altro tipo di test
    - Occorre ridurre gi input senza ridurre valore

Nel testing del software, una *classe di equivalenza* è un insieme di valori di input che possono essere trattati come un singolo caso di test. L'idea alla base dell'uso delle classi di equivalenza è quella di ridurre il numero di casi di test che devono essere eseguiti, garantendo comunque che il software sia testato in modo approfondito. Testare ogni possibile input può essere inattuabile, soprattutto se il dominio degli input è ampio o se ci sono molti modi diversi in cui il software può essere usato.

Ad esempio, se si sta testando una funzione che accetta un input di tipo intero, si può definire una classe di equivalenza per gli interi validi, un'altra per gli interi negativi e un'altra per lo zero. Testando un valore rappresentativo di ciascuna classe di equivalenza, si può essere ragionevolmente sicuri che la funzione si comporti correttamente per tutti gli interi, invece di doverla testare con ogni singolo intero.

In generale, esistono tre tipi di classi di equivalenza:

- classi valide: input che ci si aspetta siano accettati dal software;
- classi non valide: input che ci si aspetta siano rifiutati dal software;
- classi borderline: input che si trovano al limite tra l'accettazione e il rifiuto da parte del software.



Esistono cinque intervalli, non tre come sembrerebbe:

- illegali al di fuori dei confini (due, illegali inferiori/superiori)
- illegali dentro i confini (uno, illegali interni)
- gli altri tre detti (equivalenti alle classi di equivalenza)

Conclusione Verifica e validazione: analisi dinamica (Vardanega)

Le classi di equivalenza raggruppano dati con il medesimo comportamento ma, rappresentano casi limitati.
Da soli i test funzionali non bastano per accertare la correttezza della logica interna e vengono affiancati da:

- Test strutturali (white box)
  - Verifica la logica interna del codice dell'unità cercando massima structural coverage
    - Che ha più dimensioni ed è complementare alla requirements coverage
  - Ogni singolo caso di prova deve attivare un singolo cammino di esecuzione all'interno dell'unità
    - Creando le condizioni logiche che causano la scelta di quel cammino
  - Ogni caso di prova è costituito dall'insieme di dati di ingresso e di configurazione di ambiente che produce uno specifico cammino d'esecuzione
  - La loro esecuzione può essere facilitata dall'uso di debugger.

Qui esistono varie dimensioni di coverage:

- Si ha *Statement Coverage* al 100%
  - Quando l'insieme di test effettuati sull'unità esegue almeno una volta tutti i comandi (statement) dell'unità, con esito corretto (e senza danno)

$$\textbf{Statement Coverage} = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements} \times 100$$

- Si ha *Branch Coverage* al 100%
  - Quando ciascun ramo (then/else) del flusso di controllo dell'unità viene attraversato almeno una volta da un test, con esito corretto
    - Almeno un test va nel vero o nel falso; individua cammini "lotteria", cioè che non si sa precisamente come si possano comportare in tutte le situazioni

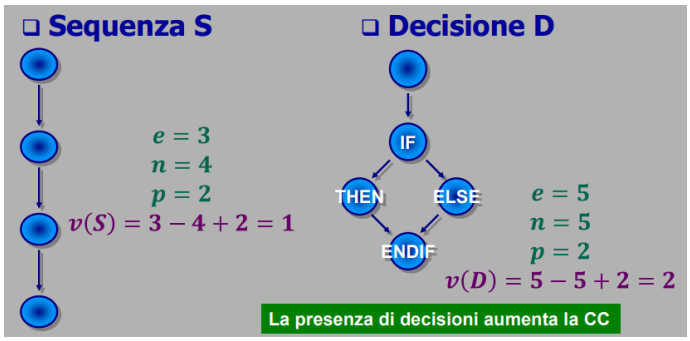
$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

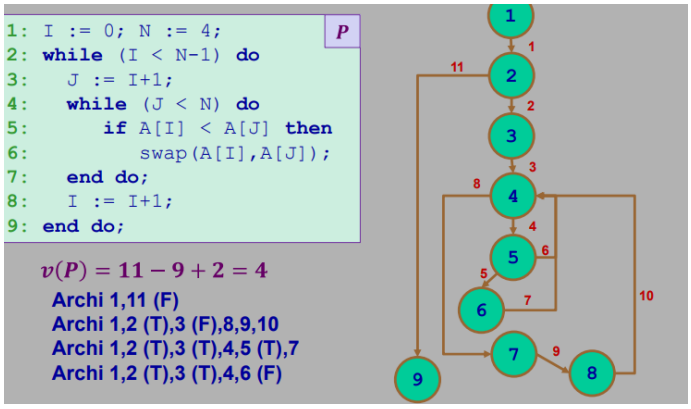
  - Il numero di percorsi linearmente indipendenti in una esecuzione con singolo ingresso e singola uscita (unità) è detto *complessità ciclomatica/cyclomatic complexity/CC*.
    - Inizialmente 1, incrementata da branch, salti, e iterazioni

La CC del grafo G che descrive i flussi d'esecuzione all'interno dell'unità, è $v(G) = e - n + p$

- e numero degli archi in G (flusso tra comandi)
- n numero dei nodi in G (espressioni o comandi)
- p numero delle componenti connesse da ogni arco (l'esecuzione sequenziale ha $p = 2$, avendo 1 predecessore e 1 successore per ogni arco)

- I nodi sono statement singoli, mentre gli archi sono avanzamento tra statement
- La sequenza sono nodi connessi da archi con un solo successore (connessioni di statement)
  - Si ha nell'esempio seguente un solo cammino e una struttura con un singolo *if-else* (immagine di sx)
  - Ciò diventa più complicato quando si ha a che fare con codice "creativo", annidato e con vari cammini (immagine di dx)





- La branch coverage è più potente della statement coverage in quanto testa per ogni istruzione condizionale entrambi i casi anche se ciò non porta ad esaminare un numero maggiore di linee di codice

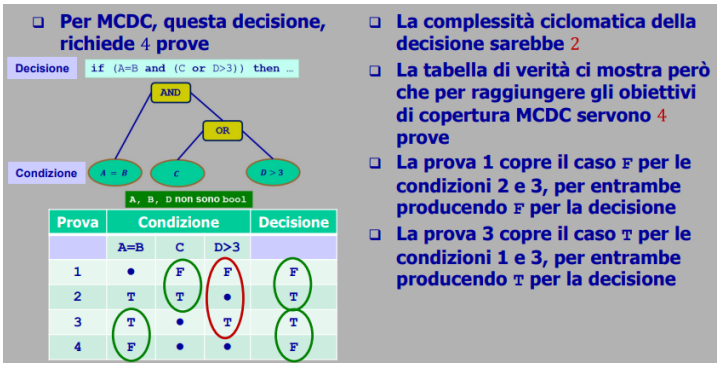
- Si ha *Decision/Condition Coverage* al 100%
  - Quando ogni condizione della decisione (branch) assume almeno una volta entrambi i valori di verità in un test dedicato
  - Metrica più precisa della branch coverage
  - Necessaria in presenza di espressioni di decisione complesse
    - Il meccanismo di controllo di cammini deve essere seriamente implementato

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

  - È la più stringente in quanto non basta testare la decisione finale di un'espressione condizionale ma obbliga anche a testare ogni decisione atomica al suo interno(condizione). Spesso massimizzare questa tipologia di coverage diventa troppo oneroso

  - Condizione: espressione booleana semplice
    - Ciascuna condizione va soggetta a prove che producano T/F nella decisione almeno una volta ciascuna
  - Decisione (branch): espressione composta da più condizioni
    - Ciascuna decisione va soggetta a prove che producano T/F almeno una volta ciascuna

La tecnica *Modified Condition/Decision Coverage (MCDC)* è una tecnica che non testa ogni singola condizione ma garantisce che ogni possibile output ottenibile a seguito di una decisione sia coperto da test.



Si richiedono tabelle di verità sulle condizioni logiche in cui si intende controllare il numero di condizioni minimo (uso come combinazioni delle celle ottimizzate, cioè collassando più esiti in uno).

Il principio del test di integrazione è mettere insieme solo cose buone, cioè mettendo insieme più unità testate funzionalmente e strutturalmente.
Essi vengono definiti durante la progettazione architetturale e si basano sui componenti in essa specificati.
Per definire i test di integrazione è necessario selezionare quali funzionalità integrare individuandone le componenti coinvolte e ordinandole per dipendenze crescenti.

In particolare:
- Quando metto insieme le unità, testo i dati che passano l'una all'altra e chi comanda chi
- Si applica alle componenti individuate nel design architetturale
    - La loro integrazione totale costituisce il sistema completo
- Rileva difetti di progettazione architetturale o bassa qualità di TU
    - I dati scambiati attraverso ciascuna interfaccia concordano con la specifica?
    - Tutti i flussi di controllo specificati sono stati verificati corretti?
- Assembla incrementalmente, a ogni passo aumentando il valore funzionale disponibile
    - Integrando componenti nuove in insiemi già verificati, i difetti rilevati da TI su tale passo sono più probabilmente da attribuirsi all'ultima aggiunta
- Assicura che ogni passo di integrazione sia reversibile
    - Potendo sempre retrocedere a un precedente stato sicuro (baseline)

I percorsi di esecuzione sono divisi in alberi e grafi.
- Gli alberi sono percorsi di esecuzione completa in cui si ha relazione gerarchica tra diversi componenti o moduli di un sistema
    - Il vertice dell'albero rappresenta il componente o modulo di livello più alto e ogni ramo rappresenta un componente o modulo di livello inferiore collegato al componente di livello superiore
    - Ogni nodo foglia di un albero rappresenta un singolo componente o modulo e l'albero nel suo complesso rappresenta l'intero sistema
- Un grafo è una rappresentazione a diagramma delle interconnessioni tra i diversi componenti o moduli di un sistema
    - Esso è composto da nodi, che rappresentano componenti o moduli, e da archi, che rappresentano le connessioni o le relazioni tra tali componenti o moduli
    - Il grafo nel suo complesso rappresenta l'intero sistema e i nodi e gli spigoli vengono utilizzati per identificare le relazioni e le interazioni tra i diversi componenti o moduli

La continuous integration assicura di inserire solo frammenti buoni di codice, per cui si integrano solo pezzi appartenenti ad insiemi ben verificati ed eventuali difetti sono certamente dovuti alle ultime parti aggiunte.
Ne esistono di due tipi:
- Integrazione incrementale di tipo bottom-up (dalle foglie alla radice)
    - Le foglie sono gli oggetti non chiamati, mentre la radice spesso è un main
    - Si integrano prima le parti con minore dipendenza funzionale e maggiore utilità
        - Quelle che sono molto chiamate/attivate ma chiamano/attivano poco o nulla
        - Quelle più interne al sistema, meno visibili a livello utente
    - Questa strategia richiede pochi stub ma ritarda la messa a disposizione di funzionalità ad alto livello; utilizza driver (perché non ha subito il main)
    - Strategia spesso scelta da chi non ha interazione frequente con il proponente

- Integrazione incrementale di tipo top-down (dalla radice alle foglie)
    - Si sviluppano e si integrano prima le componenti con maggiori dipendenze d'uso e quindi maggiore valore aggiunto esterno
        - Quelle che chiamano/attivano più di quanto siano chiamate/attivate

  - Questa strategia comporta l'uso di molti stub ma integra prima le funzionalità di più alto livello, più visibili all'utente
  - Integro funzioni base che assomigliano al prodotto (foglie) e, se ben create e verificate, il sistema sarà solido da subito ("dal basso")

Arrivo ai test di sistema, che verificano il comportamento dinamico del sistema completo e vengono definiti durante l'analisi dei requisiti ma hanno inizio solo con il completamento dei test di integrazione e sono precursori del collaudo:

- Si misurano in requirements coverage rispetto alla copertura misurata dai TU funzionali
- Essi devono includere tutti i casi di prova (TU, TI) che siano precedentemente falliti
- Sono inerentemente funzionali (black-box)
- Possiamo "non guardare dentro" in questo test, perché è già stato fatto da altri test

Differenza tra black e white box: https://www.tutorialspoint.com/software_engineering/se_quality_qa8.htm

Altri tipi di test (fondamentali):

- Test di regressione
  - Accertano che correzioni o estensioni su specifiche unità non danneggino il resto del sistema e che non ci sia accoppiamento
  - Si testano tutte le unità in relazione con quella toccata, verificando non venga danneggiata l'interazione con le altre
  - Consiste nella ripetizione selettiva di TU, TI e TS per accertare non siano stati introdotti errori
  - Essi vanno decisi nel momento in cui si approvano modifiche al software ma vanno eseguiti solo in seguito al superamento dei test di unità legati alle componenti coinvolte

- Test di accettazione/di validazione (collaudo)
  - Accertano il soddisfacimento dei requisiti utente alla presenza del committente
  - Possono essere definiti già a partire dal capitolato

Il punto in cui so esattamente di quanti test ho bisogno è dopo la progettazione di dettaglio è il vertice basso del modello a V. Se non lo so in basso, come farò a risalire?
Il cruscotto mi dice "per certo" i tipi di test e se siano pronti, sapendo dove sono per prepararli (numero di test pronti rispetto al totale).

Esse sono le *misure di copertura*, che dicono quanto le prove esercitano il prodotto.

- La copertura funzionale rispetto ai requisiti del prodotto
- La copertura strutturale rispetto alla sua logica interna

Esse quantificano la bontà della campagna di test.

- La copertura del 100% complessivo non garantisce assenza di difetti
- Raggiungere il 100% di copertura complessiva può non essere possibile
- Per ragioni di tempo/costo, di codifica, di strumenti

Gli obiettivi di copertura sono specificati nel PdQ.

Le *misure di maturità* del prodotto servono:

- Per valutare il grado di evoluzione del prodotto
  - Quanto il prodotto migliora in seguito alle prove
  - Quanto diminuisce la densità dei difetti
  - Quanto può costare la scoperta del prossimo difetto
- Le tecniche disponibili sono spesso empiriche
  - Ma sono stati fatti buoni passi avanti (vedi bibliografia)
- Serve definire un modello ideale di crescita della maturità
  - Modello base: il numero di difetti del SW è una costante iniziale
  - Modello logaritmico: le modifiche possono introdurre difetti

Scritto da Gabriel

Quando conviene smettere i test?

- Il contatore del numero di test è nel cruscotto, che deve convergere prima o poi ad asintoti (rette).
- Più si abbassa l'errore, meglio è; questo dipende anche da tempi/costi, sorpassando costi sui benefici.

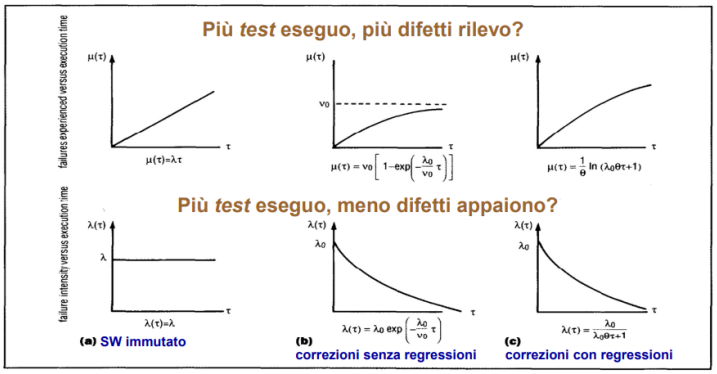
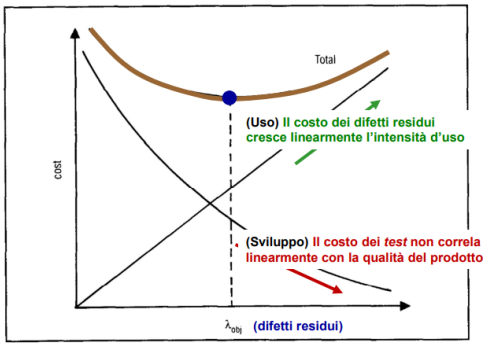


Figure 1. Three useful software-reliability models: **(a)** static, **(b)** basic, and **(c)** logarithmic Poisson. These are shown comparing both failures experienced versus execution time and failure intensity versus execution time.

Occorre avere consapevolezza a livello di costi di quando fermarsi sul testing, oltre a costi anche rischi, a livello di rilascio, standard e regole.
Seguiamo questo link: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=28120&tag=1

Quando si tolgono molti difetti, il costo si riduce in proporzione al suo uso, che capisce "se sta facendo o meno danni". Di fatto, si produce un calcolo che decide il costo minimo a parità di sviluppo e che cerchi di trovare difetti in base all'intensità d'uso e con un processo incrementale alla qualità del prodotto ottenuto.



Bisogna decidere quali densità di difetti residui sia accettabile, che minimizzi il costo d'Uso entro il costo di Sviluppo sostenibile

Si considerino questi punti:

- Gli errori gravi spesso sono meno costosi di quelli più lievi
  - I primi sono trattati con urgenza, i secondi in modo più trascurato (technical debt)
- Correggere gli errori è molto costoso quando comporta modifiche architetturali
- Il costo degli errori non corretti cresce esponenzialmente con l'avanzare del progetto
- Il numero di errori rilevati cresce linearmente con la durata del progetto
- Usare bene Continuous Integration focalizza meglio le attività di sviluppo e amplia l'intensità di test
- Il costo del progetto cresce esponenzialmente con la realizzazione del prodotto

Il TD/Technical Debt può essere consapevole o meno (meglio che sia il primo).

L'atteggiamento rispetto al TD ha quattro varianti

- PD: consapevolezza
- RD: superbia
- PI: umiltà
- RI: incompetenza

Esercitazione in preparazione alla prova scritta (1 di 2) (Cardin)

Da questo momento, le revisioni sono aperte; il PoC non è una demo, ma serve a noi. Le tecnologie devono integrarsi tra di loro in una cosa utile per noi (non importa quante funzionalità, ma quante tecnologie). Non interessa la progettazione, il pattern o altro ma solo qualcosa di utile per noi.

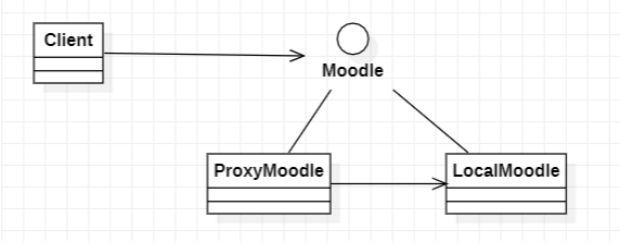
Esami

- Quiz (Risposta multipla /Vero-Falso/Esercizio) in 10 minuti
    - Tre domande di Vardanega + Tre domande di Cardin
- Parte collaborativa (gruppi di 3/4 persone)
    - Diagrammi UML + Diagrammi di attività + Terzo esercizio

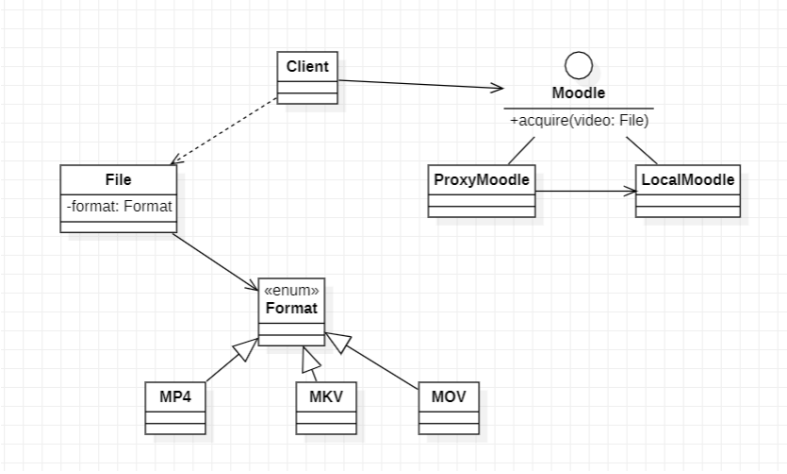
Dal pdf E02, primo esercizio (da leggere bene).

Pattern individuati dal testo:
- Proxy
- Observer

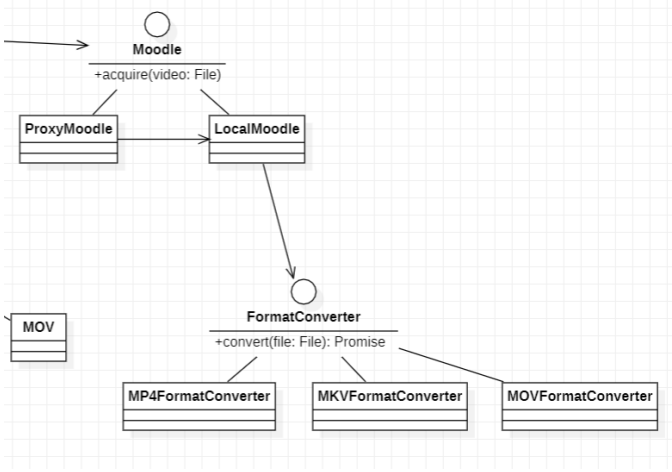
Di fatto, abbiamo un collegamento del client con il server che ha un collegamento locale e remoto, quindi:



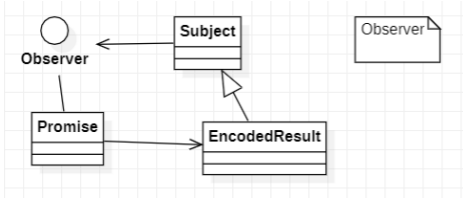
Essendoci vari tipi di file, modelliamo una enumerazione:



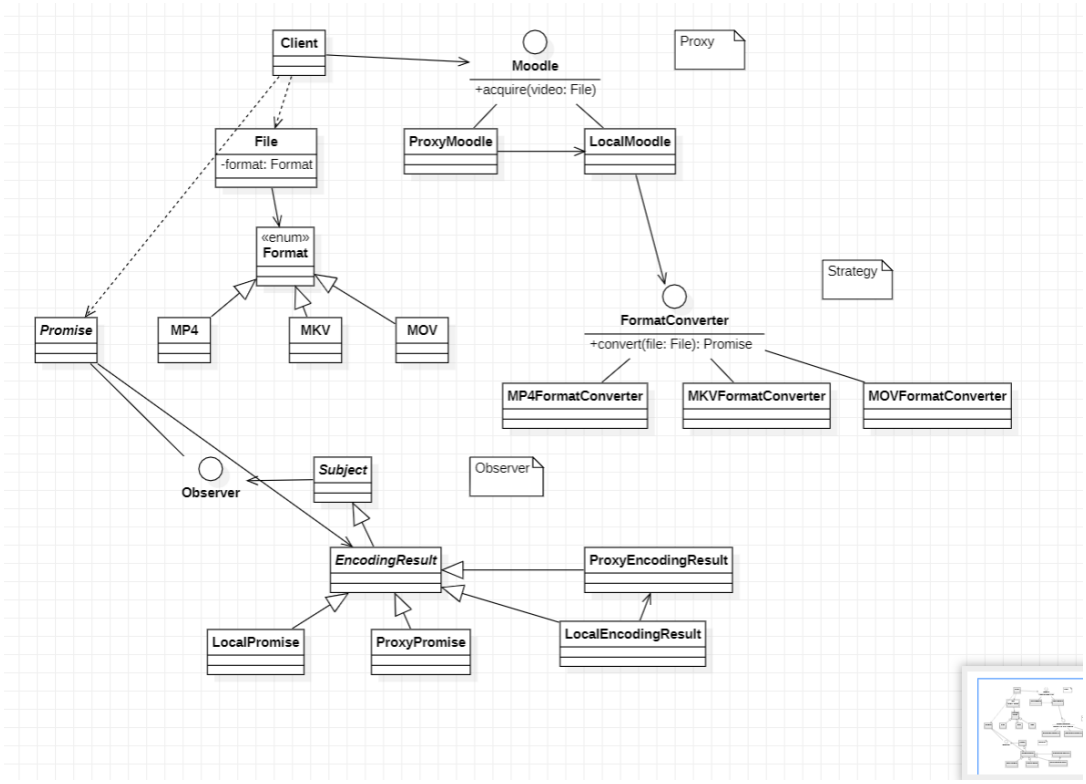
Avrò poi un algoritmo che trasforma i tre algoritmi nel formato corretto, convertendo un formato promesso:



Ciò che deve essere notificato (Observer) è Promise.



Avrò una chiamata da un'altra parte e uso Proxy. Non uso un collegamento con Proxy; le classi Proxy vengono messe come astratte; successivamente implementiamo un proxy codificato per il risultato da ottenere.



Ogni volta che c'è l'Observer, non per forza ci vuole il Proxy.

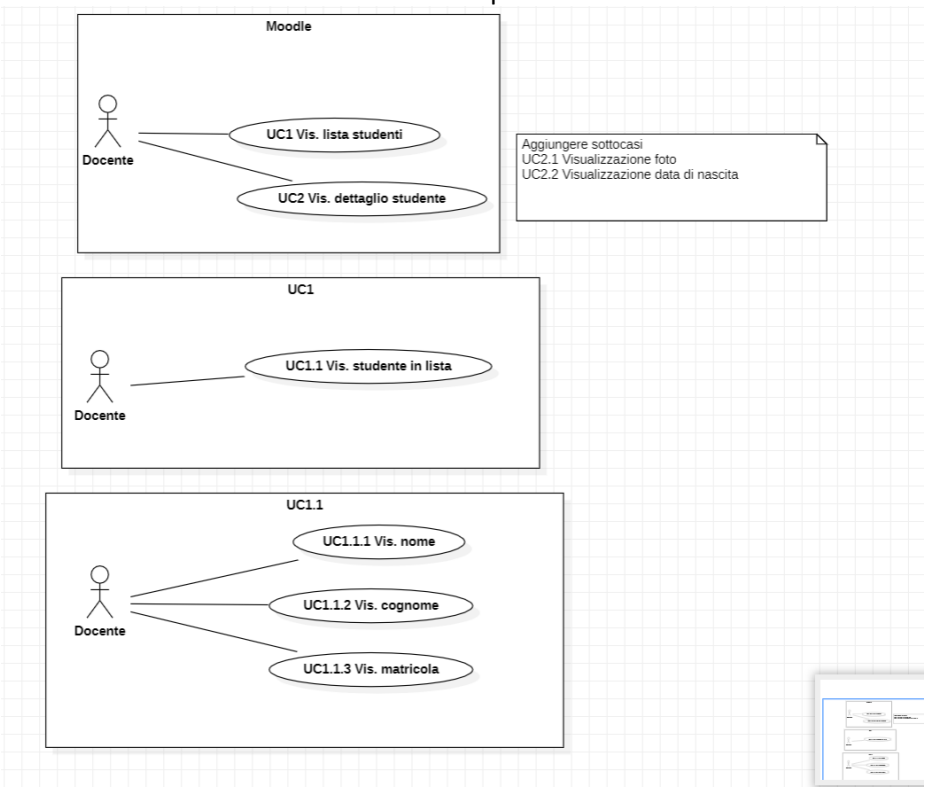
Esercizio 3 (2 punti)

Descrizione

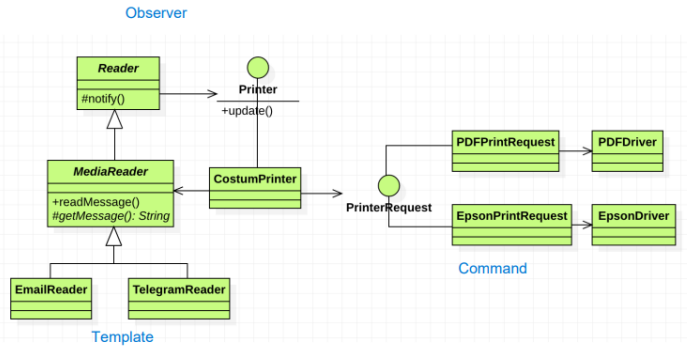
Moodle permette ai docenti di visualizzare la lista degli studenti iscritti ad un corso. All'interno della lista, ogni studente è rappresentato dal proprio nome, cognome e dalla matricola. Selezionando uno studente dalla lista, si possono invece visualizzare le sue informazioni di dettaglio, che riportano anche una foto dello studente e la sua data di nascita.

Utilizzando un diagramma dei casi d'uso, si modelli quanto descritto. Non è necessaria alcuna descrizione testuale del diagramma.

Inseriamo casi di visualizzazione dei singoli dati, poi degli studenti in lista, lo studente alla lista e successivi sottocasi per individuazione.
I casi d'uso trovano funzionalità e non implementazioni.



Domanda da quiz:



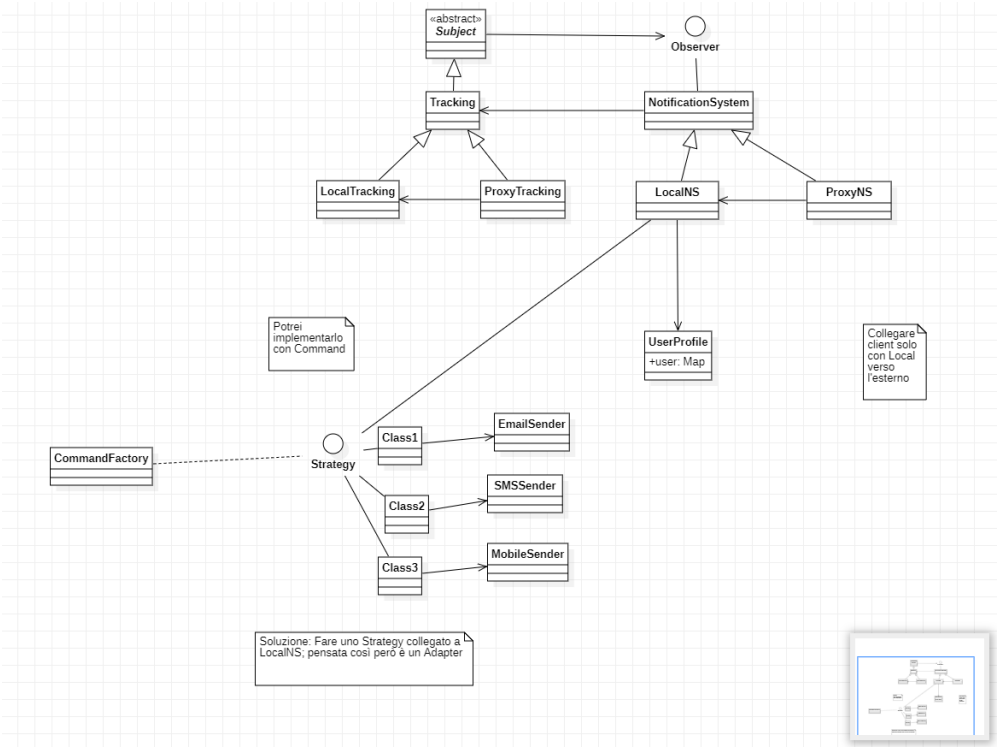
Quali pattern possono essere individuati al suo interno?

1) Observer pattern
2) Strategy pattern
3) Proxy pattern
4) Command pattern
5) Abstract Factory pattern
6) Template Method pattern

Descrizione

Amazon è un sistema distribuito di enorme complessità. Per mantenere alta la qualità dell'esperienza utente, il sistema di basa su un'architettura *reactive*, costruita completamente sul *pattern publish – subscribe*. Apprezzato dagli utenti è il sistema di notifiche sullo stato degli ordini. Ogni qualvolta un pacco parte alla volta di un utente finale, il corriere inserisce una notifica nel sistema di *tracking*. Questo avvisa il sistema di notifica, posizionato in un altro nodo della rete. Le notifiche possibili sono diverse: e-mail, messaggio SMS e notifica sull'applicazione *mobile* di Amazon. Sulla base delle impostazioni del profilo dell'utente, il sistema decide quali media utilizzare. Per permettere al sistema di evolvere semplicemente, l'invio delle notifiche avviene utilizzando un'interfaccia unica.

Si modelli tale sistema mediante un diagramma delle classi, comprensivo dei *design pattern* a esso pertinenti.

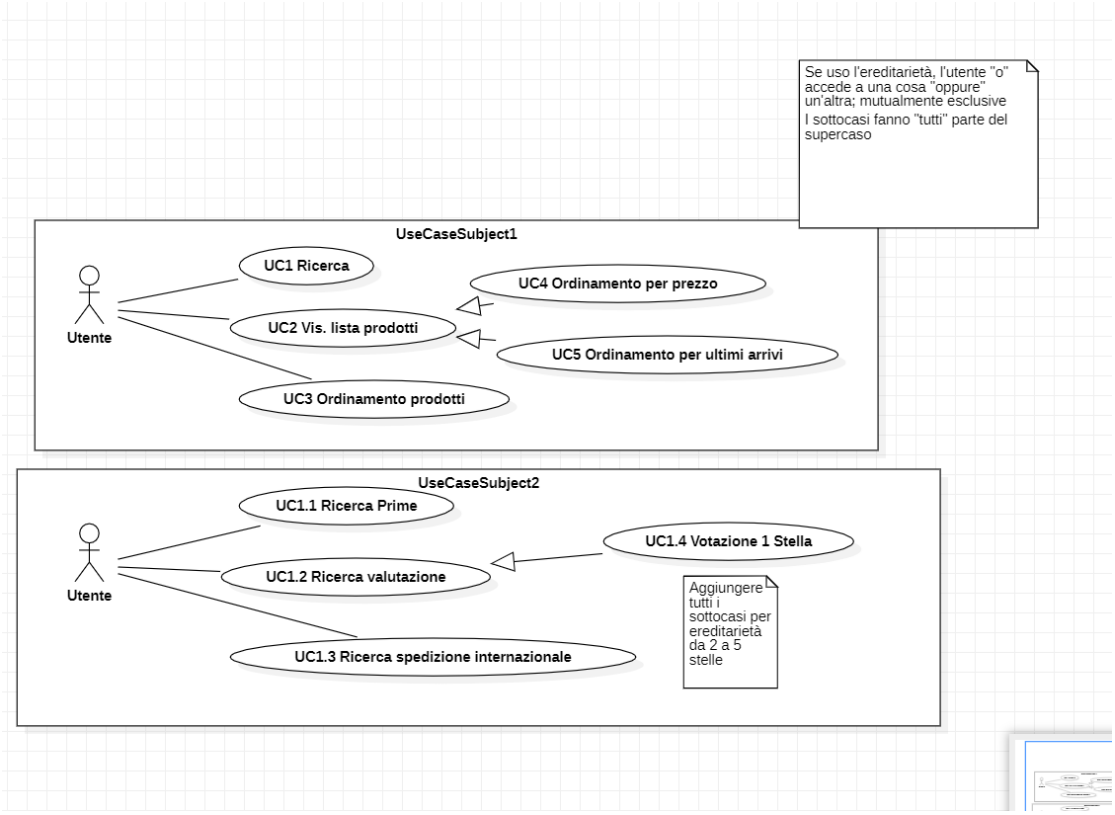


Esercizio 3 (3 punti)

Descrizione

Il sito di Amazon permette di effettuare una ricerca avanzata dei prodotti disponibili a catalogo selezionando sia il fatto che sia il prodotto disponga della spedizione *Prime*, sia la media delle recensioni positive dei clienti (1 … 5 stelle), sia se per il prodotto è disponibile la spedizione internazionale. Il risultato della ricerca è una lista di prodotti, che può essere ordinata per prezzo crescente o per "ultimi arrivi".

Si utilizzino i diagrammi dei casi d'uso per modellare gli scenari sopra descritti. Non ne è richiesta la descrizione testuale.



Incontro informativo sugli stage curricolari (Vardanega)

Lo stage/internship/tirocinio fa parte del piano di studi (con un certo periodo e un certo numero di crediti, che sono tolti da altri corsi dentro il CCS) ed è obbligatorio per l'esame di laurea. Viene inteso come attività formativa, applicando le proprie competenze nel mondo reale.
Normalmente, come in informatica del mondo vero, si fa qualcosa simile a cose già conosciute, ma anche no naturalmente.

La tesi, fatta alla fine del percorso formativo, dimostra le competenze acquisite prima e dopo il percorso regolare per dimostrare il merito dell'esame di laurea. Non è produrre scienza, ma dimostrare di essere capace di affrontare problemi soggettivamente nuovi con un metodo scientifico (basato sulle abilità pregresse/acquisite). Lo stage pesa 11 crediti per il vecchio ordinamento, 12 per il nuovo.
Lo stage pesa $12 * 25 = 300$ ore; $11 * 25 = 275$ ore; parliamo quindi di poco meno di 8 settimane, in linea di massima. Esso viene approssimato per eccesso al limite invalicabile a 8 settimane lavorative a tempo pieno (non più né meno).

Scritto da Gabriel

Essendo attività full immersion, il tempo viene misurato direttamente da noi. Viene svolta normalmente alla fine del percorso, dato che è full time (essendo fatta all'esterno). Anche se manca una materia, lo stage è fatta a tempo pieno e non lascia spazio per alcuna altra attività di studio.
Lo stage, a livello di curriculum, è un'attività che ne fa parte.
Il curriculum è uno strumento di presentazione:

- a livello anagrafico;
- cosa so fare (a livello di competenze) di rilevante e concreto;
    - non si parla di esami passati, ma le competenze, per un informatico, si parla di progetti reali, cose fatte, attività informatiche svolte (cose visibili fatte);
- cosa sto cercando (ambiti culturali di interesse).

L'università intende 1500 ore individuali di lavoro sia per studenti che per studenti-lavoratori (ovviamente, è obbligatorio anche per loro).
Lo stage è un'attività individuale che si attiva in modo asincrono (ognuno decide quando, dato che non è attività d'aula), rispetto alla fine (quand'è che mi voglio laureare) e rispetto all'inizio (quand'è che posso attivare lo stage).
Occorre capire quando laurearsi, in quanto serve il giusto calcolo a livello di tempo.

Quando si accede all'esame di laurea:

- occorre aver acquisito tutti i crediti formativi entro 15 giorni dall'appello di laurea

Quando sapere quando ci sono gli appelli di laurea:

- https://informatica.math.unipd.it/laurea/laurea/ (in fondo alla pagina)

Lo stage deve essere chiuso/approvato all'interno del libretto e possiede una sua valutazione

- In Uniweb viene definito come "Approvato" quando passato. Produce crediti attraverso svolgimento, completamento e lo stato detto.

Le parti dello stage:

- Entità esterna (certifica che la persona abbia svolto lo stage, attraverso il tutor aziendale)
- Entità interna (docente che svolge il ruolo di relatore e accompagna lo studente alla prova finale, definito/a anche come tutor interno)
    - Approva la proposta dello stage, riceve informazioni sullo stage regolarmente e conferma la validità di quest'ultimo
- Entità di approvazione degli stage (interagisce con tutor interno ed esterno e conferma la validità dello stage)
    - Ci mette almeno 2 giorni lavorativi; essa serve a certificare l'uscita dall'ambiente di studio
    - Per questo motivo, viene fatto un corso sulla sicurezza

La durata dello stage:

- Max 8 settimane consecutive (se svolto a tempo pieno)
- Può essere svolto a tempo parziale, ma con attenzione (con pianificazione reale e continua, non ottimistica e con dei margini)

Quindi: 10 (o poco più giorni) dall'appello di laurea meno (-) 8 settimane

- Es. dal 3 di luglio rispetto al 15 luglio di laurea = primo di maggio (o fine aprile, esagerando)
- Chi attiva lo stage in quel momento, deve soddisfare le condizioni per poterlo fare

Se durante lo stage devo fare l'esame X:

- Ovviamente, ciò viene giustificato e ammesso dallo stage stesso

Condizioni per lo stage:

Scritto da Gabriel

- Calcolare i tempi detti considerando che si tratta di attività a tempo pieno non compatibile con il fare altro
- Rispetto di atti amministrativi all'inizio e alla fine (tutor interno, tutor esterno e approvatore stage)
- L'evidenza (stato "Approvato") dell'aver svolto lo stage viene dato da SIAGAS, in cui le tre parti dette controfirmano il modulo di attivazione (che deve essere generato prima)
- Il modulo di attivazione impegna l'entità ospitante, l'ateneo e viene recepito dal corso di studi
    - Viene generata una firma digitale per convalidare questo modulo
    - Vengono date le informazioni anagrafiche e di massima per l'attivazione dello stage
    - Esso comprende anche una firma digitale nostra, presente in un portale UniPD
    - Periodo con azienda: forse 7-10 giorni di scambio dell'organizzazione
    - La discussione con questa è in carta intestata dell'azienda
- Il modulo di conclusione riguarda solo lo studente ed il corso di studi (evidenza dei crediti dello stage)

Informazioni sul dove svolgere lo stage:

- Le aziende da cercare devono svolgere uno stage di tipo "curricolare"
- Non è un'attività finalizzata all'assunzione, ma a fine didattico
- Deve essere un'attività che si avvicina ai nostri interessi, essendo un'esperienza significativa e che merita di essere aggiunta nel curriculum
- Per aiutarci nella scelta, c'è un evento annuale a fine marzo con aziende esterne che cercano personale (almeno 70/80 aziende partecipano)

Lo stage è *formativo*:

- È assicurativamente regolato (presso un'organizzazione che ha convenzione ad ospitare sancita con l'Università di Padova e dura 5 anni; questa deve esistere *prima* che lo stage abbia luogo
    - Normalmente, questa impiega 2 settimane ad essere stipulata (indagine amministrativa + procedura di firma e registrazione)
- Ha un interesse nostro e non siamo dipendenti
- Non è un tempo deciso da loro e potremmo non venire pagati; quantomeno, si può avere un rimborso spese oppure una retribuzione (a discrezione dell'azienda)
- Non c'è un vincolo geografico; se azienda estera (purché abbia convenzione con UniPD), dipende dal tipo di azienda e purché si rispetti il periodo di 8 settimane (può essere difficile)

Per chi già lavora:

- Si parla direttamente con Tullio, parlando singolarmente della singola situazione
- Chiaramente, anche la persona che lavora, è tenuta a fare lo stage

Per partecipare all'esame di laurea, occorre una tesi.

- Essa è la relazione finale di stage, cioè il documento in cui lo studente, supervisionato dal proprio relatore, descrive in modo intelligente e formativo le esperienze avute e quanto fatto.
- Esso è un documento ufficiale ed è di proprietà dello studente ed è pubblico per definizione
    - Come tali, non si hanno dati sensibili

Rispetto al progetto di SWE, si concorda tra colleghi rispetto ad un certo momento; quando si cerca lo stage; lo stato di avanzamento viene documentato in modo reale e concreto con Cardin e Tullio.

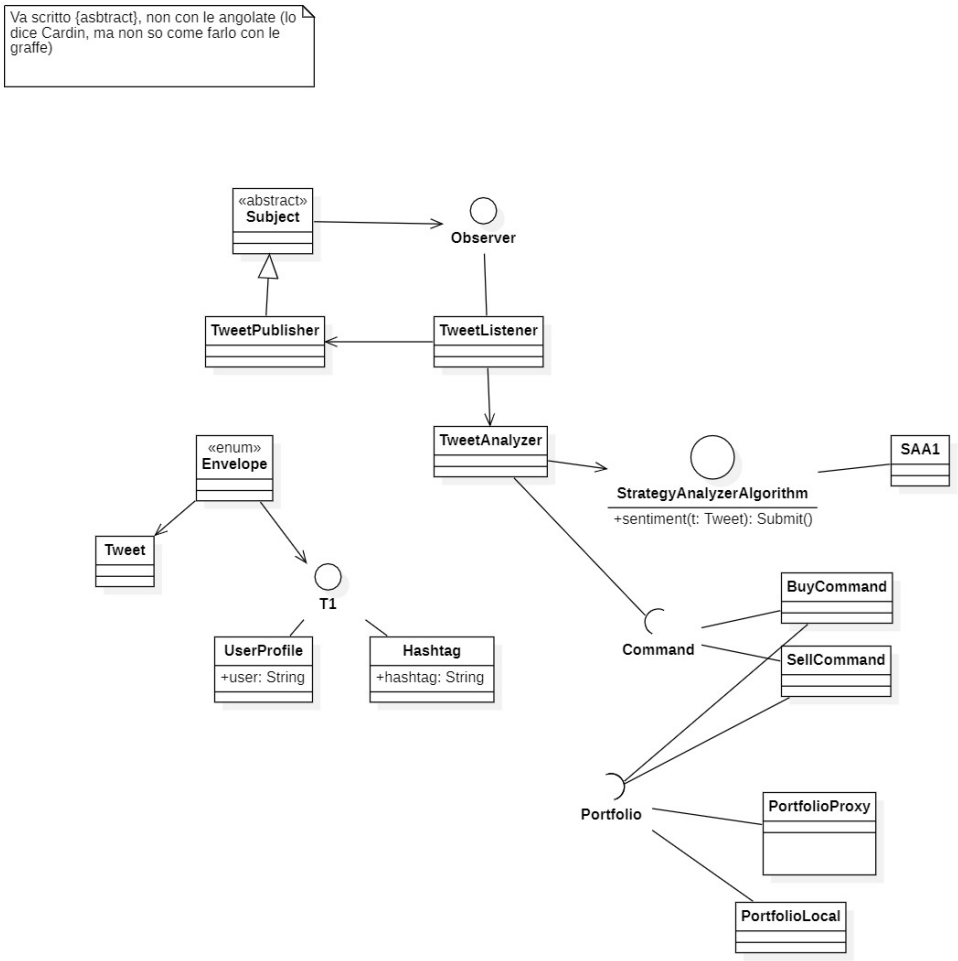
Esercitazione in preparazione alla prova scritta (2 di 2) (Cardin)

Esercizio 1 (5 punti)

Descrizione

Un'azienda di Los Angeles ha sviluppato un sistema capace di interpretare i *tweet* pubblicati su *Twitter* ed in risposta fornire delle indicazioni se comprare o vendere un insieme di azioni contenute in un *portfolio*. In particolare, l'applicazione può restare in ascolto su un utente (profilo) o su un *hashtag*. Ogni qualvolta un nuovo *tweet* viene pubblicato, il sistema utilizza un algoritmo di *sentiment analysis* sul testo. L'algoritmo, in fase prototipale, deve poter essere cambiato con il minimo sforzo nel futuro. L'applicazione resta in ascolto parallelamente sui *tweet* di più utenti o *hashtag*. Una volta ottenuto un risultato dall'algoritmo, il sistema decide se comprare o vendere una determinata azione. L'interfaccia di compravendita è unica e deve consentire di comprare o vendere un'azione usando lo stesso metodo. Il sistema di compravendita è remoto, ma viene utilizzato dall'applicazione come se fosse locale.

Si modelli tale sistema mediante un diagramma delle classi, comprensivo dei *design pattern* a esso pertinenti.

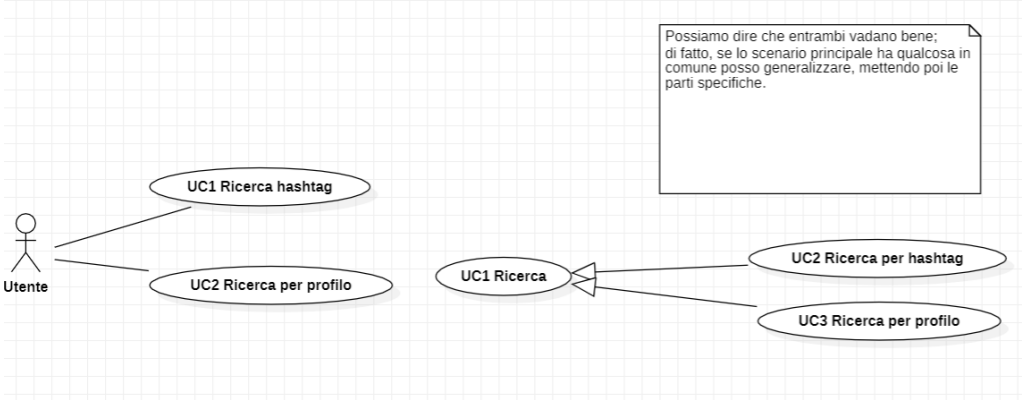


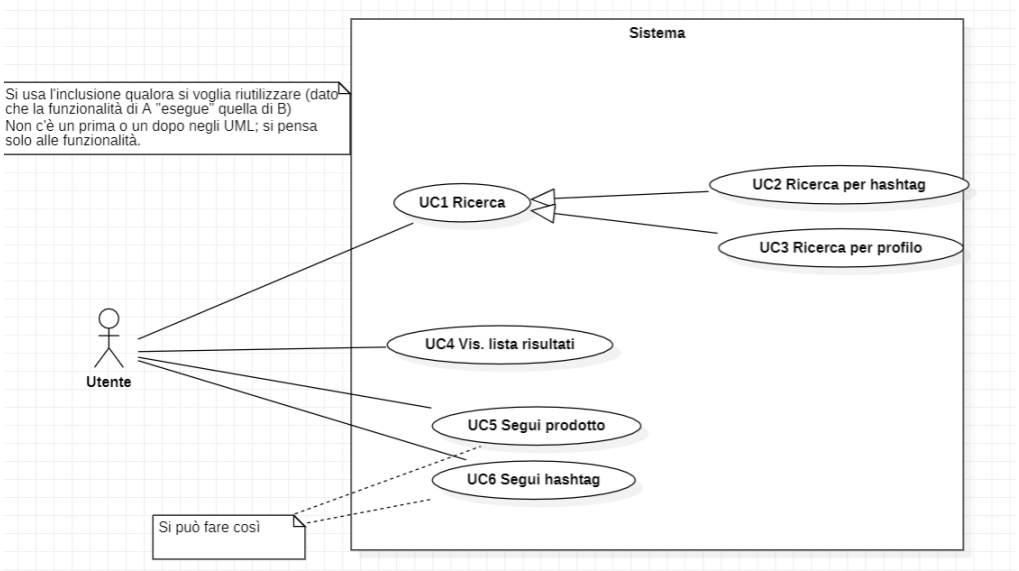
Esercizio 3 (2 punti)

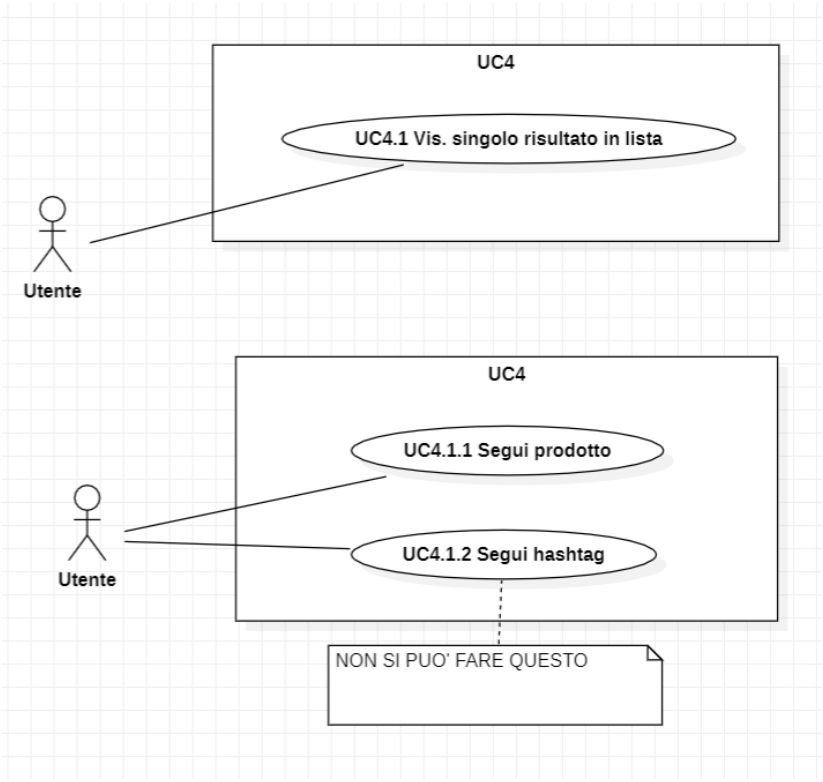
Descrizione

L'applicazione permette agli utenti di configurare i profili e gli *hashtag* da seguire e sui quali il sistema deve fare analisi. Sono disponibili due tipi di ricerca, una per tipologia. Nella lista, risultato della ricerca, è disponibile un pulsante per ogni elemento, con il quale l'utente può suggerire al sistema cosa seguire.

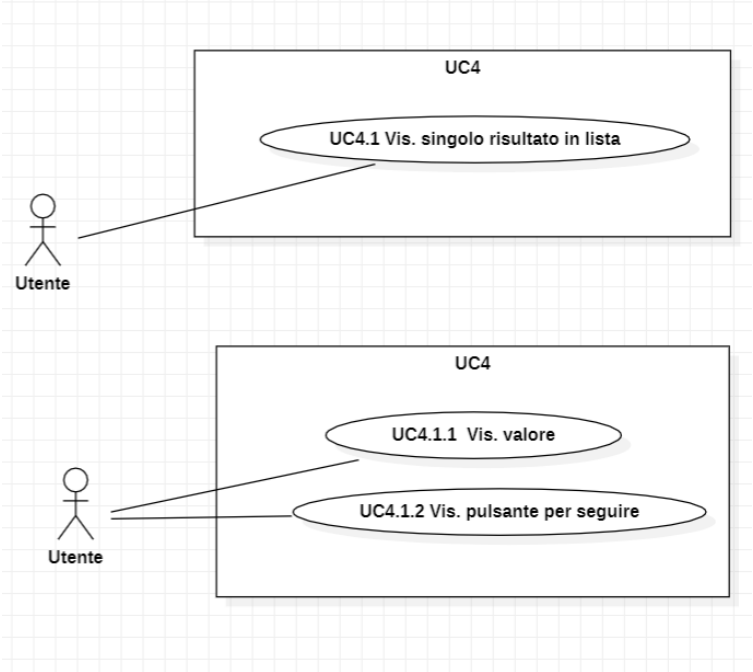
Utilizzando un diagramma dei casi d'uso, si modelli quanto descritto. Non è necessaria alcuna descrizione testuale del diagramma.







A livello di dettaglio, può avere senso fare come sotto (questo si avvicina più a livello UX):



Mai usare casi d'uso di raggruppamento, ma meglio scomporre e partizionare le funzionalità.

Esercizio 1 (6 punti)

Descrizione

Kafka è un *message broker* che implementa un modello di *logging* distribuito. Ogni log (o messaggio) ha associata una chiave ed un *payload* di tipo T. I messaggi vengono pubblicati su una struttura chiamata *topic*. Un *cluster* è composto da una serie di nodi che si suddividono il lavoro. Ogni nodo gestisce una serie di partizioni per un topic. Le partizioni sono un sottoinsieme delle chiavi possibili per i log. Un nodo è definito *master* ed è responsabile della comunicazione con i *producer* e possiede le tabelle di *routing* per dirottare ogni log alla partizione corretta. Un *producer* invia un log (chiave, valore) per un topic al nodo *master*. La comunicazione avviene via rete. Il nodo *master* utilizza un algoritmo di partizionamento per comprendere a quale altro nodo dirottare il *log*. Questo algoritmo può variare, ma data in input una chiave e un topic restituisce sempre un IP e un numero di partizione. Il nodo *master* invia al nodo deputato il log, che lo salva in un file sequenziale su *filesystem*. Un *consumer*, ossia colui che consuma i messaggi nel *topic*, resta in ascolto da remoto e legge il nuovo log non appena disponibile.

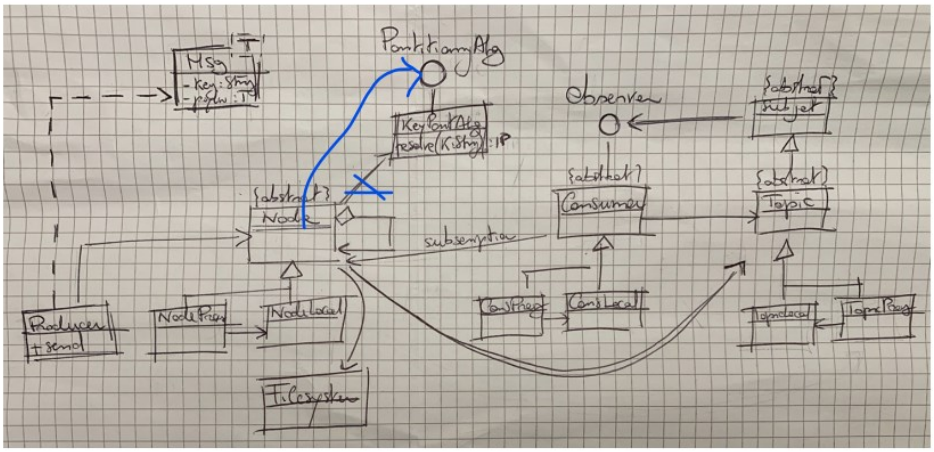
Si modelli tale sistema mediante un diagramma delle classi, comprensivo dei *design pattern* a esso pertinenti.

Il prof dice che era molto difficile; per questo esamina direttamente la soluzione.
Si consideri che (si devono spezzare le singole cose, qui le scrivo *sequenzialmente*):

- Si ha una classe Msg di tipo T
- Si ha una classe Topic
- Si ha una classe Node
- Si ha un collegamento tra classe Node e Topic
- Deve esserci un modo per cui un Producer per inviare messaggi ad un nodo; creo il Producer e genera i messaggi (dipendenza)
- Un producer ha bisogno di un Proxy per inviare i messaggi (il Proxy va dalla parte di chi riceve la comunicazione
- Ogni nodo ha una lista a runtime di nodi Proxy
- Si ha uno Strategy e una sua implementazione che prende una chiave e partiziona le cose
- I consumer restano in ascolto sui Topic; quindi c'è un Observer e il Subject è un Topic (l'Observer è il Consumer)

Soluzione

La soluzione prevede un uso estensivo del pattern Proxy. Inoltre, vengono usati il pattern Observer ed il pattern Strategy.



Usare i pattern aggiunge complessità; devo chiedermi se il problema che ho è simile a quello del pattern, in quel caso ha senso implementarlo. In altri casi, sta solo complicando la situazione.

Pronunce e parole corrette

Si noti che:

- Le parole con gli accenti vanno bene, non quelle con gli apostrofi
    - Esempio: Liberta' non è corretto per Tullio, ma Libertà sì
- Si deve dire repòsitory e non reposìtory
- Le date vanno scritte nel formato YYYY-MM-DD