

Esame finale scritto + progetto

Parte obbligatoria esame scritto e progetto, mentre l'esame orale è un esame opzionale

Esame scritto se in presenza presenta un esercizio scritto ed un paio di domande aperte

Da remoto invece abbiamo un esercizio di modellazione ed un paio di domande aperte che vengono chieste in modo orale

## Software projects fail

Problema che si è ingigantito all'aumentare della loro complessità

Con complessità intendiamo

Standish Caos report: studio creato in maniera sistematica in cui viene analizzato lo stato di alcuni progetti software. Dal 1994 al 2015 si è notato come Challenged (parte software che funziona, ma non nei tempi o i costi prestabiliti) sia la fetta maggiore dei progetti

	Successful	Challenged	Failed
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	34%	17%
Small	62%	16%	11%

## Cercare una soluzione

Lo sviluppo software è una parte importantissima soprattutto per progetti molto grandi in cui con grande afflussi di denaro un eventuale fail del progetto avrebbe ritorsioni abbastanza grandi dal punto di vista monetario e tempistico

Negli anni 69/70 si è iniziato a parlare di software engineering in particolar modo in ambito aeronautico

## Software engineering

è differente in base al tipo di ingegneria che si prende in considerazione.

Una dalla intangibile natura del software e uno alla discreta natura delle operazioni software, cercando di unire il concetto matematico con

**Definizione:** L'applicazione di un sistematico, disciplinato e quantificabile approccio allo sviluppo, all'operatività e alla manutenzione del software

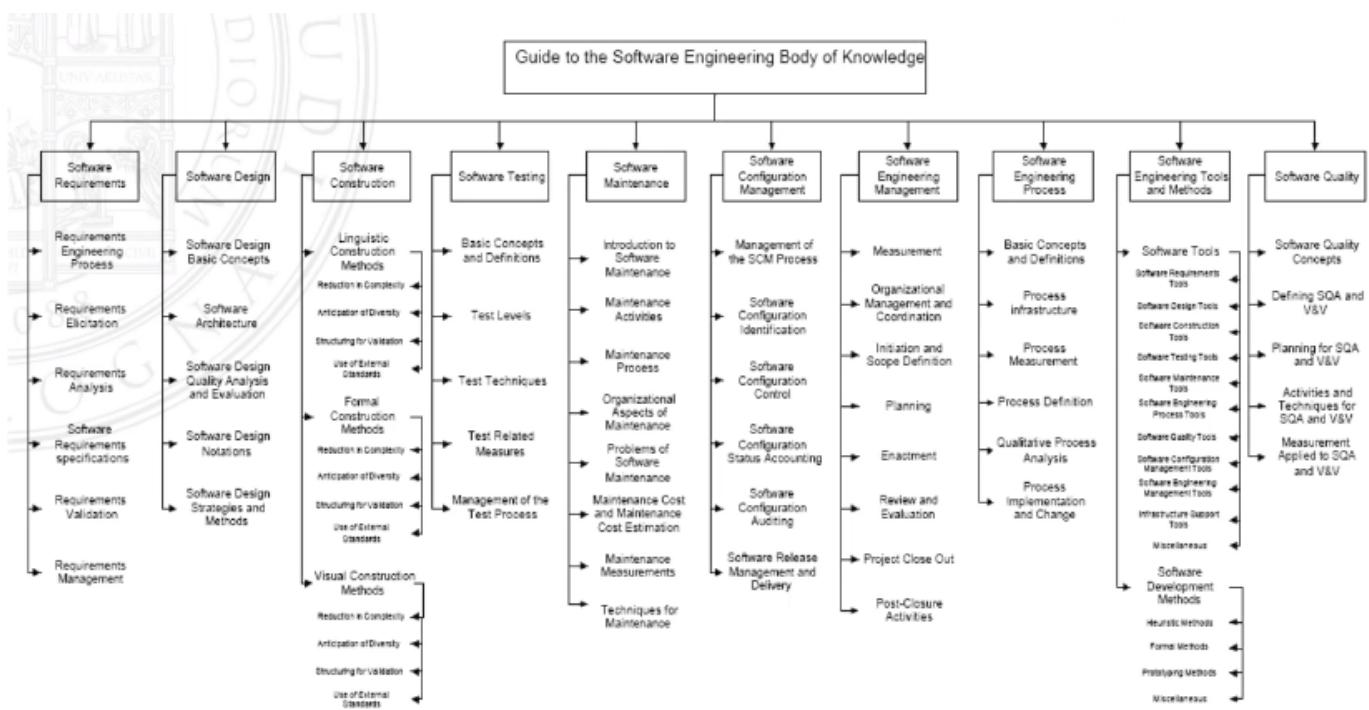
**definizione 2:** L'uso di principi ingegneristici rispetto alle risorse economiche disponibili e se lavora sulla macchina

**Definizione 3:** è una forma di ingegneria che applica principi rigorosi della matematica e dell'informatica per avere una soluzione che vada a sfruttare nel miglior modo le risorse che si hanno.

**Definizione 4:** si occupa della costruzione di software multiversione attraverso team di multipersone

Unendo tutte le definizioni abbiamo che l'ingegneria del software si occupa di creare software che funziona su tutte le macchine del cliente in maniera sistematica ed efficiente.

Per fare tutto questo dobbiamo porre enfasi su una serie di attività: analisi, valutazione, progettazione ed evoluzione del software. In aggiunta ci sono problemi in relazione alla gestione e alla qualità, alla creatività e agli standard.



Questo schema ci indica i vari componenti che vengono inglobati all'interno del campo dell'ingegneria del software

## Affrontare la complessità

Gli strumenti per affrontare la complessità sono:

- Astrazioni
- Modelli

## Astrazione

processo attraverso il quale generalizziamo delle idee in modo che siano applicabili a classi di idee. Andiamo quindi a creare soluzioni non specifiche, ma generali per diversi casi. Meccanismo di astrazione utile per definire dei ragionamenti generali che non si applichino ai singoli elementi, ma a classi generali.

Le astrazioni per loro natura sono più semplici dell'elemento concreto stesso essendo più generale dell'elemento concreto, portando il focus dell'attenzione e del ragionamento sugli elementi più semplici dell'elemento concreto.

Più il progetto diventa complesso, più dobbiamo alzare l'astrazione per renderlo semplice da capire

## Modello

Per affrontare la complessità dei sistemi attraverso l'astrazione dobbiamo usare i modelli. E' una rappresentazione di un sistema che è in grado di fornire delle risposte allineate a quelle del sistema per rispondere ad un set di domande.

**Il modello è un'astrazione o una rappresentazione della realtà in maniera più semplice rispetto a quella che è effettivamente.**

A seconda delle domande che ci poniamo possiamo avere modelli diversi che vanno a rappresentare quella che è la realtà.

Esistono tanti momenti diversi all'interno dello sviluppo di un software in cui possiamo usare dei modelli.

# Modello di analisi

## Astrazione

Tutto il processo di sviluppo risulta essere un processo in cui andiamo ad aumentare il livello astrazione di esso.

Il processo del software può essere visto come una sequenza di attività dal più alto al più basso in grado di astrazione

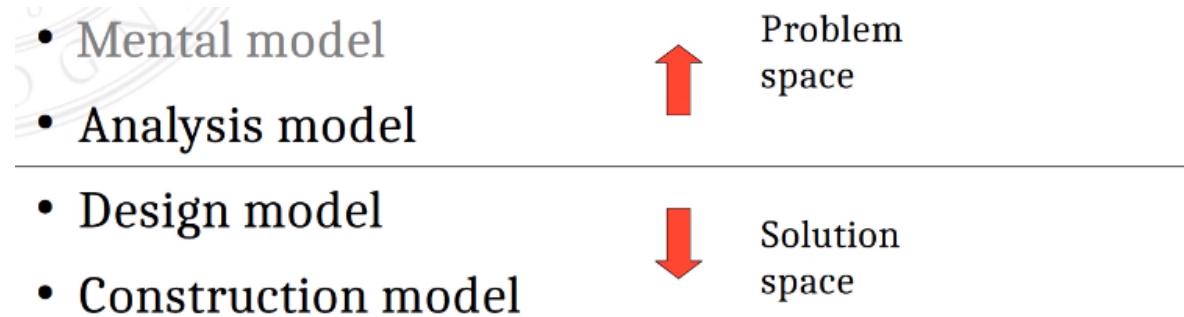
- Analisi
- Design
- Costruzione

## Modelli

Il processo software può essere visto come un raffinamento dei modelli:

- Modello mentale
- Modello di analisi
- Modello di design
- Modello di costruzione

Si parte dalla parte più semplice del modello è ciò che abbiamo noi in testa a modelli sempre più dettagliate e concrete che è la scrittura effettiva del codice



Se operiamo nello spazio del problema allora operiamo a modelli su cosa deve fare il sistema, mentre nello spazio delle soluzioni operiamo a modelli su come deve fare il sistema.

I modelli possono avere forme e strutture diverse. Possiamo anche arrivare ad avere come modelli dei semplici fogli di testo

## Obiettivi del modello di analisi

Devo capire in maniera precisa ciò che è richiesto dal software, devo comunicare lo sviluppo al team di sviluppo e agli azionisti, definendo un set di requisiti che possono essere validati dal software una volta costruito.

Modelli possono avere caratteristiche diverse a seconda di ciò che devono considerare

## Analisi dei modelli artefatti

Soltamente parliamo di insieme di modelli come modelli in generale.

Insieme di modelli che concorrono per fornire delle informazioni (modelli artefatti) che a loro volta sono modelli.

Esempi di **modelli artefatti**:

- documenti necessari
- glossario
- modello dominante
- Specifiche supplementari

## Requisiti

I requisiti dei software sono le restrizioni e ciò di cui ha bisogno un prodotto software

Le aree di conoscenza dei requisiti software:

- **Elicitazione**: estrarre informazioni dal modello mentale
- **Analisi**
- **Specifiche**: se vogliamo definire nello specifico i requisiti
- **Validazione**: dobbiamo definire strategie, una volta che è stato concluso, se è coerente con i requisiti iniziali

Dobbiamo anche suddividere i requisiti principali, in:

- **funzionali**: servono a rappresentare le interazioni osservabili tra il sistema e l'ambiente (es. utente preme il pulsante, sistema deve fare una cosa specifica)
  - **che cosa ci aspettiamo che il sistema faccia**
- **non funzionali**: riguardano quelle proprietà percepibili dall'utente, ma che non sono direttamente collegate agli aspetti funzionali (architettura dei sistemi)
  - **come il sistema ci risponde all'interazione dell'utente**

Ci sono anche ulteriori suddivisioni più specifiche (FURPS +)

- funzionali
- usabilità
- interfaccia
- affidabilità
- performance
- supportabilità
- +

Soltanmente per evitare problemi bisogna definire dei requisiti e dei testi validativi per far sì che non ci siano problemi nella definizione di progetti che vengono definiti a monte in maniera sbagliata.

## Linguaggi di modellazione

Sono dei modelli che vengono espressi con un linguaggio

Sono un linguaggio **Object Orientation**: paradigma in cui spostiamo l'attenzione dagli algoritmi ad una vista del sistema software composto ad entità autonome che collaborano tra di loro caratterizzate dall'avere uno stato (informazioni delle variabili di istanza) ed un comportamento.

Con object orientation parliamo di:

- **Astrazione**: dobbiamo focalizzarci sulle caratteristiche rilevanti di un elemento
- **Incapsulamento**: i dettagli che l'astrazione ci dice di non vedere, noi non dobbiamo vederli
- **Ereditarietà**: quanto lo stato, anche il comportamento può essere specializzato
- **Polimorfismo**: comportamento dipende dal singolo oggetto dal quale andiamo ad operare

## Ereditarietà

il meccanismo di funzionamento può dipendere dagli oggetti o dalle classi (si vanno ad estendere le classi)

Nei linguaggi tipicamente class-based il meccanismo di ereditarietà dipende dal meccanismo dei tipi

In alcuni modelli ad oggetto possiamo vedere che una *sottoclasse* può sovrascrivere alcune parti della *superclasse* (classe padre) inserendo anche elementi aggiuntivi

## Polimorfismo

nei linguaggi che supportano il class-based, un'istanza di una *sottoclasse* può essere usata quando viene richiamata una *superclasse* (classe padre)

## UML

UML assume un approccio orientato agli oggetti sia per l'analisi che per il design/progettazione non considerano però come deve farlo.

UML è linguaggio di modellazione grafico software intensive.

E' inoltre semiformale (la sintassi è formale, non la semantica) e serve per documentare artefatti software.

Presenta inoltre regole sintattiche e semantiche che servono per definire come creare diagrammi validi e come crearli in maniera corretta

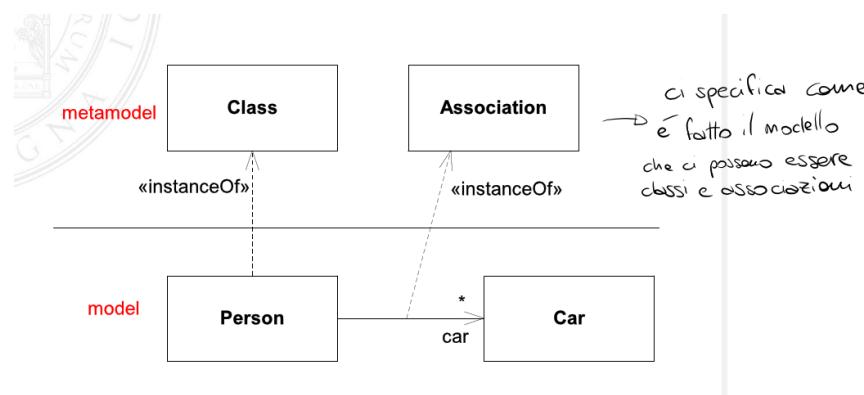
## MOF

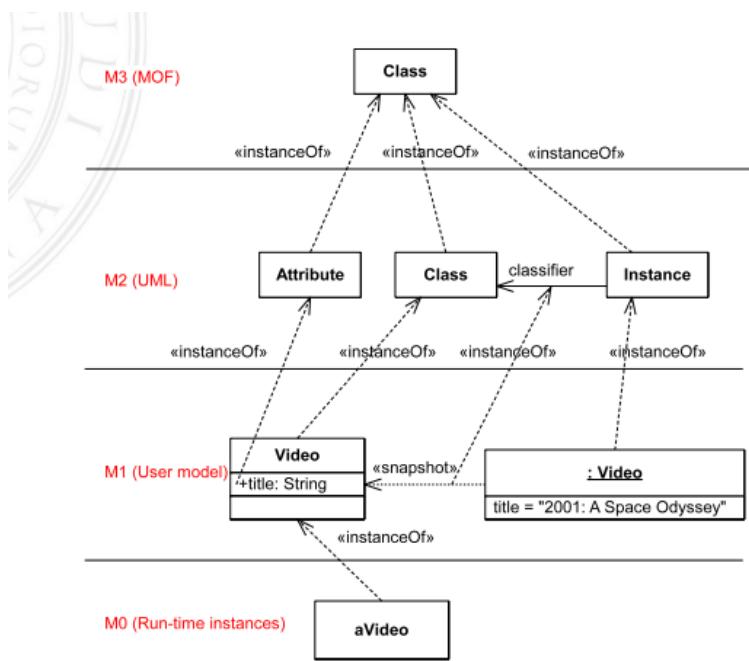
UML è definito come il top di un OMG standards come il MOF (basandosi su di esso).

Il MOF serve a definire un framework concettuale per i linguaggi di modellazione.

Il MOF è strutturato per livelli:

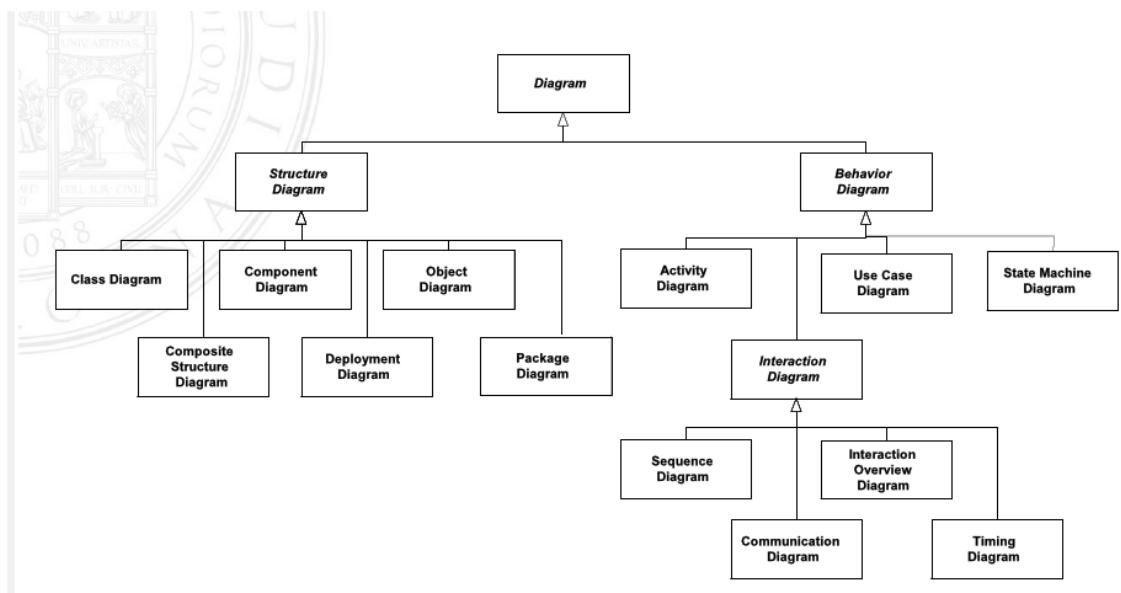
- M0
- M1
- M2
- M3





E' il livello m2 ad indicarci cosa possiamo inserire all'interno del modello

Ciascun diagramma va ad utilizzare il proprio linguaggio



## Strutture dell'entità UML

Per unire numerosi elementi in UML si vanno ad utilizzare dei package

- classi
- Collaboratori: **inserire cosa sono**
- interfacce: **inserire cosa sono**

## Entità comportamentali UML

**Interazione:** unità comportamentale basata sullo scambio di messaggi

**Evoluzione dinamica dei sistemi:** descrive il comportamento del sistema cambierà a seconda dello stato in cui si trova

## Entità di informazione

Ci possono essere delle annotazioni per specificare alcuni dettagli

## Relazioni UML

Le relazioni possono essere correlate da due o più elementi in un modello. Sono rappresentate da linee e possono avere dei nomi.

Sono basate su 4 tipi di relazioni diversi:

- Associazione
- Generalizzazione
- Dipendenze
- Realizzazione

### Associazione

Relazione strutturale tra due modelli di elementi che mostrano gli oggetti di un classificatore connessi che possono navigare verso oggetti di un altro classificatore

### Generalizzazione

Ci indicano che un certo modello è definito sulla base di un altro elemento più generale.

La freccia da una “classe” ad un’altra punta sempre verso la classe più generale.

Questo concetto è strettamente legato al concetto di **ereditarietà**.

### Dipendenza

Collegamento sotto forma di segmento (quando binaria) tra due elementi. Un eventuale cambiamento ad un elemento interno provoca il cambiamento anche dell’altro elemento a cui è collegato

### Realizzazione

una relazione di realizzazione esiste quando esiste tra due elementi quando uno dei due deve essere realizzato o implementato (con uso di interfacce)

## Stereotipi

Uno stereotipo viene renderizzato come un nome chiuso tra << >>.

Possono specificare che determinati elementi (classe) hanno caratteristiche particolari.

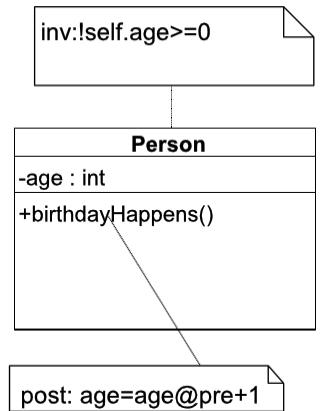
Permettono quindi di estendere il vocabolario di UML in ordine per creare nuovi elementi derivanti da uno già esistente, andando a definire con precisione il nuovo vocabolo.

## OCL

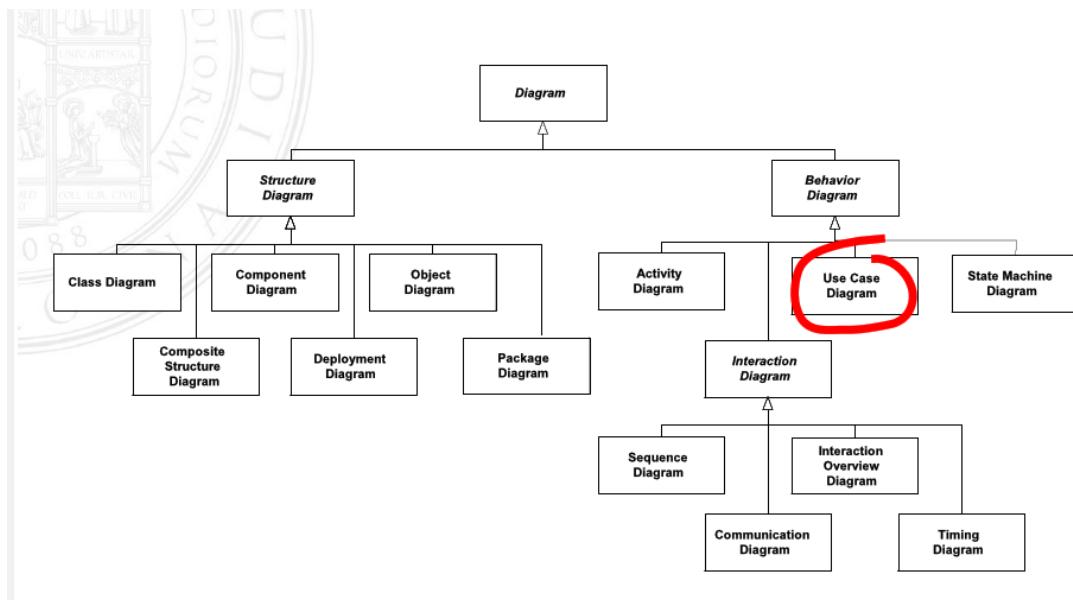
OCL è una specificazione OGM utilizzato per specificare dei vincoli.

Cerca di catturare determinate regole di dominio che solitamente non possiamo prendere con l'UML.

Si legano a degli invarianti, delle precondizioni, postcondizioni ecc..



## Diagramma a casi d'uso

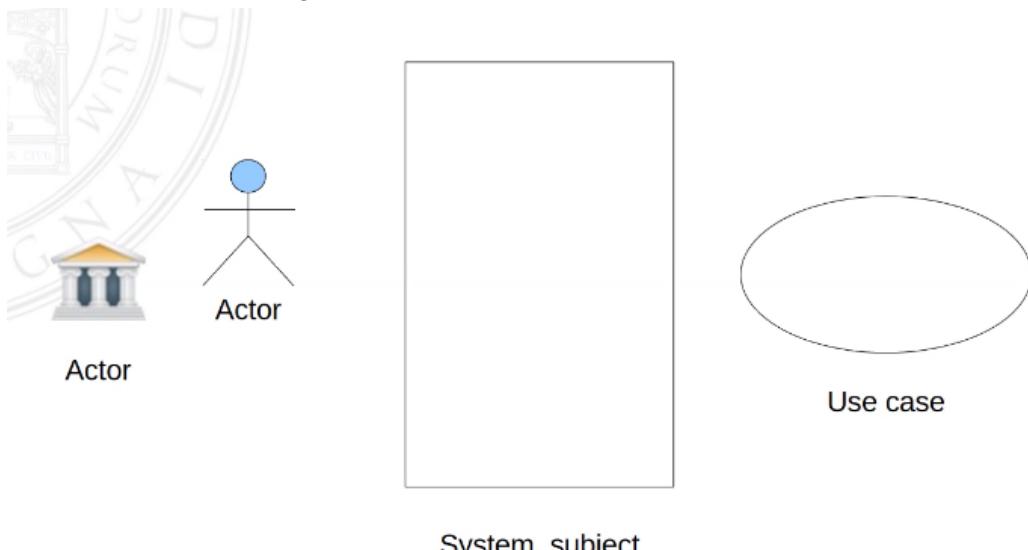


Viene usato per descrivere una serie di attori (soggetto esterno) che un sistema o performance può performare con uno o più soggetti esterni al sistema.

## Uso dei diagrammi a casi d'uso

I diagrammi a casi d'uso vengono usati per specificare requisiti di un soggetto, requisiti sull'uso di un sistema per catturare e capire cosa il sistema deve fare.  
Specifica inoltre le funzionalità offerta dal soggetto (cosa il sistema può fare) e specifica i requisiti che il soggetto considerato pone sull'ambiente, andando a definire come l'ambiente deve interagire con il soggetto

Lo scopo di questo diagramma è quello di catturare dei requisiti



In UML gli attori specificano il ruolo che un'entità esterna ha rispetto al *subject*

Il soggetto rappresenta il sistema che stiamo analizzando o progettando in cui andiamo poi ad applicare i casi d'uso

I casi d'uso sono classifier che specificano sequenze di azioni realizzate dal sistema in funzione dell'interazione con gli attori

## Relazioni tra attori

A seconda del diagramma che utilizziamo possiamo avere diverse relazioni.

Nel nostro caso (UC) abbiamo le generalizzazioni, che ci vanno a definire in modo astratto o concreto specializzandoli usando relazioni generali e le associazioni tra attori e casi d'uso.



In questo caso Customer è l'attore tra il sistema ed il caso d'uso

Con più attori associati ad un caso d'uso ogni attore ha il suo ruolo e ci sarà sempre **l'attore iniziatore** che inizia per prima la "comunicazione".

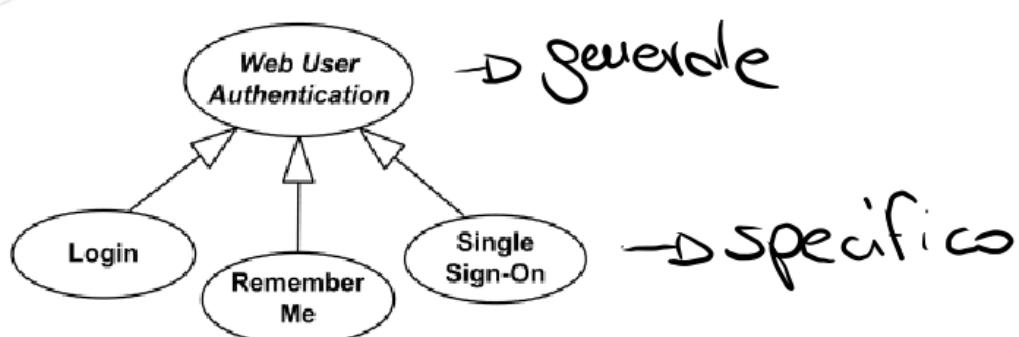
Non sempre si ha la certezza di chi è l'iniziatore.

Quando si sa è buona norma posizionare l'iniziatore a sinistra, anche se non obbligatorio

## Relazione tra casi d'uso

possiamo avere relazione tramite **generalizzazione** ed **estensione**.

In caso di generalizzazione il caso d'uso può essere correlato quando quello più specializzato raggiunge gli stessi obiettivi di quello più generale in maniera differente

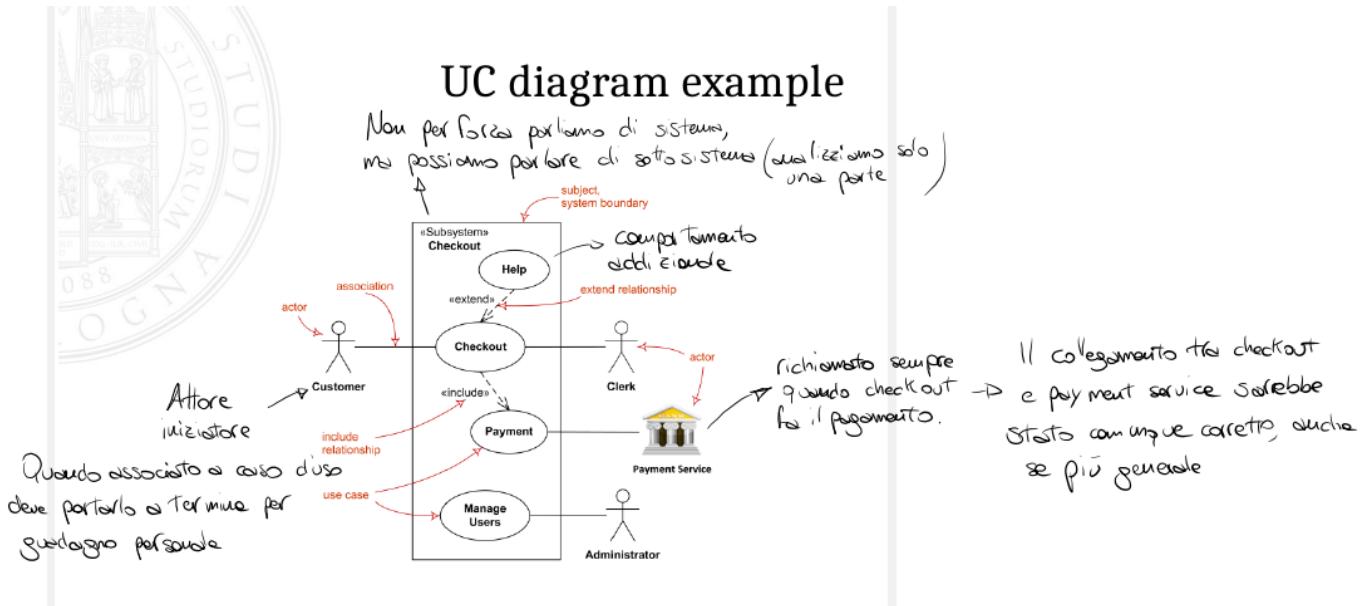
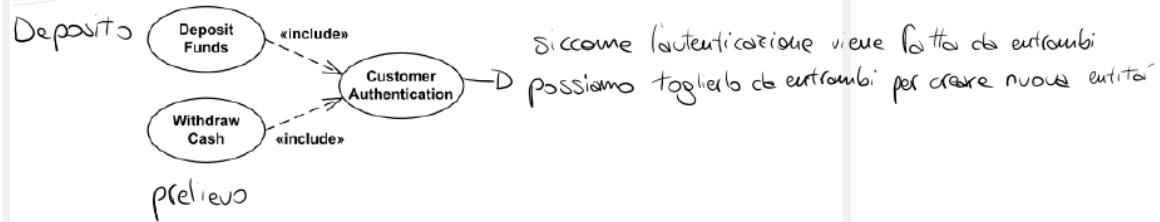


Può essere fatto anche tramite estensione e serve per specificare un comportamento addizionale ed opzionale quando avviene un preciso caso d'uso

Caso addizionale che può o meno aver luogo quando avviene il caso d'uso principale.

Un caso d'uso può avere un'estensione se e solo se il caso d'uso è stato progettato per essere esteso

Nel caso di **inclusione** il comportamento addizionale è obbligatorio, che ha sempre luogo nel caso d'uso principale andando poi a scomporre il caso d'uso addizionale



Quando è associato ad un caso d'uso vuol dire che partecipa ad un caso d'uso.

Il caso d'uso è il perché l'iniziatore vuole avviare ciò. Vuole guadagnare, dal punto di vista personale, in una particolare azione.

con **Include** andiamo ad indicare quei casi d'uso obbligatori che dobbiamo fare addizionale a quello standard.

**Estensione** invece serve per rappresentare un caso addizionale, ma che non è obbligatoria. Quando ci sono più attori quello iniziatore lo mettiamo sempre a sinistra.

Nel caso di **Manage User** abbiamo che per forza di cose **Administrator** è iniziatore in quanto è l'unico che può invocare quel particolare caso d'uso.

All'interno di un caso d'uso, non possiamo inglobare altri casi d'uso, se ciò avviene vuol dire che abbiamo progettato male il caso d'uso

## System Use Case vs Business use case

Possiamo rappresentare un sistema a casi d'uso anche per la rappresentazione di Business. Esempio: con ciò possiamo modellare il caso in cui un cliente entra in banca.

Nei business use case possiamo trovare gli attori che possono essere anche fuori il sistema da noi progettato

## Estensione del sistema a casi d'uso

Un caso d'uso non deve essere troppo complesso.

E' la più piccola unità di un'attività che provvede a dare un risultato soddisfacente all'utente. Descrivere un caso d'uso che può avere tempi lunghi è improprio, stiamo cercando di mettere insieme troppa roba, dovendo quindi scomporre in sotto obiettivi andando poi a modellarne uno come caso d'uso.

Soltamente dovrebbe durare al massimo qualche minuto, non di più.

Dobbiamo pensare obiettivo non come funzionalità generica, ma come obiettivo del singolo attore. (es. sito di aste online, non dobbiamo immaginare di modellare l'acquisto di un prodotto, perché è un lasso di tempo troppo alto, andiamo quindi a modellare semplicemente il caso d'uso in cui uno mette all'asta, uno che punta ecc..)

Questi casi d'uso concorrono per arrivare all'obiettivo finale della transazione del bene

## Linee guida di un sistema a casi d'uso

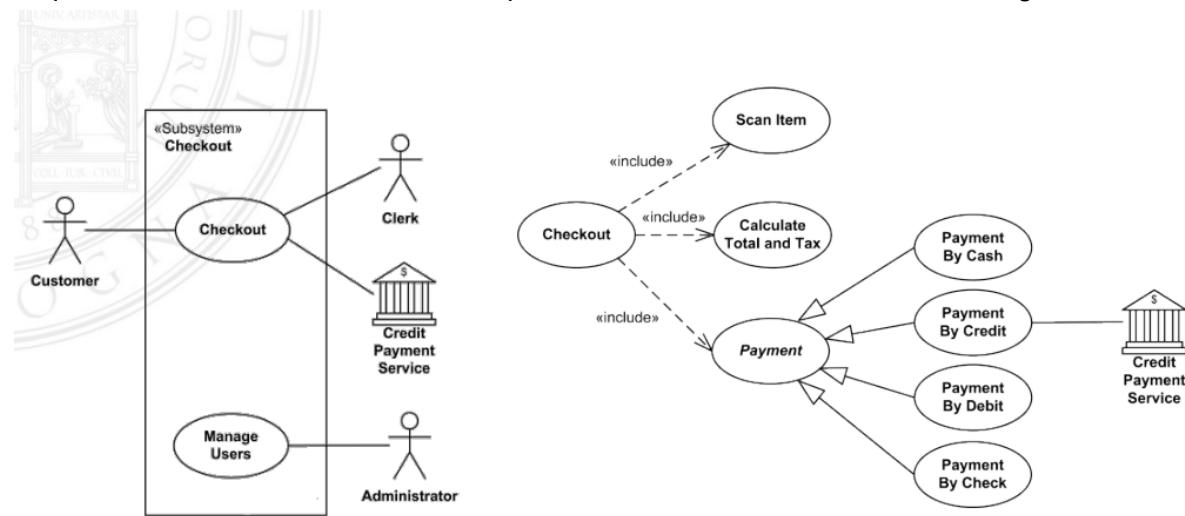
Non dobbiamo utilizzare la decomposizione( da caso d'uso unico ad più di uno di cui uno principale) per altre ragioni oltre il riuso, perché con sistemi d'uso che presentano tanti casi d'uso l'aggiunta di un ulteriore caso d'uso andremmo ad aumentare la complessità.

In certi casi, però uscire dagli standard può essere vantaggioso,ma solitamente è sempre un peggioramento della complessità

Bisogna avere chiaro che gli attori devono essere esterni al sistema, mai interna al sistema, se ciò accade vuol dire che abbiamo sbagliato qualcosa a livello di progettazione

Anche il tempo può essere considerato come un attore. Es. sistema di backup di una banca. in questo caso possiamo definire il tempo come un attore.

In questi casi viene utilizzata un'icona particolare come la clessidra o un orologio.



E' lo stesso modello che abbiamo inserito prima, che però risulta semplificato perchè non ha i vari include ed execute.

In questo caso la decomposizione non utilizza il riuso.

In un modello dei casi d'uso ci aspettiamo di sapere come l'attore deve interagire con il sistema per attivare una certa funzionalità e cosa risponde il sistema per rispettare ciò che chiede l'utente.

Il diagramma dei casi d'uso concorre per definire il caso d'uso. Il modello deve poi essere integrato con artefatti addizionali, non standardizzati, che ci dicono quando si possono applicare particolari casi d'uso

## Scenario o sequenza

con sequenza si intende una sequenza di interazione tra attori e sistema. Servono a definire le sequenze osservabili tra attori e sistema.

Per rappresentarla possiamo utilizzare un elenco numerato.

Se tutto va bene, parliamo di scenario di successo, ma non necessariamente va tutto bene. Andiamo quindi ad utilizzare scenari alternativi che sono proposti sotto forma di estensione in quanto comportamenti che non avvengono sempre.

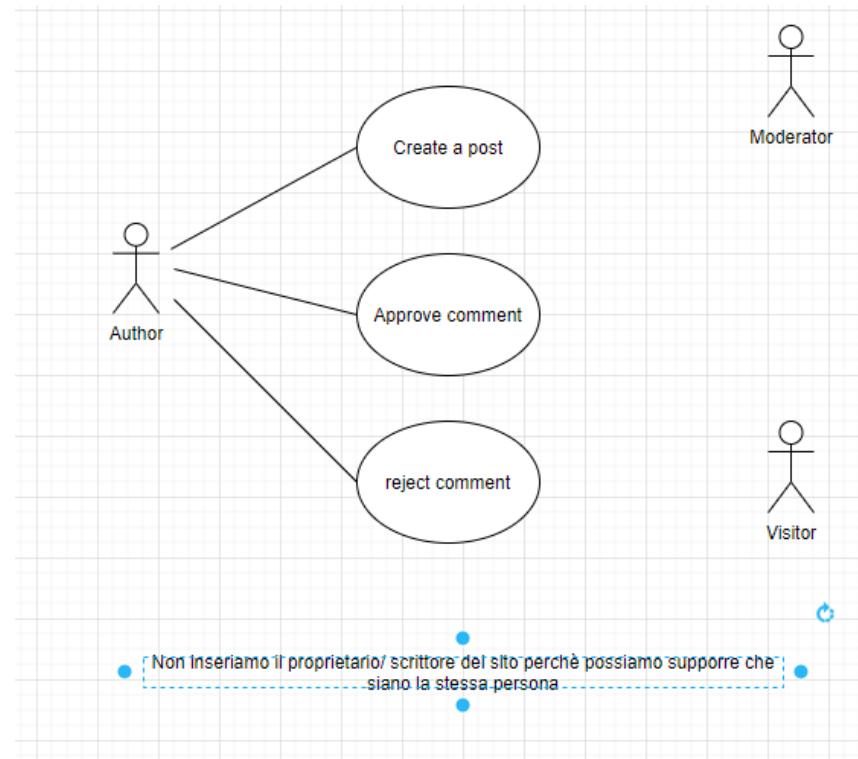
Non per forza devono essere considerati estensioni del modello.

La sequenza può anche essere vista come un'istanza di un caso d'uso e quindi possiamo usare gli scenari. Può essere utile collegare con un preciso scenario perché ci può essere una difficoltà con gli stakeholder.

Gli scenari possono avere anche un valore nel fatto di essere utilizzati come elementi di base per creare dei test.

Molte volte gli artefatti vengono creati e poi raffinati con il tempo andando ad arricchirsi passando per vari step di raffinazione

## Esempio



Modellazione che però non risulta perfetta

## Attività del processo del software

attività che vengono raggruppati in diversi insiemi:

1. Specifica: cosa deve fare il sistema
2. design/progettazione: qual è la struttura che deve avere il sistema
3. implementazione del codice
4. validazione: testing del codice, ma non include solo il testing del codice
5. evoluzione: come si evolve il codice dalla prima release del sistema.

queste attività sono **dipendenze causali** ovvero sono attività che devono seguire un preciso ordine. Prima di fare l'evoluzione, dobbiamo fare la validazione e così via.

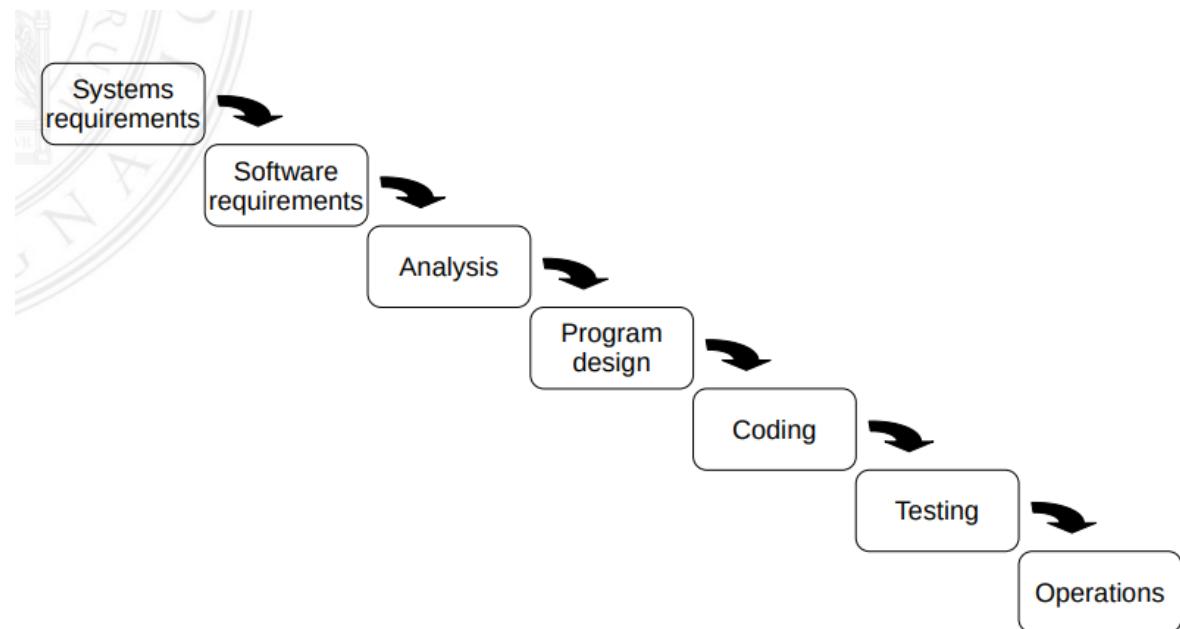
Ci possiamo trovare anche in casi in cui possiamo avere anche fasi in contemporanea

## Prodotto finale

il software è intangibile che ci porta ad una carenza di visibilità (non si sta capendo cosa si sta facendo)

Questo è un problema grosso, e per evitare questi problemi si vanno a creare degli artefatti in cui andiamo a controllare che il programma stia andando avanti nello sviluppo  
per artefatti possiamo intendere documenti di progettazione, piccoli prototipi per confrontarsi con l'utente o per verificare che funzioni ciò che è stato creato, eventuali meeting per far capire a tutti a che punto si è con lo sviluppo

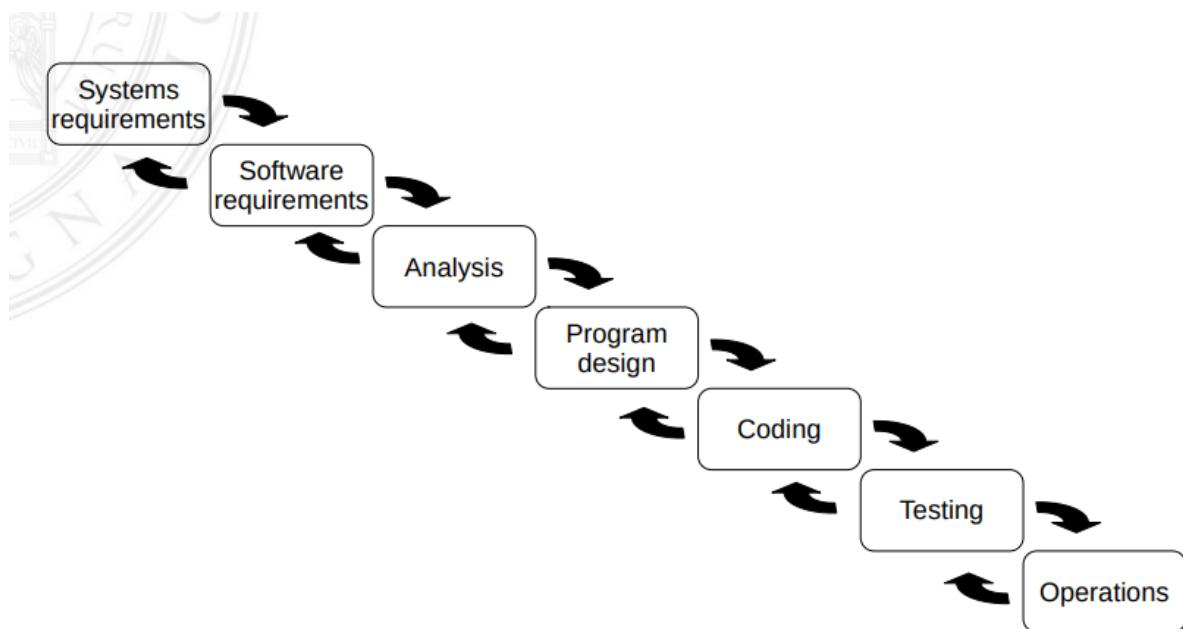
## Modello a cascata



per passare alla fase successiva dobbiamo prima completare la fase precedente in tutte le sue fasi.

Questo modello, però non è del tutto funzionante perché quando arriviamo in fase di operazione molte volte sono sbagliati i requisiti di sistema.

In questo caso bisogna ritornare indietro



Questo quindi inizia ed essere un continuo loop, che nella maggior parte dei casi possono portare alla morte del progetto

Pensando ad un progetto come una piramide, un piccolo errore in fase di analisi, diventa a sua volta una piramide più piccola in cui ereditiamo quelli che sono i problemi dovuti all'errore nella fase

alcuni errori potrebbero anche derivare da un cambio di idee da parte degli stakeholder.  
Nel tempo si sono pensati anche a modelli diversi da quello a cascata, in quanto questo modello risulta essere approssimativo e non utilizzabile.

Pro:

- facile da capire
- fa rispettare le buon maniere
- identifica il prodotto finale
- identifica gli artefatti

Contro:

- non realistico
- ci possono essere dei ritardi nella consegna
- elimina il rischio di management
- difficile coprire eventuali cambiamenti durante lo sviluppo
- può provocare un alto sovraccarico

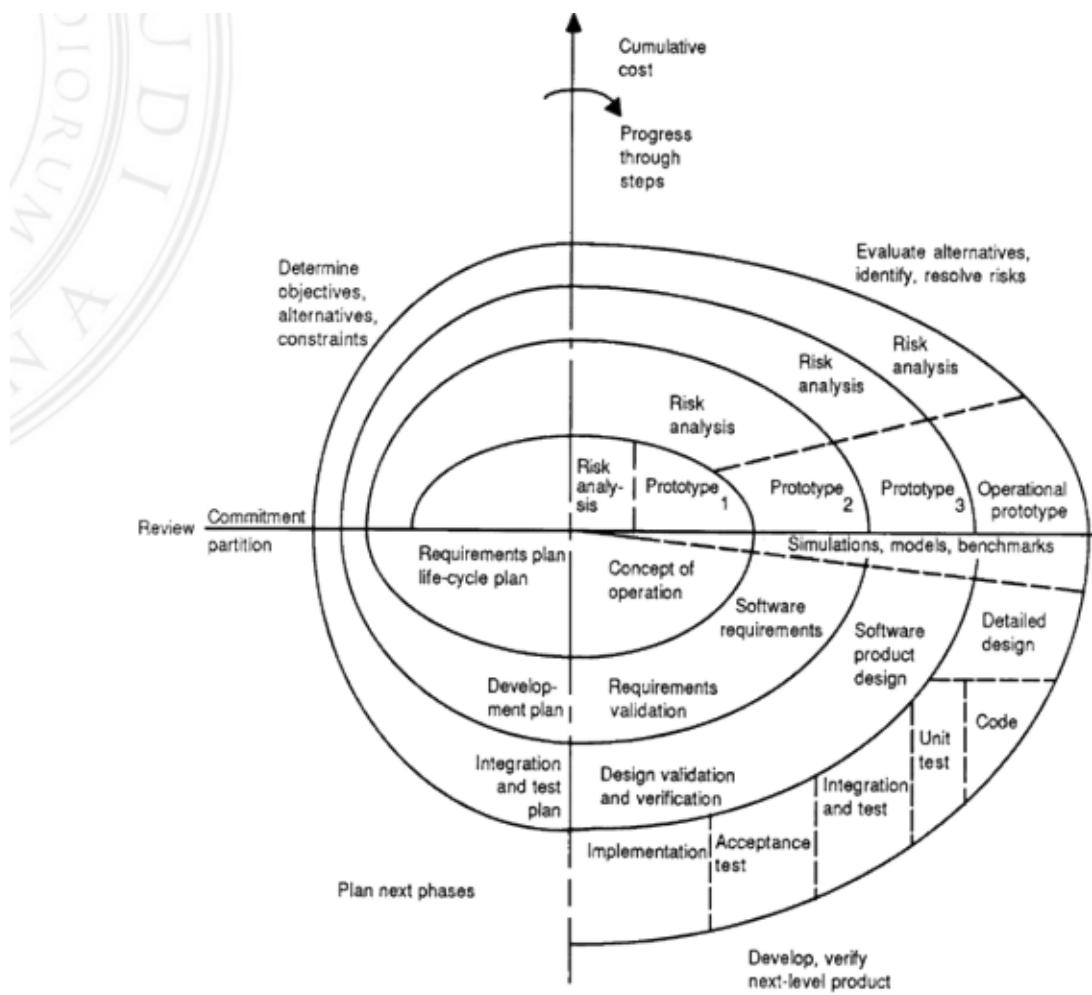
Un altro problema è che noi ci basiamo più sugli artefatti intermedi che sull'artefatto finale

## Modello a spirale

introduce dei concetti che ritroviamo nei modelli più moderni. Pur essendo in circostanze particolari è ormai introvabile come modello da applicare.

è un processo risk-driven e che quindi prende come elemento essenziale il rischio.

Altra caratteristica molto importante è che è un modello iterativo.



Pro:

- introduce della visione interattiva sulla natura dello sviluppo software
- permette un buon livello di visibilità
- introduce il concetto di rischio come elemento guida nella definizione delle attività

Contro:

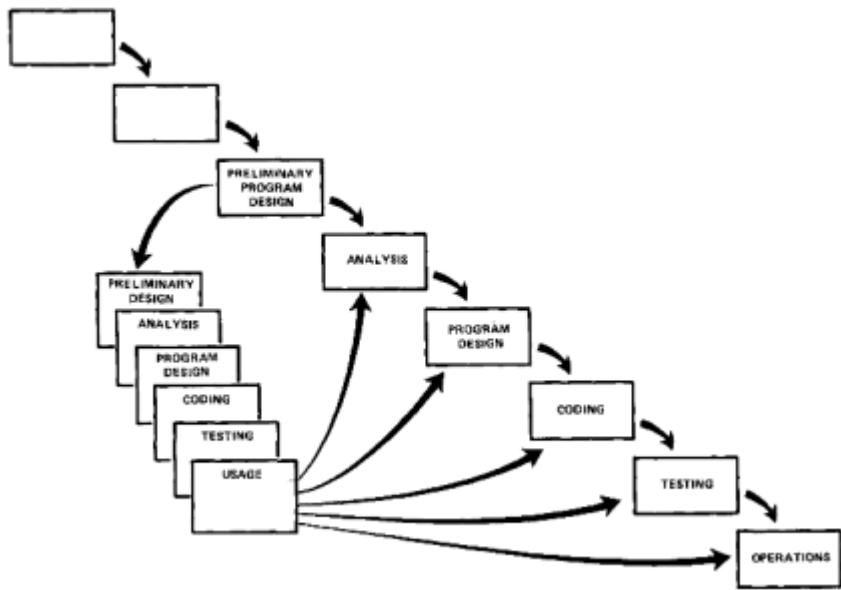
- presenza di modello matematico complesso per valutare il rischio
- difficile realizzazione pratica (ci vogliono precise competenze che in pochi hanno)
- alto sovraccarico

## Iteratività dello sviluppo incrementale

ripetere il ciclo coinvolgendo una prima fase di programmazione e di testing di un sistema parziale.

E' normale iniziare lo sviluppo prima che tutti i dettagli siano definiti nel dettaglio. Sono i feedback ad essere usati per chiarire le specifiche di evoluzione

Degli studi empirici mostrano che questo metodo iterativo è associato ad un alto successo e un alto rapporto di produzione



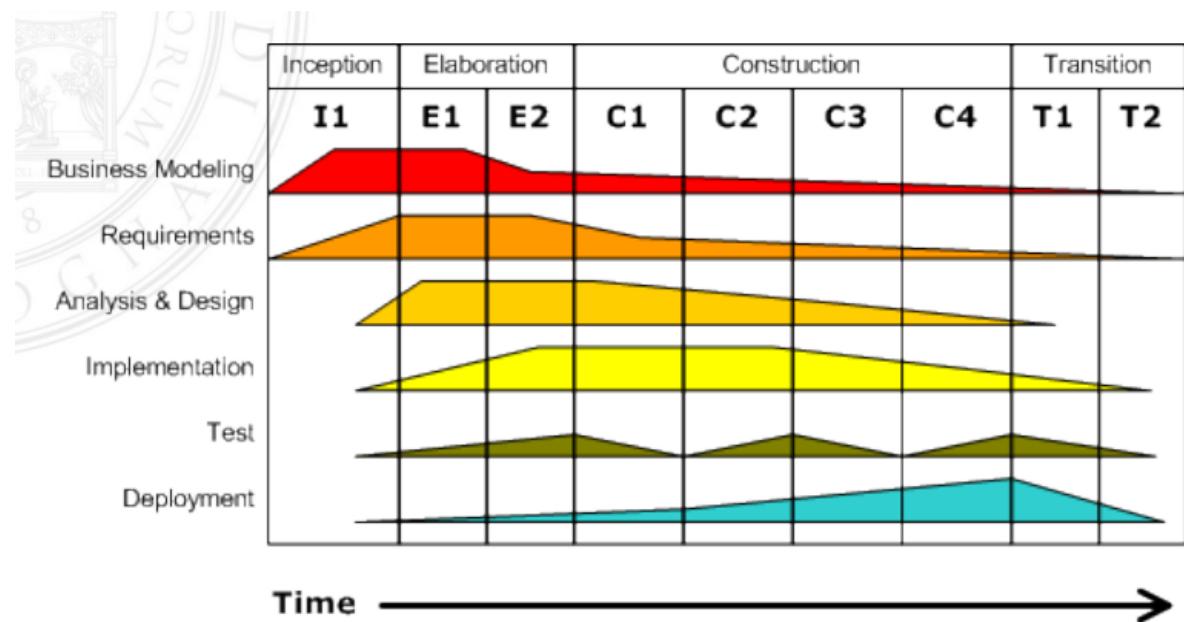
## Processo Unificato (UP)

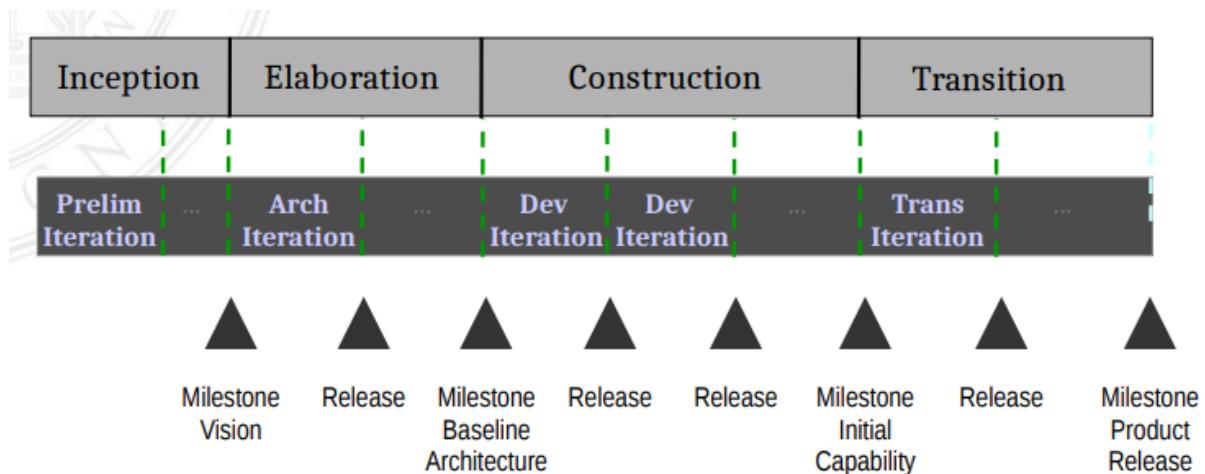
UP è un framework iterativo ed incrementale di sviluppo di processi software.

E' case driven, ovvero utilizza i casi d'uso, è ad architettura centrica e ha il focus sul rischio

L'UP divide il progetto in 4 fasi principali:

1. Inception
2. Elaboration
3. Construction
4. Transition





I Milestone indicano un certo livello di funzionamento e di progresso. Per ogni milestone abbiamo dei concetti e dei criteri da rispettare.

In caso di fallimento che normalmente allunga tempi e spese dobbiamo risolvere questi problemi prima di passare alla fase successiva

## Obiettivi/goals dell'UP

- come l'applicazione può produrre valore (Business Case)
- casi d'uso
- scelte principali che deve avere l'applicazione
- identificazione del rischio
- Concludere in maniera corretta tutti gli obiettivi che ci siamo prefissati all'interno dei Milestone

## UP inception

serve a descrivere il dominio dell'applicazione ad alto livello (gli scope) senza raffinare i casi d'uso. Esistono anche delle alternative principali alla scelta dell'applicazione (candidate architecture),

## UP elaboration

in questa fase si convalida l'architettura, si verifica che lo scheletro del progetto sia implementato e termina con un piano per la fase di costruzione

## UP construction

si implementano le features di sistema e sono azioni time-boxed, cioè ogni sottofase (C1, C2, ...) hanno una durata fissa e stabilità. in caso non vengano rispettate, se ne aggiunge un'altra.

## UP transition

serve a eseguire del training sul output prodotto e la distribuzione, includendo il feedback sul prodotto

UP: le implementazioni più conosciute

- RUP
- AUP
- OpenUP
- OUM

# Capitolo 4: il diagramma delle classi

## Introduzione

È un diagramma utilizzato anche nel dominio della soluzione e per questo è molto più complesso e dettagliato del modello dei casi d'uso che è utilizzato solo per la progettazione. Nella tassonomia dei diagrammi UML, quello delle classi si colloca tra quelli strutturali, a differenza di quello dei casi d'uso che è comportamentale. Quest'ultimo serve a descrivere i comportamenti e le interazioni, mentre quelli strutturali definiscono la struttura (e non il comportamento) degli elementi.

## Struttura

**Definizione di classe:** può essere un elemento del dominio della soluzione o una classe di un linguaggio di programmazione. A seconda del livello di astrazione utilizzato, la classe ha significati diversi e in generale rappresenta un tipo che raggruppa elementi che condividono delle caratteristiche comuni. Una classe è rappresentata come un rettangolo a contorno pieno con il nome della classe in esso ed è potenzialmente diviso in scomparti separati da linee orizzontali. Nello schema base si hanno dei comportamenti che definiscono la classe e quelli più astratti, sono arricchiti da operazioni, come dei metodi, non necessariamente invocabili, ma intesi anche solo come dei messaggi provenienti da altri elementi che attivano dei comportamenti (alto livello di astrazione).

### Sintassi degli attributi e operazioni:

- `[+,-,#,~]/[/<name>[:<type>]]/[<mult>]` il nome è obbligatorio, mentre gli elementi tra parentesi quadre sono opzionali, dove il tipo è preceduto da : e la molteplicità è facoltativa e relativa, per esempio, al nome.
- `[+,-,#,~]/[/<name>|(<params>)[:<type>]]` le operazioni che sono elementi comportamentali, non vengono inseriti nel diagramma, ma vengono descritti con un nome, una lista di parametri tra parentesi tonde, un tipo e un visibility kind.

**Visibility kind** (`[+,-,#,~]`): vengono utilizzati solo nel dominio della progettazione (soluzione) e non nel analisi. Sono identici in tutto e per tutto ai modificatori di visibilità di Java, dove indicano che gli elementi nel modello possono essere visti, acceduti e/o modificati. Questi parametri di visibilità sono:

- + : public, visibile a tutti e si è sempre autorizzati a inviare il messaggio / modificare
- - : private, visibile solo a altre entità dello stesso namespace. Per namespace si intende un identificatore con uno scope e un blocco, cioè una classe con un nome dello stesso blocco.
- # : protected, visibile agli elementi che hanno una relazione di generalizzazione tra di loro. nel caso del subclassing (extended) si può accedere al mainspace diverso, come se fosse quello della classe principale.
- ~ : package, visibile a tutti gli elementi all'interno dello stesso package; all'infuori del package è come se fosse privato. Di default su Java è impostato su package, mappato 1:1.

**Molteplicità:** sono utilizzate per quanto riguarda le relazioni e le proprietà. La sintassi è data da un multiplicity range, con una parte obbligatoria data dall'upper, e un lower opzionale. Se è presente solo il massimo, vuol dire che la molteplicità deve essere esattamente il numero massimo.

<multiplicity-range> ::= [<lower> ..] <upper>

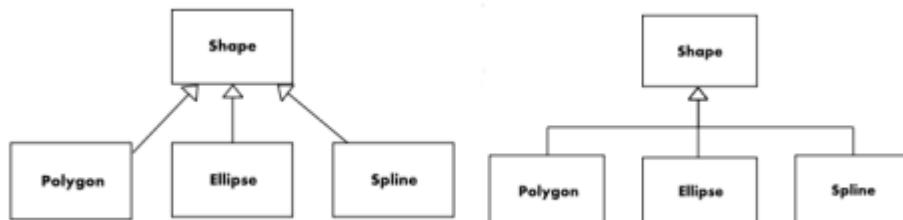
Quando si vuole indicare un massimo infinito, si inserisce l\*, per esempio un range da 5 a infinito si applica: [5..]\*. La molteplicità 0 indica che la proprietà non è presente mentre 15 1..\* indica che la proprietà è ripetuta un numero arbitrario di volte. Inoltre esistono le parole: *order* / *unordered*, *unique* / *non unique* per vincolare che le molteplicità sono diverse o meno e ordinate o meno.

**Istanze ( od oggetti):** sono i singoli elementi la cui struttura si riflette all'interno della classe. Gli oggetti di una classe devono contenere valori per ciascuno attributo che è un membro di quella classe. La notazione delle istanze è con un rettangolo e viene specificato di quale tipo è una istanza. Per esempio in myAddress: Address, la prima parte è il nome dell'istanza e la seconda è il tipo dell'istanza.

myAddress: Address
streetName : String = S. Crown Street
streetNumber : Integer = 381

**Relazioni:** esistono diverse tipologie di relazione tra cui:

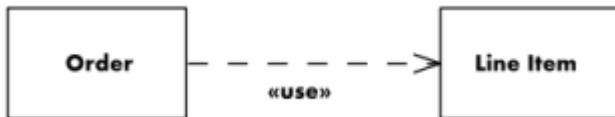
- **Generalizzazione e specializzazione:** sono elementi chiave di MOF e si trovano in tutti i diagrammi UML. La generalizzazione tra classi rappresenta che tra gli elementi c'è una relazione di ereditarietà. La classe più generale è la superclasse (padre o genitrice), mentre la più specifica è la sottoclasse (figlio) ed è quella puntata dalla freccia. Si legge al contrario, partendo dallo specifico e ricavando il generico. La rappresentazione è:



Da un punto di vista semantico si utilizza “è *un tipo di* / è *un*” dallo specifico al generico (il triangolo è un tipo di forma).

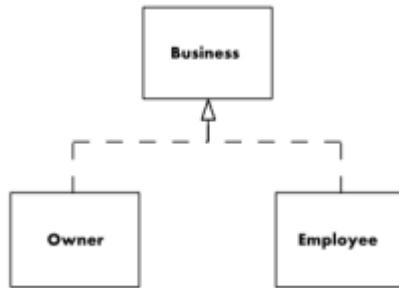
- **Dipendenza:** relazione tra fornitore e utilizzatore, supponendo che una modifica del fornitore ha un impatto sull'utilizzatore. L'elemento dipendente non è in grado di esprimere tutti i comportamenti senza l'ausilio del elemento da cui dipende. Si

rappresenta con:



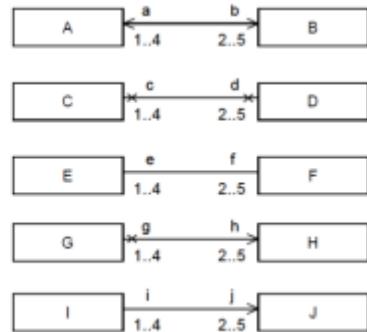
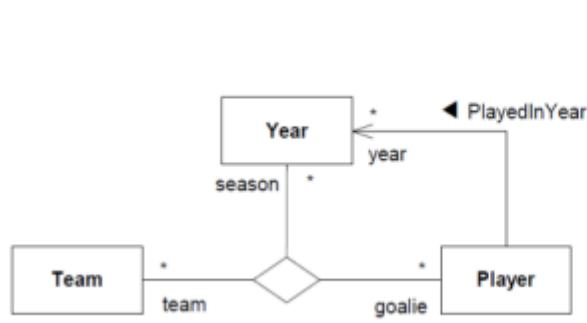
e va dall'elemento dipendente a quello a cui dipende; in questo caso order dipende da line item e senza quest'ultimo, order non funzionerebbe.

- **Realizzazione:** la notazione stessa richiama la dipendenza, essendo a sua volta un tipo di dipendenza. Si implementa con:



dove indica che l'elemento employee realizza business. È una relazione utilizzata raramente e ha senso ad esempio tra un'interfaccia Java e un elemento che la implementa. Può essere utilizzata anche nel dominio del problema, quando il concetto da realizzare non è definito e quello che lo implementa è ben definito.

- **Associazione:** è la relazione più diffusa nel diagramma delle classi e diverge da quella dei casi d'uso per semantica; esiste un collegamento tra le istanze dei tipi associati. Quando si associano delle classi a degli individui, questi ultimi saranno collegati.



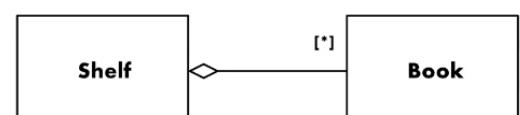
nel caso di due elementi (tupla) indica quale elemento di un tipo è collegato ad un altro elemento di un altro tipo. Se la freccia è presente nel collegamento, indica che è direttamente derivabile da un tipo ad un altro. la diamond shape lega tutti gli elementi associati tra loro, come , in questo caso, un'associazione ternaria.

Usare numerosità significa che dato un elemento elementi di a coincidono da 2 a 5 elementi di b. Dato un elemento di un tipo, l'altro argomento sia ricavabile e che ci possa essere una correlazione tra loro

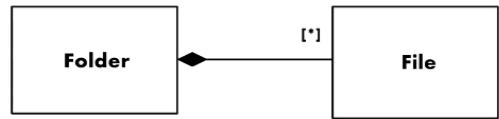
- **Aggregazione:** ci dice che il ruolo che giocano i ruoli associati non è perfettamente associato.

Lo shelf ha il, ruolo di tenere traccia dei libri

L'aggregante solitamente è unico, l'aggregato invece è più di uno.

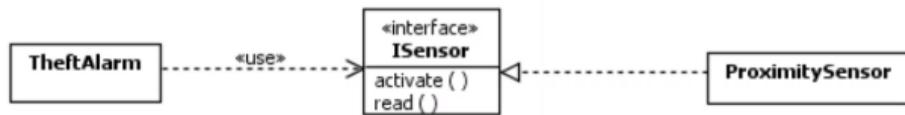


- **Composizione:** assomiglia all'aggregazione, ma viene utilizzata per rappresentare una relazione più forte dell'aggregazione. Viene utilizzata per modellare modelli tutto parte. L'elemento aggregante ha controllo del ciclo di vita dell'elemento aggregato, e l'elemento aggregato non avrebbe senso di esistere senza aggregante.
- Specificano il collegamento tra i due elementi.

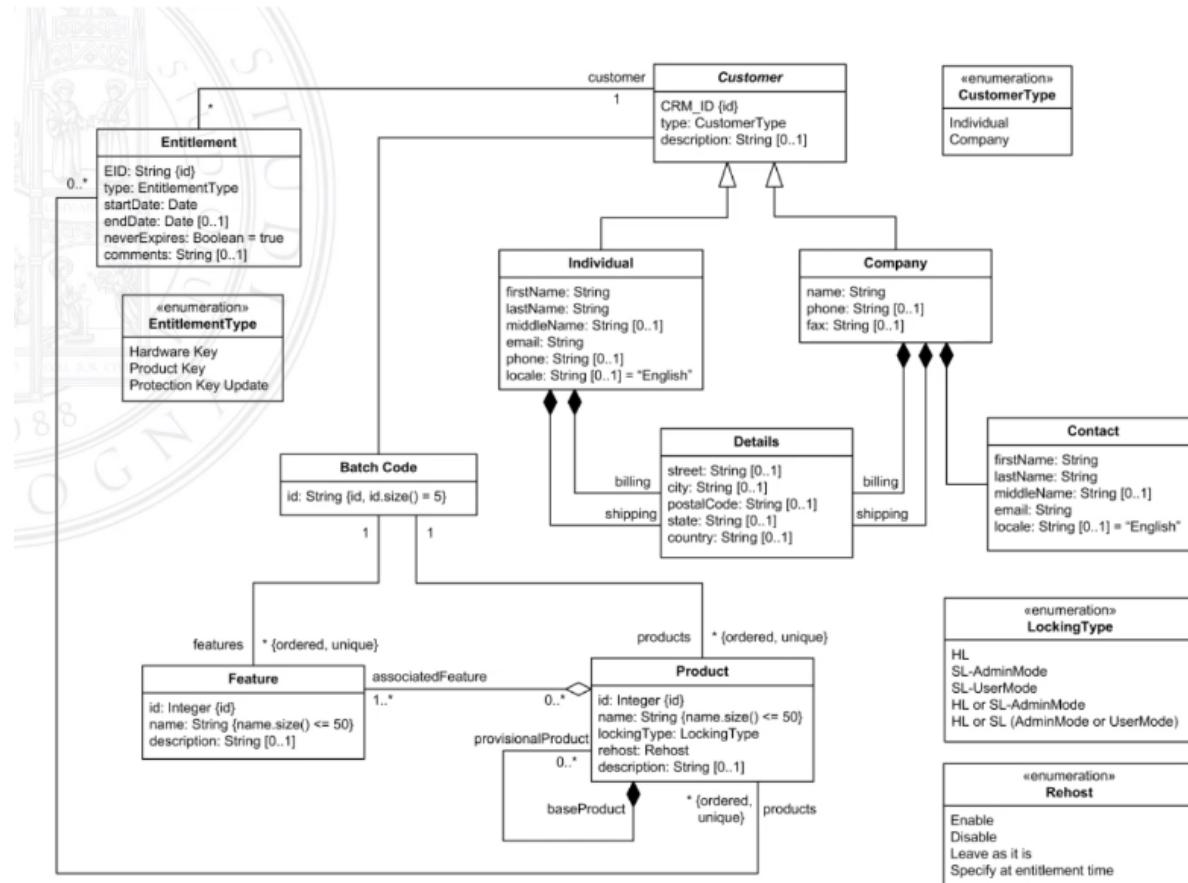


## Classi astratte

Una classe astratta non ha bisogno di istanze dirette. Le sue istanze sono istanze di una sua specializzazione, perché nasce per essere istanziata da classi concrete



Ci dice che ISensor è un'interfaccia (classe astratta)



## Introduzione Domain model

Rappresenta una specie di dizionario visuale.

Ci rappresenta il concetto all'interno del modello di dominio attraverso le features e come solo correlati con gli altri

Gli **artefatti** sono una UML class diagram.

Come prima fase andiamo ad indicare le entità e una volta fatto ciò andiamo a capire le relazioni.

## CRC cards

artefatto che può essere usato per l'analisi dei domini di modello.

Strumento grafico e visuale pensati per essere rappresentazione del dominio sotto forma di pezzetti di carta.

Nei collaboratori andiamo ad indicare le entità con cui ci si relaziona

Class name	
Responsibilities	Collaborators

Cart	
•Knows user •Knows items •Adds items •Removes items	Item

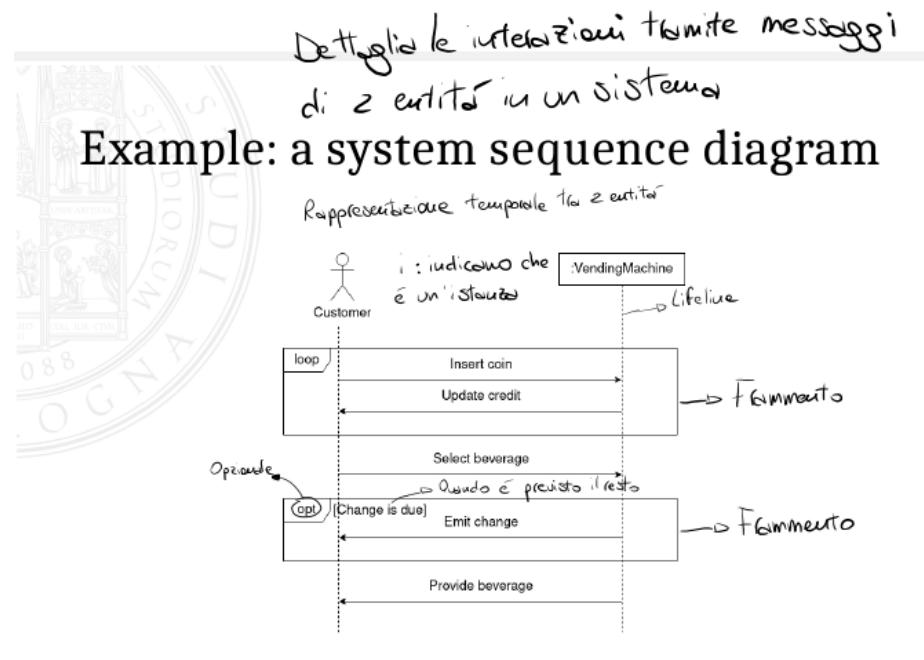
User	
•Knows name •Knows cart •Puts items in cart •Checks out	Cart

Il valore si rileva solo se utilizzo il processo che coinvolge direttamente gli stakeholder

## Diagramma di sequenza

E' il più comune diagramma di interazione e che si concentra sull'evoluzione dinamica sulla sequenza di messaggi scambiati e a cui corrispondono specifiche lifelines.

Diagramma di sequenza può essere usato sul dominio dell'analisi per specificarlo meglio.



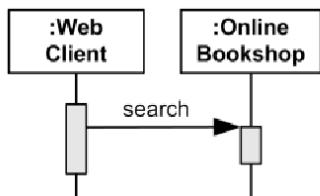
## Lifelines

sono degli elementi che rappresentano un individuo che partecipa all'interazione

## Messaggi

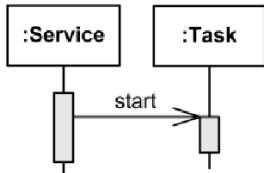
elementi che definiscono uno specifico tipo di comunicazione tra lifeline di un'interazione  
Possiamo avere diversi tipi di messaggi che vanno a cambiare gli action type:

- **chiamate sincrone/asincrone:** rappresentano operazioni di chiamata. Mandano il messaggio e sospendono l'esecuzione mentre attende una risposta. Usano frecce con la punta nera.

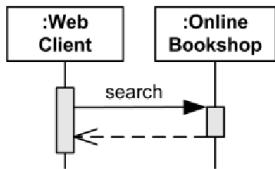


Con le chiamate asincrone, invece, non c'è bisogno di aspettare una risposta e ciò ci

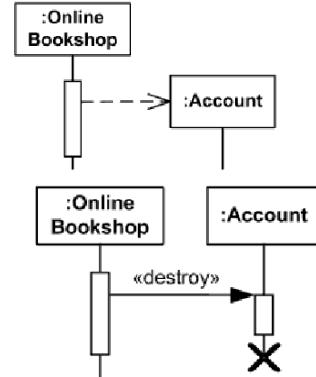
evita di aspettare e di sospendere l'esecuzione.



- **segnale asincrono**
- **risposta**

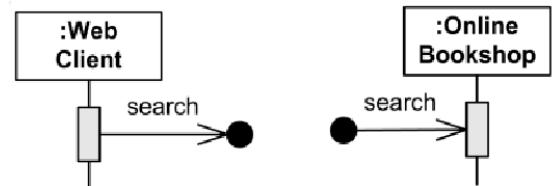


- **creazione:** messaggi speciali per rappresentare sotto forma di messaggio nuove entità
- **eliminazione:** messaggio speciale che viene utilizzato per distruggere un'entità  
Possiamo considerarla come un invito alla distruzione



## Lost & found

escamotage che serve per evitare di appesantire dal punto di vista grafico la rappresentazione



## Interaction fragment

rappresentano la più generale unità dell'interazione.

## Occurrence

è un frammento dell'interazione che rappresenta un momento in un tempo compreso tra l'inizio e la fine di un messaggio o di una esecuzione

## Execution

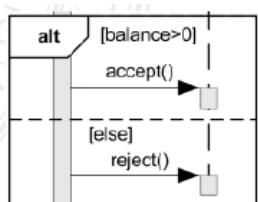
serve a rappresentare che c'è una qualche attività dal partecipante

## Frammenti combinati

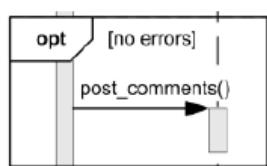
possiamo specificare che ci sono elementi ripetibili o opzionali

Abbiamo diversi tipi di frammenti combinati:

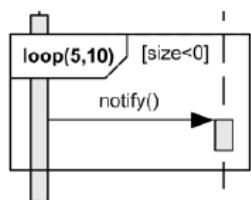
- **alt:**



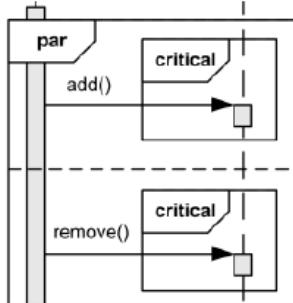
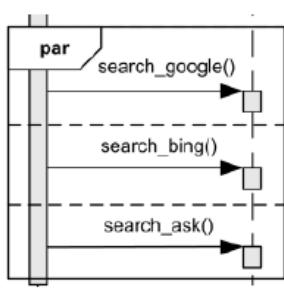
- **opt:**



- **loop:**



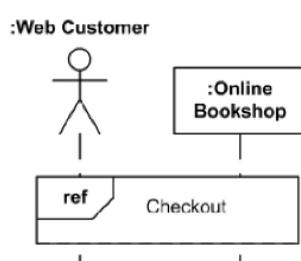
- **break**
- **par:** elementi che hanno esecuzione eseguita in parallelo



- strict
- seq
- ...

## Interaction Use

Interazione che usa un frammento dell'interazione che fa riferimento ad un'altra interazione.  
é comune usarlo per quelle interazioni tra diverse interazioni



## Suggerimenti

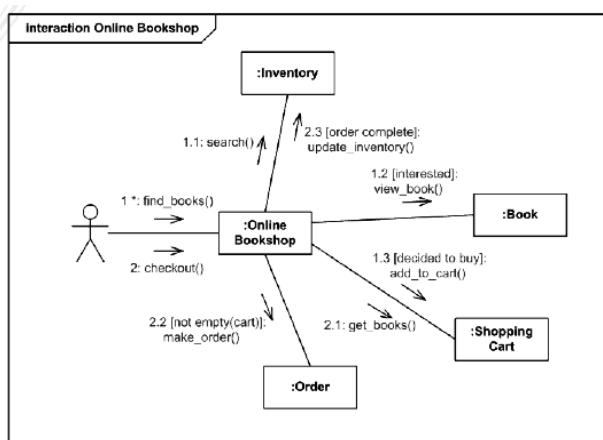
Non dobbiamo generalizzare troppo le sequenze per evitare che diventano troppo complicate da rappresentare.

Dobbiamo usare il diagramma di sequenza quando possiamo usare un caso che ci descrive più di un diagramma.

## Communication Diagram

mostra le interazioni tra due lifeline usando una rappresentazione numerazione.  
Questo avviene perché non andiamo a rappresentare dal punto di vista temporale l'evoluzione della comunicazione tra le due entità.

Possono essere diagrammi più compatti, ma che sono poco specifici in caso di rappresentazioni molto precise.



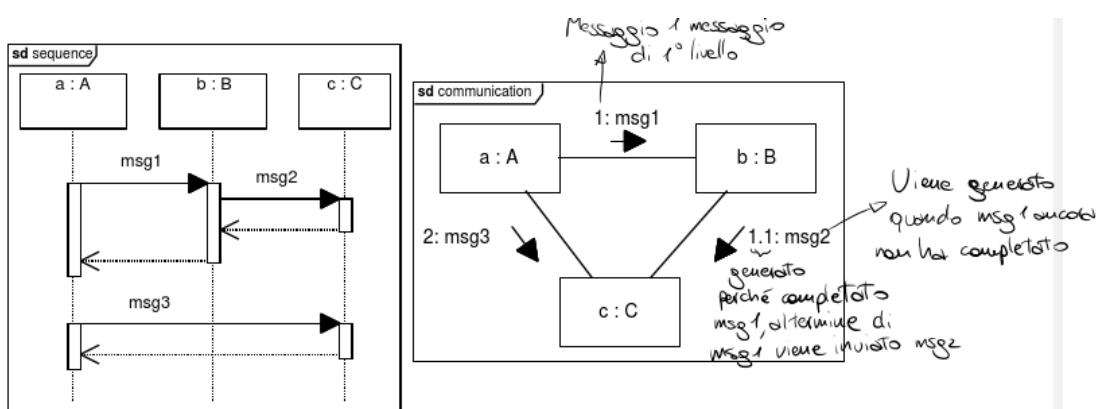
## Message

stessi tipi di messaggi del diagramma di sequenza. Abbiamo però delle **sequence expression** per rappresentare il collegamento di messaggi senza interazione

**Sequence-expression ::=**

sequence-term '.' . . . ':' message-name

**Sequence-term ::= [integer[name]][recurrence]**



## Activity diagram (diagramma delle attività)

diagramma comportamentale in termini di flusso di controllo o flusso dell'oggetto con particolare enfasi sulla sequenza e sulla condizione del flusso

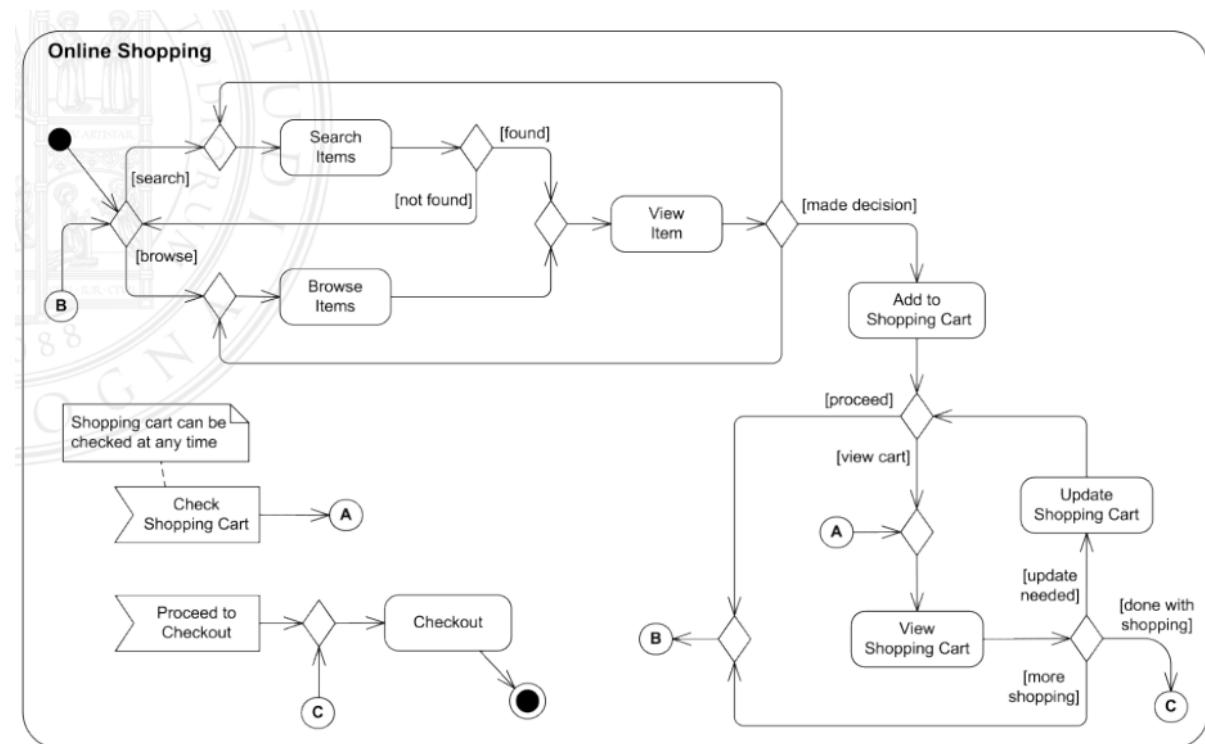
Può essere utilizzato per rappresentare diversi elementi anche di diverso tipo e livello.  
Possiamo modellare classi, casi, interfacce, componenti.

### Semantica

Una caratteristica importante è che non sono solo in grado di definire azioni interne, ma anche azioni concorrenti tra loro, quindi può catturare anche elementi non elementari

### Elementi principali di un Activity Diagrams

- Attività
- Nodi dell'attività
  - Azioni
  - Oggetti
  - Controlli
- Frecce dell'attività



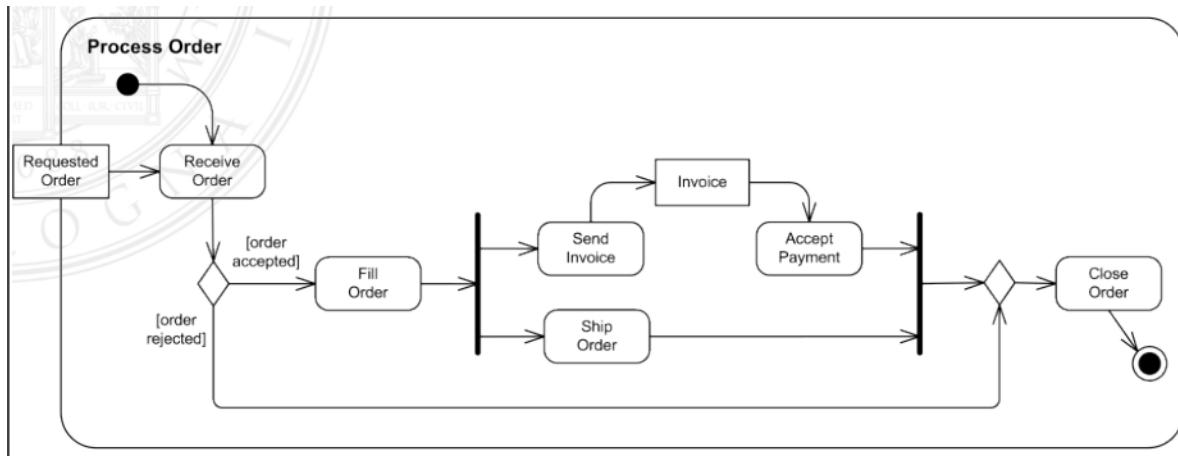
Il nodo nero in alto a sinistra ci indica l'inizio del processo (considerabile anche come flusso di azioni)

il primo nodo di controllo ha più input, ma due output con un predicato tra [] quando si verifica quei particolari casi che l'utente vuole fare interagendo con il nostro sistema.

Selezionando *[browse]* entra nel caso del *Browse item* ed entrando poi dentro *View Item* andando poi ad effettuare una decisione, ovvero entrare nel caso in cui vogliamo effettivamente comprare il prodotto oppure scegliere un altro prodotto.

Il secondo segnalino (il secondo nodo nero) ci indica la fine dell'interazione tra l'utente ed il nostro sistema.

Questo diagramma ci indica l'evoluzione del nostro processo, in cui ciascun passo indica i processi che devono essere eseguiti in ordine.



Questo diagramma, come indicato precedentemente, ci indica anche i vari passi con sistemi concorrenti.

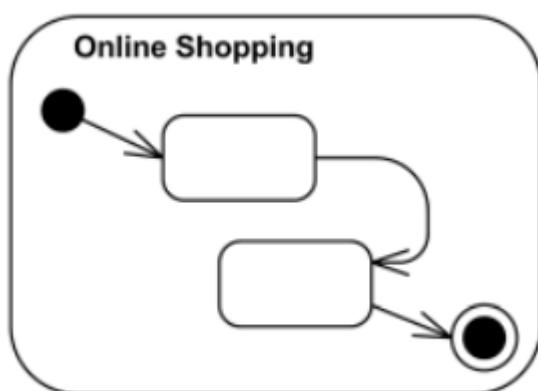
La linea nera verticale ci indica che entrambi i segnali vengono eseguiti contemporaneamente quando entriamo nel caso di *Fill Order*.

Stessa cosa nella seconda linea nera in cui andiamo ad effettuare l'ultimo caso quando entrambi gli elementi precedenti sono stati completati

## Attività

è un valore parametrico rappresentato come coordinate di flusso di un'azione.

L'esecuzione del flusso è modellata come nodi dell'attività collegati tramite frecce dell'attività



## Azioni

Un'azione è un elemento rappresentato da un singolo step atomico (non possiamo scomporre le azioni in altri elementi più piccoli).

Esistono vari tipi di azioni:

- Occorrenze di funzioni primitive
- Comunicazioni di azioni
- Manipolazioni di oggetti
- Invocazioni di alcuni comportamenti

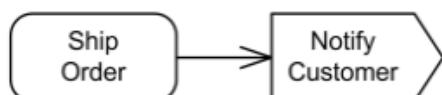
Le azioni sono disegnate come un rettangolo dagli angoli arrotondati.

Il nome o la descrizione di un'azione è interna al rettangolo.

Un'azione può avere alcuni set di input e output frecce di attività che specificano il flusso di controllo ed il flusso di dati tra un nodo e l'altro.

Esistono diversi tipi di segnali:

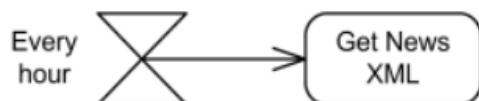
- Send Signal



- Accept Signal



- Repetitive time



## Call behavior action

Una call behavior action rappresenta la chiamata di un'attività e indicata posizionando un simbolo a forchetta all'interno dell'azione

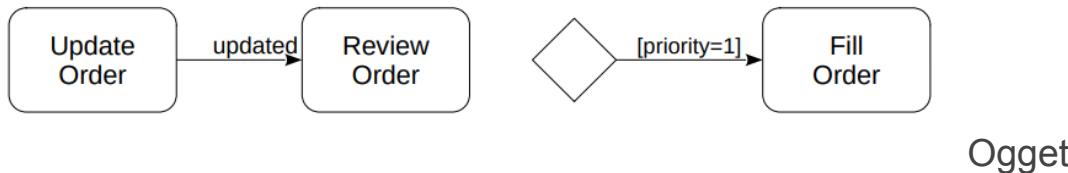


## Archi delle attività

Rappresentano il collegamento tra due nodi dell'attività collegando il flusso tra il nodo di partenza ed il nodo di arrivo.

E' una generalizzazione del flusso di controllo ed il nodo dell'oggetto di flusso.

Possiamo specificare delle guardie, ovvero un predicato (espressione logica associata a vero o falso) in cui andiamo ad effettuare una valutazione in cui se il predicato è vero può passare, altrimenti non può passare attraverso



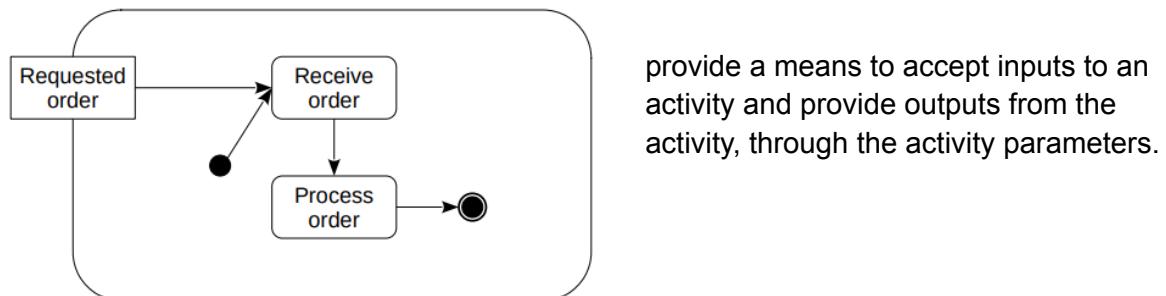
Un nodo di oggetto sono un nodo dell'attività che è parzialmente definita da un flusso dell'oggetto in un'attività.

viene rappresentato da dei rettangoli in cui si va a prendere il token dell'elemento precedente portandolo a quello successivo

Viene indicata da un'istanza da un particolare Classifier.

## Nodo parametrico delle attività

Un nodo parametrico delle attività sono nodi di un oggetto all'inizio e alla fine del flusso.

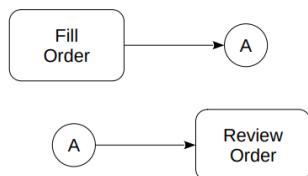


## Input / Output pins

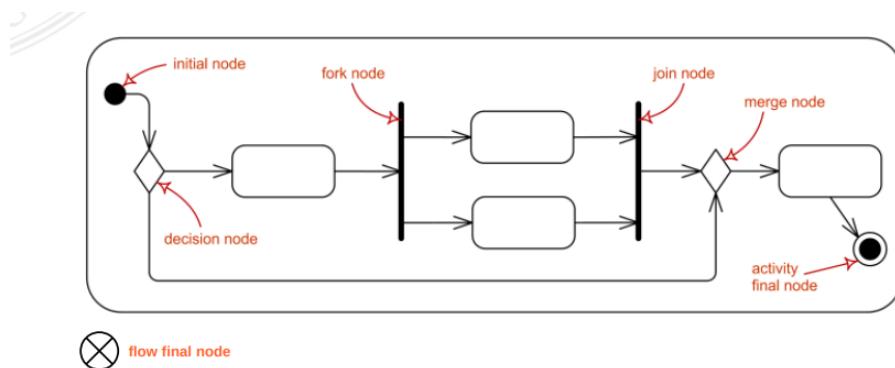
è una variante della precedente rappresentazione dei nodi

## Connettori

Alcune frecce delle attività possono essere collegate usando un connettore, un piccolo cerchio con all'interno un nome per indicarlo



## Controlli



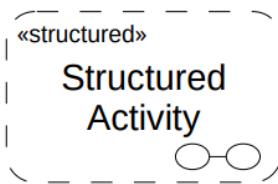
Sono un nodo di attività usando le coordinate di flusso tra altri nodi.

Include *initial node*, *flow final node*, *activity final node*, *decision node*, *merge node*, *fork node*, *join node*

- **Initial node** → nodo iniziale del processo
- **Decision node** → nodo che può avere più archi di entrata e più archi di uscita
- **fork node** → va a collegare un nodo (o più nodi) ad uno o più nodi
- **join node** → quando il token arriva dagli archi di ingresso, devono aspettare che tutti arrivino a tutti gli archi di ingresso. Quando arrivano tutti vengono cancellati tutti e viene creato un singolo token che poi viene fornito sull'arco di uscita. Usato principalmente per la sincronizzazione
- **activity final node** → indica il termine di un flusso all'interno del processo. Non vuol dire che il processo sia terminato
- **flow final node** → quando un segnalino entra dentro, cancella il segnalino che entra e tutti gli altri, anche se attivi, facendo terminare il processo

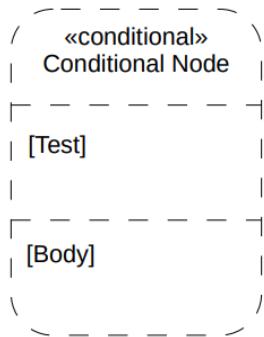
## Attività strutturate

Sono nodi che contengono altri nodi. Un nodo non può essere direttamente contenuto da più di un'attività strutturata.



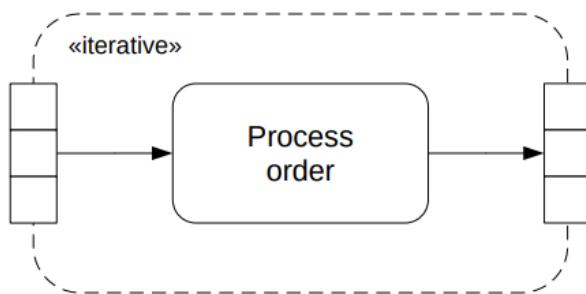
## Nodi condizionali

servono a racchiudere pezzi del diagramma. Possiamo specificare che ad un certo caso possiamo avere una parte del diagramma



## Regione di espansione

è un nodo strutturato che va a lavorare su collezioni di elementi come input, azioni o altri elementi di una collezione individuale, producendo elementi ad una collezione di output. Gli elementi di processo possono essere eseguiti sequenzialmente (iterative), concorrenzialmente (parallel) o in streamline (stream → usata quando abbiamo un continuo arricchimento di input. A differenza dei precedenti si rimane in attesa per sempre fino a che qualcuno non blocca il processo)

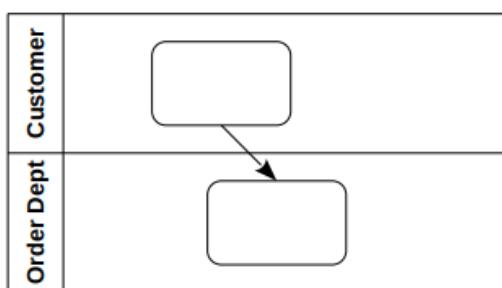


## Activity partition (line)

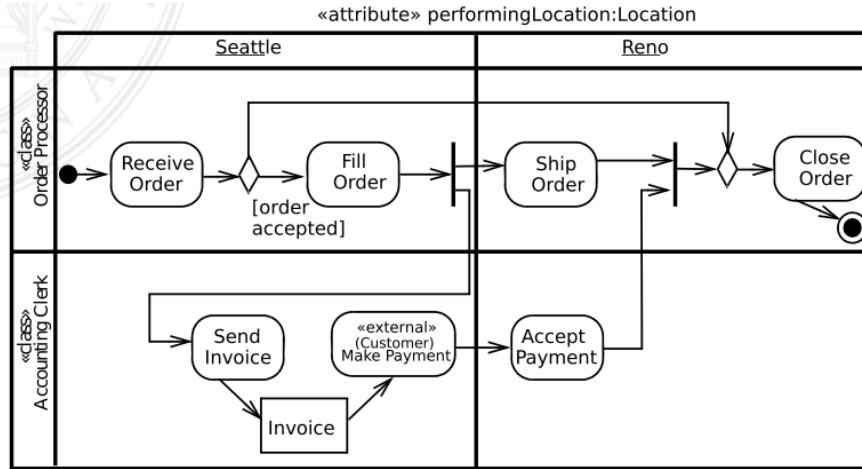
Sono gruppi di attività usati per definire chi o cosa all'interno del sistema deve prendersi carico di realizzare dell'attività che stiamo svolgendo.

Notazione che partiziona il sistema andando ad utilizzare dei rettangoli. Si espande in maniera orizzontale o verticale.

Caratterizzano gli elementi che fanno cosa.



Solitamente queste partizioni vengono effettuate da *Classifier, Instance Specification, Property*

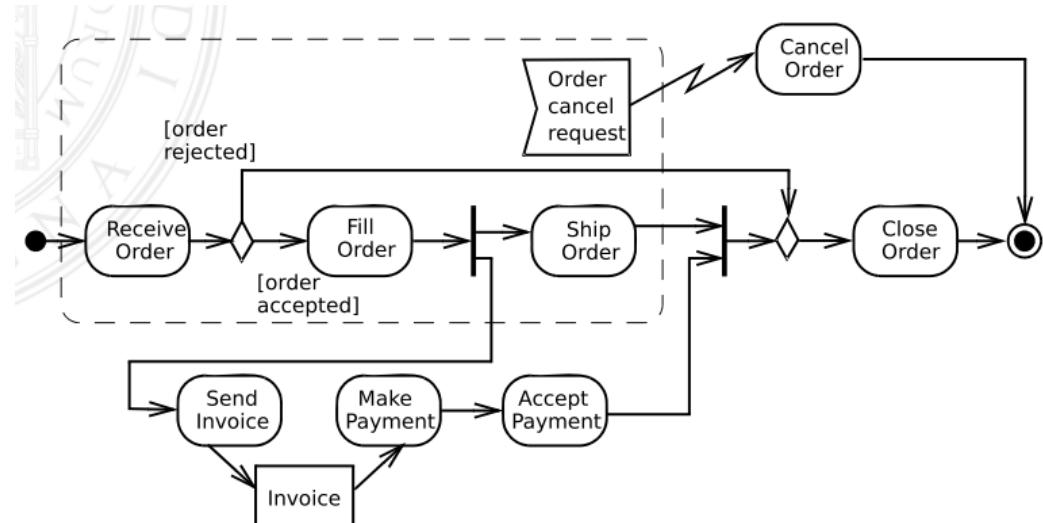


### Alternative notations

Possiamo anche non utilizzare le linee andando ad utilizzare un rettangolo mettendo tra parentesi chi effettua cosa

### Interruptible regions e interrupting edges

meccanismo con cui possiamo andare a distruggere i token quando vengono attraversati da un **interrupting edges**. Se questo arco viene attraversato allora se c'è un qualunque attivo in quella regione allora tutti i token della regione vengono cancellati



Questa notazione ci serve per cancellare eventuali cambiamenti che sono stati effettuati all'interno del sistema.

## State Machine

una state machine comportamentale descrive un evento di un sistema o parte di un sistema.

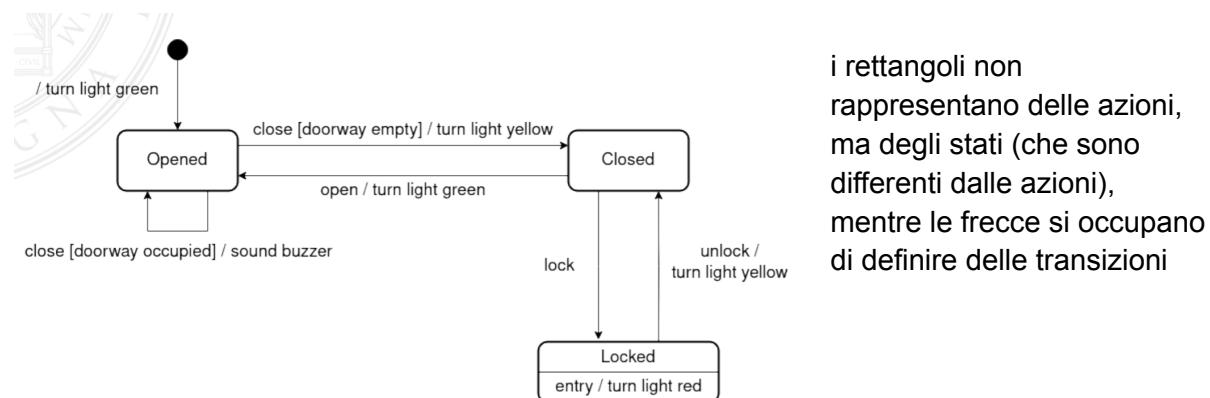
Il diagramma delle attività o diagramma comportamentale serve a definire un'orchestrazione con cui i diversi attori collaborano

Nel diagramma degli stati andiamo a vedere dentro ogni singolo elemento il comportamento che ha.

In questo caso il comportamento è dato dall'insieme di transizione degli stati che sono rappresentati sotto forma di un grafico con nodi che rappresentano gli stati e vertici che rappresentano le transizioni.

le macchine di stato possono essere di due tipi:

- behavioral: descrivono il comportamento di un soggetto. descrive le azioni che ha il singolo soggetto
- protocol state machine: servono per capire quali sono le interazioni con il soggetto



## Stato

è una situazione in cui ci valgono particolari invarianti.

Se il sistema è sempre nello stesso stato qualunque input arrivi al sistema viene processato nella stessa maniera, se lo stesso input determina l'attivazione di sistemi diversi allora il sistema si trova in stato *distinto*.

## Notazione degli stati

gli stati vengono rappresentati da dei rettangoli arrotondati in cui possiamo inserire il nome dello stato all'interno, possiamo anche identificare dei comportamenti.

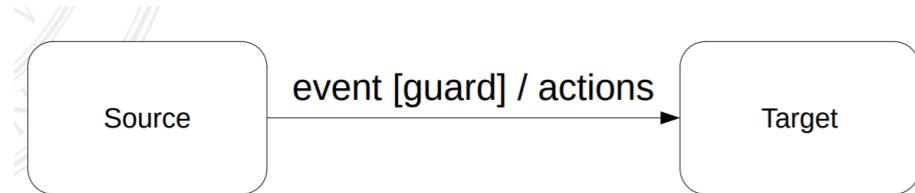
## Transizioni

ci aiutano a definire dei passaggi tra uno stato e l'altro o tra uno stato e se stesso. Sono passaggi atomici il che vuol dire che in ogni momento l'elemento si trovi in uno stato o in un altro. Non è ammesso dire che il soggetto è in una fase di transizione.

Si assume che il passaggio di stato avviene in maniera istantanea.

Le transizioni sono attivate da degli eventi. Possiamo anche associare delle transizioni a delle guardie (sono delle condizioni) in cui se la guardia è falsa allora ci potrebbe essere un cambiamento o no dello stato in cui si trova il soggetto.

### Notazione delle transizioni



### Eventi

sono occorrenze osservabili (ad esempio quale bevanda selezionare).

Nel precedente esempio abbiamo che l'evento è la selezione della bevanda, mentre le informazioni associate sono le specifiche della bevanda associata.

### Stati finali e pseudostati

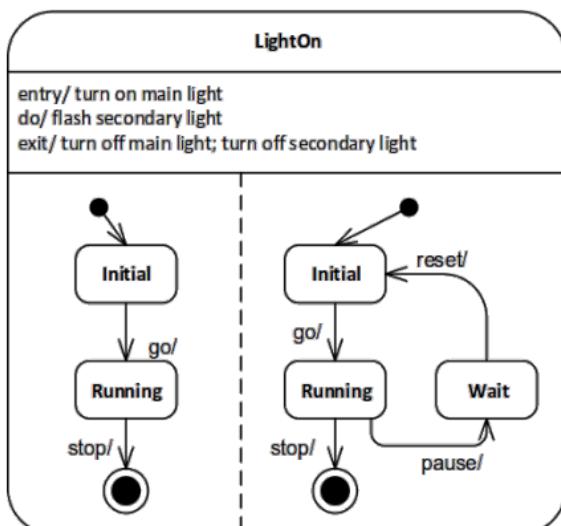
Lo stato finale serve a determinare uno stato da cui non si esce, lo stato di terminazione della vita del soggetto. Ha solo archi in ingresso e nessuno in uscita e quando si transita nello stato finale allora termina il suo ciclo di vita.

Gli pseudostati possono arricchire la semantica del nostro modello.

### Regioni

Uno stato o una transizione può essere organizzato in regioni.

le regioni ortogonali distinte servono a rappresentare le macchine interne sono in modalità concorrente.



possono essere i due possibili stati della macchina rappresentata da macchine concorrenti.

Le transizioni sono solitamente indicati da dei trigger. Quando i trigger vengono soddisfatti si passa alla conclusione delle due macchine concorrenti  
Con una singola transizione in uscita, questa viene valutata quando le macchine concorrenti vengono entrambi concluse.

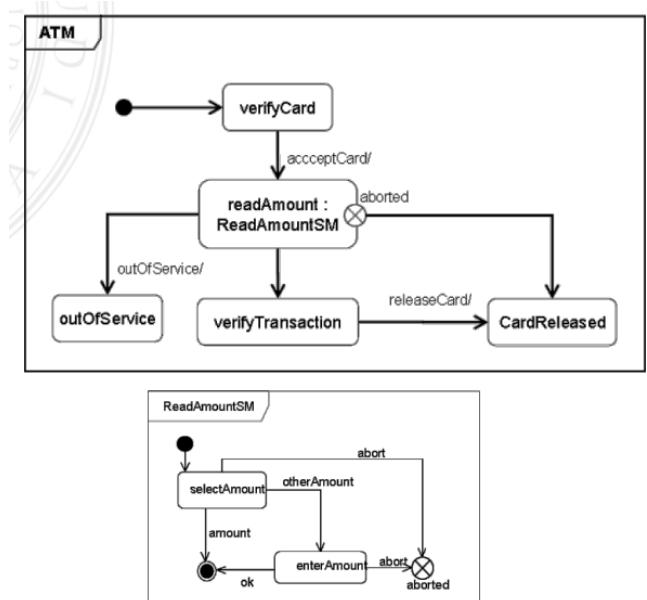
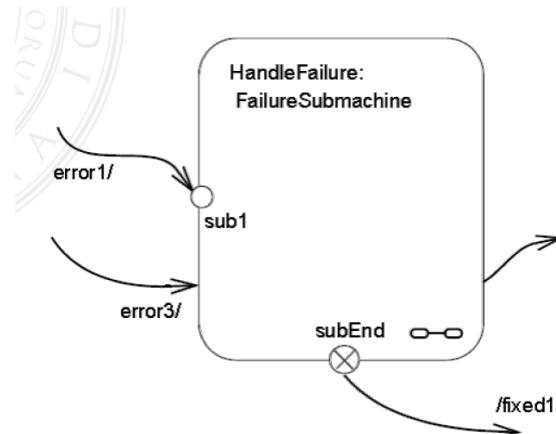
## Pseudostati (meglio non usarli)

sono altri nodi che vengono usati per controllare l'evoluzione della macchina, ma anche per tenere traccia dell'esecuzione tramite il nostro token. Quindi serve per controllare l'avanzamento dei token. Non sono stati perchè la nostra pedina a differenza negli stati, qui non si ferma.

Serve a far capire dove parte il ciclo di vita del soggetto.

Ha una transizione che lo porta al primo stato in cui si configurerà. Questa prima transizione non deve avere nessuna guardia, perchè se la guardia non passa allora si blocca la pedina e non possiamo far bloccare la pedina all'interno degli pseudostati.

*Join* e *fork* sono simili ai join e fork precedentemente visualizzati, ma con delle sostanziali differenze.



## State history

serve per tenere traccia dell'avanzamento dell'esecuzione di un sottostato.

*shallowHistory*: si ricorda solo lo stato della macchina più esterna

*deepHistory*: si ricorda anche lo stato di tutte le sotto macchine.

## Transizioni

da un punto di vista tecnico, possono essere raggiunti, o attraversati o completati.

Quando sono raggiunti la pedina è pronto per attraversarli, attraversati vuol dire che la pedina è passata e completati quando anche le transizioni sono portate alla conclusione.

Sono formate da dei trigger.

possono essere eventi o altri elementi e possono essere rappresentati da delle guardie.

**{<trigger>}\* ['[' [<guard>']]'] [/<behavior - expr>]**

*behavior-expr*: serve a specificare il comportamento della transizione.

## Compound transition

Quando abbiamo un attraverso di più pseudostati dobbiamo effettuare operazioni atomiche perché non dobbiamo fermare le pedine, perciò dobbiamo passare per le varie transizioni sapendo che dobbiamo passare dall'inizio direttamente alla fine delle transazioni, perché non possiamo fermare la pedina.

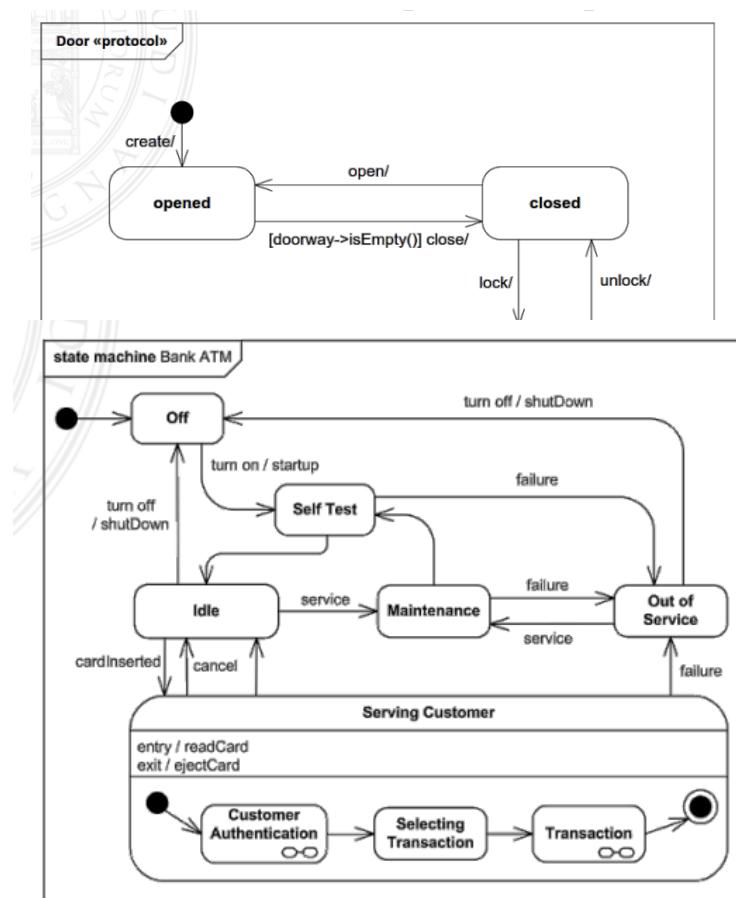
## Run to completion

Prevede che le operazioni della macchina siano run to completion.

Quando si passa da un wait (step) all'altro vengono valutate le varie compound transition

## Door protocol

Le door protocol servono per capire le interazioni con il soggetto, attraverso il protocollo. Non hanno comportamenti associati.

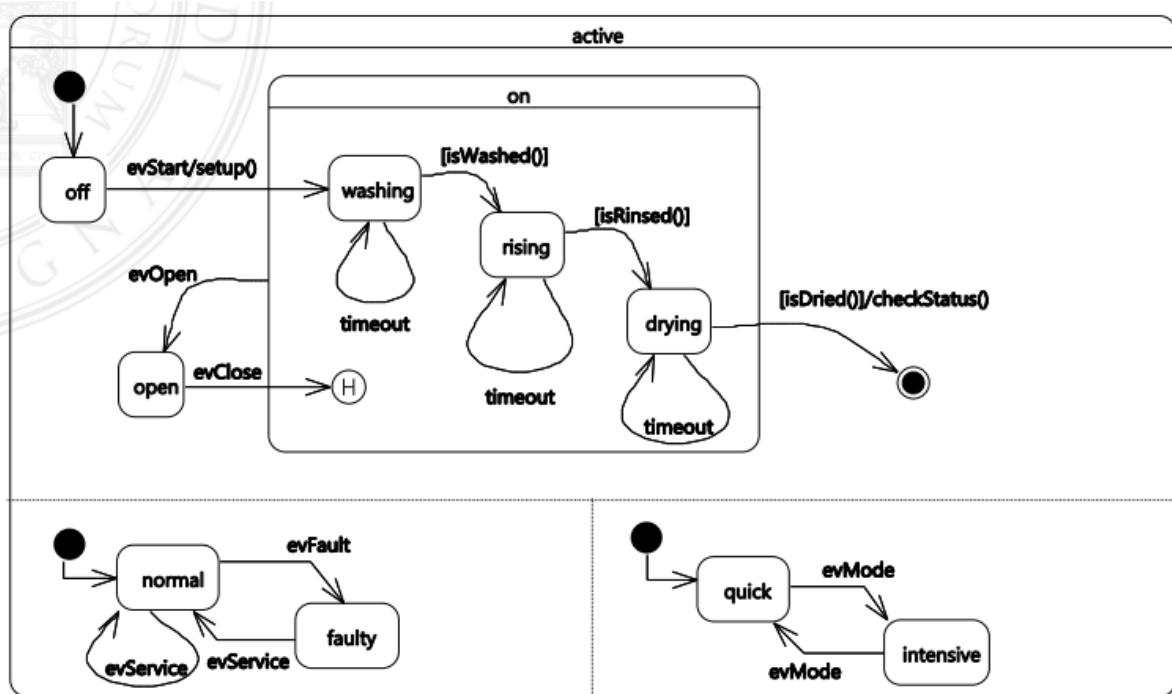


**Lock** per esempio è uno stato in cui l'unica cosa che si può fare è **unlock** per poter interagire con la macchina in quel particolare stato, cosa che invece non avviene quando si è all'interno dello stato **closed**

Il stato **self test** non vuol indicare un'azione, vuole indicare che siamo in attesa di completare il **self test**, perché è **start up** l'azione che avvia il processo. All'interno possiamo avere **out of service** quando abbiamo un failure, oppure andiamo in **Idle** se non ha problemi. Da **Idle** possiamo anche andare in **Maintenance** se ci sono problemi (tramite comportamenti interni possiamo anche ritornare in **Self Test**) oppure possiamo andare nel sottosistema di

interazione con l'utente.

La seconda freccia uscente da **Serving Customer** ci indica l'uscita e la restituzione della carta dell'utente



la sottomacchina in basso a destra ci serve per capire che in qualunque momento posso cambiare la modalità da *quick* ad *intensive*.

In basso a sinistra la parte della macchina adatta per la gestione dei malfunzionamenti. Solitamente è in stato *normal*, in caso di *evFault* andiamo in *faulty* e da lì ritorniamo in *normal* con *evService*.

Nella regione in alto abbiamo il funzionamento della lavastoviglie normale.

## OO Design

Nella parte di progettazione potremmo lavorare su diversi tipi di artefatti che ci vengono prodotti dalla fase di analisi che possono essere anche diversi tra loro.

### Super simplified UP-inspired method

Io trattiamo perché è la struttura più usata e perché si aspetta molti parametri che abbiamo in output dalla fase di analisi

Una delle caratteristiche dell'UP è essere case driven.

Come prima cosa dobbiamo trovare o creare elementi all'interno del business logic o domain layer che è il responsabile del supporto a tutte le interfacce all'interno del modello.

Il nostro problema è come esprimere dei modelli che hanno un alto livello qualitativo.

## Obiettivi di design

Dobbiamo distinguere due tipi di qualità:

- **esterne**: qualità che si riflettono sull'utente.

Parametri delle qualità esterne

- **correttezza**
- **usabilità**
- **efficienza**
- **affidabilità**
- **integrità**: non vengano alterati i dati
- **adattabilità**: che riesca a girare anche su sistemi operativi diversi
- **accuratezza**: per il calcolo matematico, più è preciso meglio è
- **robustezza**: produrre valore anche a fronte di fallimenti totali o parziali.  
Es. mi cade la rete ed il primo programma si blocca, mentre il secondo no.

In questo caso il secondo è più robusto del primo

Queste caratteristiche o qualità sono visibili o apprezzabili all'utente

- **interne**: qualità che sono relative a come è organizzato internamente il software, non appare direttamente all'utente e non apprezzate o visibili da esso.

Alcuni parametri

- **manutenibilità**: quando posso farci manutenzione sopra. Più è alto il livello più è semplice effettuare manutenzione
- **flessibilità**: disponibilità ad aggiunta di nuove caratteristiche
- **portabilità**: quanto è portabili su diversi sistemi
- **riusabilità**: quanto è semplice prendere elementi di una soluzione passata in una soluzione nuova
- **testabilità**: quanto è facile testare in maniera automatica il mio codice
- **comprendibilità**

## Software non write-once (una volta scritto non verrà mai chiuso definitivamente)

I costi associati all'evoluzione del codice può essere molto alto con una stima che può variare dal 50 al 90%.

Siccome la qualità del codice interno è critico, ovvero una bassa qualità è sinonimo di un maggiore costo in caso di aggiunta di eventuali aggiornamenti.

Per capire il livello di qualità del codice andiamo ad utilizzare degli standard che possano verificare tutti i parametri delle qualità interne

Esistono 3 modelli di qualità:

1. Software product quality model
2. data quality model
3. quality in use model

Per ognuno dei tre modelli ci sono delle particolari caratteristiche che vanno a considerare e controllare alcuni parametri come operabilità o sicurezza

## Consorzio per la qualità dei software IT

## Principi Object Oriented

abbiamo una prospettiva object oriented motivato dal fatto anche che è uno dei più usati. Abbiamo un sistema object oriented così possiamo identificare i nostri oggetti a trovare il giusto mix tra encapsulamento, ereditarietà e polimorfismo per ottenere un alto punteggio nella qualità del software.

I principi possono essere usati per garantire il massimo negli aspetti qualitativi del software.

- **Incapsulamento:** dettagli nascosti nelle classi, ma sono visibili tramite interfacce
- **ereditarietà:** condividono gli stati e i comportamenti tra le classi
- **polimorfismo:** possiamo operare sulle referenze come se fossero una sola

## Design smells

Gli smell sono degli indicatori che non necessariamente c'è un errore, ma che potenzialmente ci potrebbe essere. Ci dicono che qualcosa non è andato nel verso giusto nella progettazione del codice.

Gli smells sono:

- **Rigidità:** tendenza a resistere al cambiamento anche potenzialmente semplice  
Molto rigido se un piccolo cambiamento potrebbe portare a dei cambiamenti a cascata.
- **fragilità:** quanta "roba" si rompe quando facciamo dei cambiamenti
- **immobilità:** legata alla difficoltà di prendere un pezzo di una soluzione e trapiantarla in un'altra parte del codice
- **viscosità:** può essere dell'ambiente (ambiente inefficiente: piccolo cambiamento ci provoca un dover rieseguire il codice che magari ci mette molto tempo) o del software. Ci dice quanto resiste, passivamente al cambiamento.

- **inutile complessità**: una eccessiva flessibilità del codice può portare ad una troppo complessa struttura del codice.
  - **inutile ripetizione**: un eventuale copia incolla del codice invece di richiamare il metodo direttamente
  - **opacità**: tendenza ad essere difficile da comprendere dal punto di vista concettuale
- Alcuni principi che ci possono aiutare a progettare in maniera corretta solitamente si uniscono in insieme di principi come l'insieme **SOLID**.
- **Single Responsibility principle**
  - **Open closed principle**
  - **Liskov substitution principle**
  - **Interface segregation principle**
  - **Dependency inversion principle**

## SRP (single responsibility principle)

una classe o comunque un'entità di programma dovrebbe avere solo un tipo di ragione per cambiare.

Le responsabilità sono quindi assi di cambiamento di una classe.

Per classe di cambiamento si intende un insieme di responsabilità allineate tra di loro.

Se una classe presenta più di una responsabilità, le responsabilità diventano coppie che porta ad un problema di fragilità quando ci troviamo di fronte ad un cambiamento ad una responsabilità.

## Open close principle

Ci dice che una classe debba essere aperta per le estensioni, ma chiusa alle modifiche.

Un esempio per spiegare la definizione superiore è l'utilizzo di una sottoclasse per specificare una superclasse, senza andare a modificare la superclasse.

Non rispettare l'OCP ci porta ad un problema di rigidità in quanto un cambiamento comporta a dei cambiamenti a cascata.

E' obiettivo dell'OCP avvisarci di fare refactor qualora ci sia un problema del genere.

## Refactoring

disciplina in cui si va ristrutturare un sistema software andando a cambiare la struttura interna di esso senza cambiare l'aspetto esterno del programma e senza cambiare il comportamento di esso con l'utente con cui interagisce.

## LSP (Liskov Substitution principle)

La relazione tra le classi, nella gerarchia, dovrebbe essere sub-type

Ci dice che un oggetto di una superclasse può essere sostituito da un oggetto della sottoclasse, senza però rompere le funzionalità del programma.

Quando definiamo una sottoclasse, la classe è un sottotipo.

Non è sempre vero che costruire dei sottotipi attraverso la gerarchia ci garantisce che possiamo sostituire una classe attraverso un'altra.

Ci può anche essere delle incongruenze tra sottoclasse e classe padre dovuta al fatto che passandogli un'istanza di una sottoclasse il programma risponde in un particolare modo, mentre se gli passiamo un'istanza di una superclasse, il programma risponde in un altro modo.

Il problema che ci dice Liskov è che il linguaggio ci garantisce la compatibilità di tipi, ma non del linguaggio.

Ci tende a sottolineare il concetto di sostituibilità e di sottotipo, possono non essere coincidenti

Es. se un metodo f, accetta come argomento una reference a B e ci porta ad un risultato inatteso, quando viene passato una reference all'istanza D (sottoclasse di B), allora D è fragile nei confronti di f

Per risolvere questo problema dobbiamo pensare se ci serve o no creare un sottotipo invece di creare una nuova entità, magari anche più complessa.

Soltamente la soluzione più semplice è di usare in maniera limitata l'ereditarietà tra classi concrete, prestando molta attenzione a ciò

## ISP (interface segregation principle)

La dipendenza di una classe da un'altra dovrebbe dipendere dalla più piccola interfaccia possibile.

Non rispettare questa condizione il programma diventa più complesso diffondendo dipendenze in maniera erronea, può portare ad una eccessiva complessità e ad una potenziale violazione dell'LSP.

## DIP (dependency inversion principle)

Ci dice che le dipendenze devono essere indirizzate verso l'astrazione.

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello, ma entrambi devono dipendere dalle astrazioni (soltamente coincide con un'interfaccia) che devono essere create dai moduli di alto livello e che non devono dipendere dai dettagli (classi di basso livello), ma il contrario, ovvero che i dettagli (classi di basso livello) devono dipendere dalle astrazioni (classi di alto livello)

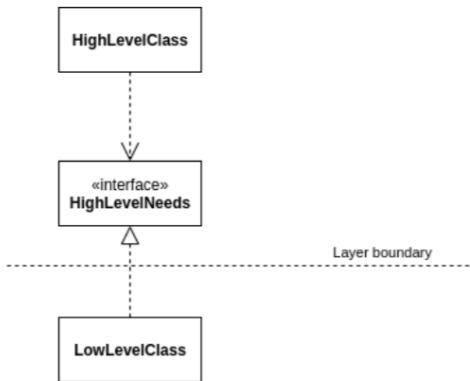
I moduli di alto livello servono per servire le funzionalità per l'utilizzatore del sistema.

E' irragionevole, perché le classi di basso livello tendono ad essere volatili perché si basano su tecnologie che si basano su quelle presenti in quel momento

E' un principio che oltre a sottolineare i problemi ci indica anche una possibile soluzione per evitare di non rispettare questo principio.

Per cercare di migliorare la situazione , dobbiamo pagare un prezzo a discapito di una riorganizzazione che ci rende più stabili.

La soluzione più comune è quella che abbiamo detto precedentemente in cui le classi di basso livello devono dipendere dalle astrazioni.



Con questo tipo di soluzione le interfacce, solitamente, sono gli elementi che sono più stabili all'interno del programma.

Solitamente quando dobbiamo rappresentarla usiamo delle **Layered Architectures** in cui andiamo a disporre le classi in base al livello a cui appartengono.

## Responsibility-driven design (GRASP)

Linee guida che ci possono aiutare, durante la progettazione della soluzione, per garantire che i principi precedenti siano rispettati.

E' una metodologia di progettazione che si basa sull'attribuire delle responsabilità a degli elementi e a garantire che i diversi elementi dialogano tra di loro in base alle responsabilità.

Fare responsabilità di un oggetto include:

- fare qualcosa per se stessi come creare un oggetto o fare dei calcoli
- iniziare un'azione in un altro oggetto
- controllare e coordinare attività in altri oggetti

Conoscere le responsabilità di un oggetto include:

- conoscere l'area privata incapsulata
- conoscere sulla relazione degli oggetti
- conoscere alcune cose che possono derivare o calcolare

Le Responsibility driven design sono assegnate alle classi di oggetti durante la progettazione di essi

Il GRASP può essere usato per capire l'assegnazione di responsabilità ed è uno strumento per aiutare la comprensione quando si fa progettazione con RDD.

Permette di definire quando dare responsabilità ad un oggetto cercando di offrire delle basi solide.

## Pattern

Per pattern intendiamo come idea di risolvere un problema ricorrente per una soluzione che ha dimostrato di funzionare. Sono problemi ricorrenti che però portano con se soluzioni che hanno dimostrato funzionare.

## GRASP Pattern

### 1. **Information expert:**

ci dice che la responsabilità deve essere assegnata a quella classe che ha tutte le informazioni necessarie per arrivare alla soluzione.

Questo ci dice però che nessuno può avere tutte le informazioni per completare la soluzione, perchè diverse classi potrebbero spartirsi le informazioni. E' il primo controllo da fare per evitare studi inutili

### 2. **Creator:**

Chi si prende la responsabilità nell'istanziare una nuova istanza dell'oggetto?

Dipende, a seconda di quante domande riesce a rispondere positivamente a delle domande. Ciascuna domanda vale un punto e viene scelto quell'entità che ha un punteggio maggiore..

Si prende un punto se:

- a. B contiene A
- b. B memorizza A in una sua variabile d'istanza
- c. B utilizza a stretto contatto con A
- d. B ha tutte le informazioni che servono per inizializzare A. Se noi abbiamo tutti i dati da passare al costruttore allora prendiamo un punto.

### **3. Controller:**

Abbiamo le system operation (operazioni tra attore e sistema), ma quando noi immaginiamo di avere una funzionalità dell'applicazione rispetto ad un input dell'esterno, chi è che riceve una **system operation**?

Controller ci dice che non dobbiamo passarlo direttamente alla soluzione, ma dobbiamo passarlo ad un intermediario.

L'intermediario può essere unico per tutto il sistema oppure possiamo anche avere un intermediario per ogni sistema o caso d'uso, a seconda di quanto è complicato il sistema.

### **4. Low Coupling:**

Low coupling e high collision possiamo considerarli come due principi valutativi. Il nostro obiettivo è quello di mantenere basso il numero di dipendenze. Dipendenza strada di diffusione del cambiamento, per cui noi dobbiamo limitare la diffusione di questo cambiamento riducendo quindi l'effetto a catena che avviene quando abbiamo dipendenze a catena.

Dobbiamo utilizzare questo principio in maniera valutativa per capire quale è l'elemento che contiene meno dipendenze e sceglierlo.

### **5. High Cohesion:**

Gli insiemi delle responsabilità devono essere coesi. Le responsabilità devono essere assegnate per mantenere alta la coesione tra di loro.

### **6. Pure Fabrication:**

Quando dobbiamo assegnare una responsabilità e gli elementi che abbiamo analizzato non sono buoni e non ci sono elementi del dominio da saccheggiare, ci fanno comodo delle classi che sono astratte, entità che esistono solo nello spazio della soluzione e che non sono legate al dominio.

Sono classi che sono pensate per essere legate alla soluzione e che sono inventate apposta per migliorare in maniera artificiale il livello di qualità interno della progettazione.

Sono classi fortemente coesive, progettate per ottimizzare le dipendenze ecc.

(Solitamente nel caso in cui non ci sia nessuna "soluzione" possiamo creare un nuovo elemento la cui prima fonte di ispirazione sono gli elementi del dominio. Se devo creare una nuova classe andiamo a vedere nel dominio se c'è qualche elemento che possiamo trasformare.) DA RIFINIRE E MIGLIORARE

### **7. Indirection:**

Quando l'assegnazione delle responsabilità ci porta ad un eccessivo accoppiamento dobbiamo andare a togliere l'accoppiamento delle responsabilità.

E' un'istanza specifica della Pure Fabrication.

Ci permette di spostare le dipendenze

### **8. Polymorphism:**

Ci dice che nel nostro codice che se ci rendiamo conto che una responsabilità si trasforma in un metodo e che implementandolo andiamo ad usare strumenti di controllo del flusso in cui andiamo a distinguere i comportamenti che ci portano una ad una variazione a seconda dell'elemento su cui stiamo lavorando allora stiamo sbagliando.

Normalmente il polimorfismo è un meccanismo che ci permette di attivare comportamenti diversi a seconda di ciò che gli passiamo senza andare ad utilizzare controlli di flusso.

il polimorfismo è importante perchè se ci dovessimo trovare in diverse parti del programma, e dovessimo usare dei controlli di flusso come if per capire a quale classe appartiene, il programma risulta essere fragile perchè in caso di cambiamento dovremmo fare dei grossi cambiamenti al codice.

#### **9. Protected Variations:**

Ci dice che ci sono elementi nel codice che sono presumibilmente più soggetti al cambiamento rispetto ad altri.

Bisogna però controllare che un aumento della complessità non sia inutile in quanto molto difficilmente potrebbe non essere soggetto a modifica, portandoci ad un aumento della complessità in modo inutile.

Una buona struttura può aiutare a rispettare il LSP evitando anche altri problemi. Esiste una regola (**Law of Demeter**) che limita la possibilità di richiamare metodi a seconda dei parametri che vengono passati. Questa regola viene suggerita (anche se non particolarmente sensata) tendendo a garantire la protected Variation.

## Agile

Nella realizzazione del codice potremmo avere un problema di overhead, soprattutto se dobbiamo creare tanti artefatti intermedi che non servono per la realizzazione del codice.

Bisogna quindi trovare un punto di equilibrio tra le due cose per evitare di avere uno sbilanciamento tra la realizzazione e l'effettiva implementazione del codice.

Cercare di fare uno sforzo costoso per migliorare il processo, non necessariamente si trasforma in un miglioramento del prodotto.

Tutto il tempo speso per definire l'adesione al piano, negoziazione dei contratti, documentazione su come stiamo portando avanti il processo, valutazione dei rischi potrebbero essere attività inutili nel caso in cui non sappiamo cosa dobbiamo fare o cosa stiamo facendo.

Molto spesso le specifiche cambiano, portando quindi ad uno spreco in queste attività qualora ci sia un cambiamento importante o tanti cambiamenti in breve tempo.

## Manifesto for Agile Software Development

E' un documento che ci permette di scoprire nuove via sempre migliori di sviluppo software facendolo e aiutando gli altri a farlo.

Attraverso questo lavoro diamo valore a:

- individui ed interazioni su processi e tool
- Software che lavorano su documenti comprensivi
- Collaborazione con il cliente sulla contrattazione del contratto
- responsività ai cambiamenti sul seguire il piano di sviluppo

Quello che ha sinistra, ha più valore rispetto agli elementi scritti a destra (processi e tool)

Agile software non è un processo o una metodologia di sviluppo.

è una **collezione di pratiche** guidate da dei **principi** ispirati a dei **lavori**.

## Principi

- la nostra più alta priorità è di soddisfare il cliente attraverso continue consegne, anche anticipate rispetto alla consegna di software che può valutare. Andiamo quindi ad includere il cliente nello sviluppo facendogli vedere come procede il software mostrandogli delle demo funzionali
- Ben vengano i cambiamenti, anche in ritardo nello sviluppo. I processi agili sfruttano il cambiamento per avere del vantaggio competitivo verso il cliente
- Consegnate frequenti del software su cui stiamo lavorando in un tempo compreso tra alcune settimane ed alcuni mesi
- uomini d'affari e sviluppatori devono lavorare insieme, giornalmente su tutto il progetto
- Costruire progetti attorno a persone che sono motivate
- Offrire l'ambiente ed il supporto che hanno bisogno per portare a termine il lavoro
- il miglior modo di trasporto è il dialogo faccia a faccia a da parte del team
- Un software funzionante è la misura primaria del progresso
- Un processo Agile promuove uno sviluppo sostenibile.

- Sponsor, sviluppatori ed utenti devono essere in grado di mantenere un ritmo costante
- Una costante ricerca all'attenzione, all'eccellenza tecnica ed un buon design migliora l'agilità
- Semplicità: l'arte di massimizzare la quantità di lavoro non fatto è essenziale
- Le migliori architetture, requisiti e design derivano da team che si auto organizzano
- ad intervalli regolari il team riflette su come diventare più efficiente, sintonizzandosi e regolando il proprio comportamento di conseguenza

## Metodi Agile

- Agile modeling
- Agile Unified Process
- Crystal Clear
- Extreme Programming
- Scrum: metodo più diffuso

questi metodi sono tutti metodi che si rifanno ai principi precedenti.

Le differenze consistono nelle pratiche (che possono essere diverse) che possono essere diverse tra di loro

## Pratiche

Le pratiche seguono tutti i principi che ha l'Agile.

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• Refactoring</li> <li>• Small release cycles</li> <li>• Continuous integration</li> <li>• Coding standard</li> <li>• Collective ownership</li> <li>• Planning game</li> <li>• Whole team</li> </ul> | <ul style="list-style-type: none"> <li>• Daily meetings</li> <li>• Test-Driven Design</li> <li>• Code and design reviews</li> <li>• Pair programming</li> <li>• Document late</li> <li>• Use of design patterns</li> <li>• ...</li> </ul> |
|---|---|

Con collective ownership vuol dire che è il team che deve aver proprio il codice e non una sola persona, per evitare che qualora quella particolare persona dovesse mancare non si blocchi tutto il processo

Test-Driven Design: ci dice che prima di scrivere del codice dobbiamo fare dei...

Document late: documentazione opportuna, senza perdere tempo nel documentare per evitare di buttare via tempo qualora quello che abbiamo fatto sia in linea con il cliente

Tipicamente la nostra metodologia prende alcune pratiche

Possiamo dividere le pratiche in tecniche (pratiche con cui si scrive e si progetta la soluzione) e pratiche che servono per la gestione del progetto.

## Test Driven design

stile di programmazione in cui le classiche attività che vengono realizzate sono invertite rispetto al normale 'ordine. Si parte dal test che il codice che deve superare e poi il codice stesso.

Può essere descritto seguendo le seguenti regole.

1. scrivere una singola unità di test che mi descriva un aspetto del programma
2. runnare il test, che dovrebbe failare perché il programma non ha tale feature
3. scrivere abbastanza codice, il più semplice giusto per passare il test.
4. si fa un refactor del codice andando a modificare fino a che non rispetta i criteri di semplicità
5. ripete accumulando le unità di test

## Code review

dice che quando viene scritto del codice, il codice non può essere direttamente aggiunto al codebase, ma deve essere revisionato ad un altro membro del team. Se viene approvato allora viene aggiunto al codebase.

Questa pratica serve ad avere un doppio valore in cui il valore secondario risulta essere più importante rispetto a quello primario (sottoporre il codice a revisione ci aiuta ad evitare possibili sviste o errori) che ci porta a diminuire la probabilità di incappare in errori.

Il valore secondario preminente ci dice che obbligando a fare del review da parte di più membri del codice aiuta a diffondere la conoscenza di come sta avanzando la soluzione del codice, andando anche ad ampliare la conoscenza del linguaggio ai membri del team.

Questo valore risulta essere anche più importante del primo punto della correttezza del codice.

## Pair programming

forma estrema di code review che ci dice che quando ci sediamo a scrivere codice, lo facciamo in coppia in cui giochiamo in due ruoli, driver e navigator in cui ci si scambia anche di posto invertendosi.

Oltre ai vantaggi elencati nel code review ha un vantaggio ulteriore è che non c'è il tempo morto tra la conclusione del codice e la revisione di esso.

è però fortemente vincolante ed infatti non è molto usato, alcuni lo usano solamente alcune volte, altre volte viene usato sostituendolo con altre pratiche Agile.

## User stories

Prendono il ruolo di quello che è dalla descrizione dei casi, pur non essendo dei casi d'uso. sono delle frasi concise che servono a catturare le aspettative dell'utente nell'interagire con il sistema.

In quanto ruolo voglio raggiungere quel particolare obiettivo per un beneficio personale.

Se non c'è valore allora implemento un qualcosa che all'utente non serve.

Mettere insieme una serie di frasi per catturare degli aspetti e delle funzionalità possono far aumentare notevolmente la complessità.

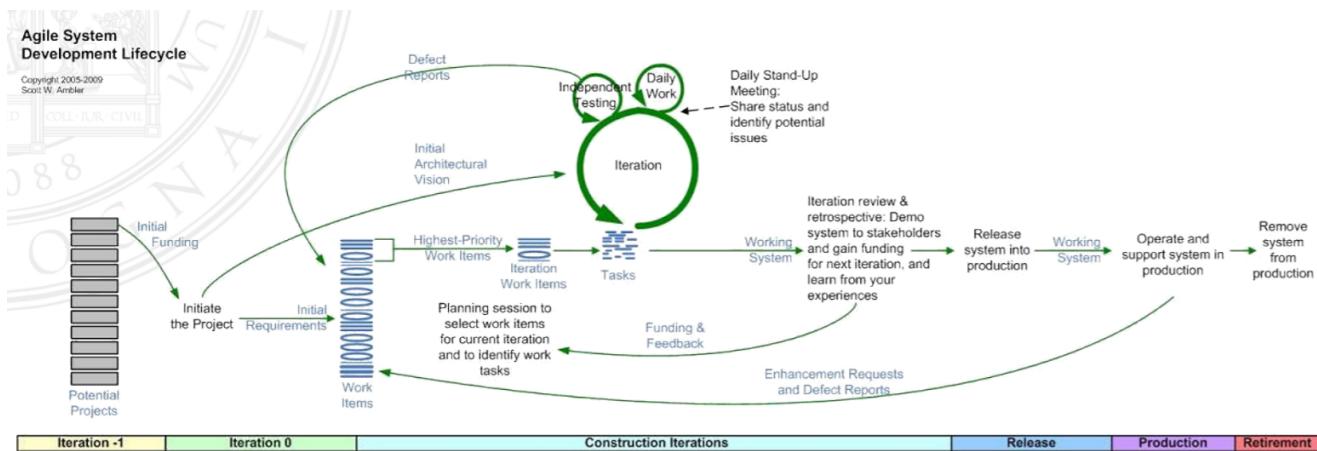
Un altro problema è che le frasi devono essere validate attraverso un concetto di accettabilità tramite INVEST

## Invest

Ci aiuta a memorizzare una serie di qualità sotto forma di checkbox

- **indipendente**: rispetto alle altre user stories
- **negoziabile**: devono essere negoziabili con il committente
- **valutabile (di valore)**: deve produrre del valore al cliente, specificando un beneficio
- **stimabile**: stimabile di effort necessario per realizzare il sistema che accetta la user stories
- **piccola**: devono essere frasi molto concise
- **testabile**: ogni user stories deve avere associata un meccanismo per verificare se il sistema è in grado o no di supportare la feature

## Agile life cycle



## Agile ed evoluzione

Per un certo tipo di lavoro un team può trovarsi meglio con Agile, un altro team che ha competenze e modi di lavorare diverso può trovarsi meglio con altri strumenti.

Agile si porta dietro delle problematiche come lo sviluppo del software che è solo una parte come la parte evolutiva.

In Agile le informazioni si scambiano a voce tra i membri dei team, arrivando anche a suggerire di creare dei meeting per scambiare la conoscenza, ma questo ha un problema in quanto queste conoscenze possono essere volatili qualora il team di sviluppo del codice dovesse andare via dalla nostra azienda portandosi via la conoscenza di esso.

Solitamente si va a fare una documentazione quando il team deve andare via, ma questo non avviene mai e la documentazione non viene mai effettivamente effettuata.

## Extreme programming (XP)

Si basa su 4 attività: coding, testing, listening e designing,  
5 valori: comunicazione, semplicità, feedback, coraggio e rispetto

## 3 principi

### XP practices

1. fine scale feedback: il nostro processo è pensato per avere feedback sul nostro progetto il prima possibile
  - a. pair programming: feedback all'interno del team stesso
  - b. planning game: si va decidere quali elementi devono essere estratti per il passo successivo.
  - c. test driven development:
  - d. whole team: significa che le competenze necessarie devono essere tutte possedute dai membri del team
2. continuous process
  - a. continuous integration: ogni volta che ho scritto un nuovo pezzo di codice vado a rieseguirlo immediatamente per andare a creare un nuovo prototipo
  - b. design improvement: si basa in particolar modo sul refactoring
  - c. small releases: facciamo delle release in tempi ragionevoli per confrontarci con l'utente oltre a quelle che facciamo all'interno del team e che l'utente non vede
3. shared understanding: pratiche che sono legate alla necessità di avere una comprensione comune su tutte le caratteristiche del codice
  - a. coding standard: uno standard che il team deve seguire per evitare problemi
  - b. collective code ownership: non ci può essere il caso in cui una singola persona ha il possesso del codice
  - c. simple design: richiede di trovare soluzioni che siano semplici, facilmente condivisibili e capibili
  - d. system metaphor: suggerisce di usare anche nella comunicazione interna al team delle metafore per referenziare delle parti dentro il nostro programma
4. programmer welfare:
  - a. sustainable pace: pratica in cui ci dice che se domani dobbiamo consegnare un progetto che ancora non è mai finito, non possiamo fare straordinario e se qualcosa non funziona andiamo dal committente con un qualcosa non previsto.  
Con uno sprint a ridosso della deadline non funziona, anzi ci porta a dei problemi dovuto alla poca attenzione e alla fretta di concludere

### Planning game

tutte le metodologie agile richiedono ogni volta le cose che devono essere affrontate.

Il planning game si tratta del processo di pianificazione dell'interazione.

E' basata su user stories e si divide in due parti:

- release planning: include anche i clienti
- iteration planning: coinvolge solo gli sviluppatori

L'idea è che il committente ordina le storia in ordine all'importanza che ha per loro.

L'idea è che si cerca di trovare un ragionevole meccanismo per capire le necessità del cliente e del team di sviluppo ordinandole in storie critiche (quelle più importanti) in significative business value e nice to have (belle da avere, ma che sono opzionali).

il team invece fa l'ordinamento a seconda del rischio (a seconda delle funzionalità che dobbiamo aggiungere per far funzionare il programma).

## Validazione e verifica

Il testing fa parte delle attività che solitamente fanno parte delle attività di verifica e validazione

- Verifica:  
Di verifica sono le attività che valutano gli artefatti prodotti tramite una maniera interattiva da persona o automatizzata dal software testing.
- Validazione: Serve per capire se il software riesca a rispondere a ciò che ha richiesto l'utente

Un effetto collaterale del testing che viene apportato dall'adozione di test automatico deve essere ben ingegnerizzato.

Con il testing andiamo a mostrare semplicemente eventuali errori che ci sono e questo non vuol dire che va bene il codice, ma che non ci sono errori.

## Testing e costo

Altra cosa da tenere conto è il costo quando noi troviamo dei difetti, più tardi noi troviamo questo problema peggio è in termini di costo dover sistemare questo problema.

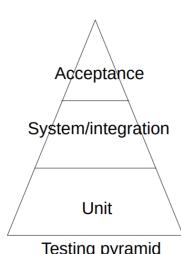
Con fasi molto avanzate di codice abbiamo che il problema potrebbe essere molto importante quando noi stiamo facendo affidamento su un errore che se sistematico potrebbe non far funzionare il nostro codice.

## Jargon

- Bug (defect): risultato di un errore logico che fa funzionare in maniera inaspettata il nostro codice da ciò che ci aspettiamo  
Ovviamente un errore logico potrebbe non essere un vero problema perché non è una failure, per esempio bug con valori negativi, ma noi usiamo solo valori positivi
- failure: effettivo errore all'interno del nostro programma che non lo fa funzionare in maniera corretta
- Issue: descrive la situazione in cui eravamo quando abbiamo un failure, tramite report
- Test case: descrive ciò che si aspetta lui in output, includendo dati, condizioni e risultati attesi.  
Sono collezionati, solitamente in gruppi

## Testing levels

piramide che dice le unità di testing che vengono prese in considerazione



I test si realizzano in maniera diversa a seconda della categoria a cui appartengono:

- test statici: test analizzato senza mandare in esecuzione
- test dinamici: quando il test viene mandato in esecuzione

## Test statici

Per le applicazioni critiche si tende a fare un mix di statico e dinamico.

Nei test statici andiamo ad utilizzare un aspetto formale come data checking ecc.

Non ci dicono se ci sono errori o no nel codice, ma ci possono dare delle garanzie formali in determinate situazioni.

I test statici vengono usati per applicazioni che nascono critiche e dove non bisogna eseguire algoritmi complicati da eseguire, ma che comunque diano garanzie che funzioni. Altro caso è dove un eventuale aggiornamento diventa critico.

## Black Box testing (test dinamici)

quando non abbiamo conoscenza del codice e lo eseguiamo per capire quali problemi ci sono e come funziona.

In state transition tables andiamo a vedere come cambia l'esecuzione con stessi input con però stati diversi. Tali situazione potrebbero fornire output diversi nonostante abbiamo gli stessi input

All pairs - testing: andiamo a considerare tutte le combinazioni possibili dei parametri.

### Boundary value e equivalence partitioning

si cerca di identificare i punti di discontinuità.

Esempio di equivalence partitioning:

discounted receives the order amount and  
returns a discounted value applying a 5% discount  
if  $1000 < \text{amount} < 5000$  and 10% discount if amount  
 $\geq 5000$

- Partitions:

- $[\text{INF}, -1]$  because of input domain
- $[0, 1000][1000, 5000][5000, \text{SUP}]$

- Possible values:

- -10, 100, 3000, 8000

## White Box testing

conosciamo la struttura interna del codice.

code coverage: passiamo per tutte le righe di codice

branch coverage:

path coverage:

## Pro e contro di approccio White Box e Black box

	Pro	Contro
Black box	<ul style="list-style-type: none"><li>• i tester non sono i programmati del codice (evita conflitto di interesse)</li><li>• si avvicina notevolmente ai requisiti richiesti</li></ul>	<ul style="list-style-type: none"><li>• conoscenza del codice sconosciuta</li></ul>
White Box	<ul style="list-style-type: none"><li>• conoscenza del codice acquisita quando si va a creare il test</li><li>• alta copertura (cercare di massimizzare il test per più righe di codice possibile)</li></ul>	<ul style="list-style-type: none"><li>• complessità: possiamo avere alta complessità nella creazione dei test</li></ul>

## Testare il test

Non tutti i test sono uguali e devo capire quale test usare per il mio programma come posso capire la qualità dell'insieme dei miei test?

Tramite **mutation testing**(meccanismo che mi permette di capire se il mio testing riesce a trovare errori):

- Se abbiamo scritto bene i test, lui passa i test, ma se il retino riesce a catturare i nostri problemi, andiamo a creare dei mutanti del codice su cui dobbiamo testare, ovvero codice che ha piccoli cambiamenti all'interno del nostro codice. Questo ci dovrebbe provocare degli errori ed un buon test set deve riuscire a catturare tali errori cacciando quindi i mutanti.

## Unit testing

non possiamo scomporre elementi in unità più piccole.

serve a capire che il codice funzioni come atteso e che i codici di unità vengano rimandati in esecuzione quando ci sono modifiche al codebase, non solo la parte di modifica, ma anche la parte di codice che non abbiamo modificato perchè un'eventuale modifica nel codebase può aver provocato errori anche nella parte già testata

Deve avere caratteristiche particolari:

- deve poter fare delle verifiche isolando l'elemento che sta testando evitando che metta in esecuzione l'intero sistema

- per poter isolare correttamente gli elementi possiamo usare diversi meccanismi (**Isolation**)
- all'interno del nostro test i test case devono essere indipendenti senza che un test debba dipendere dal precedente e così via

## Isolation

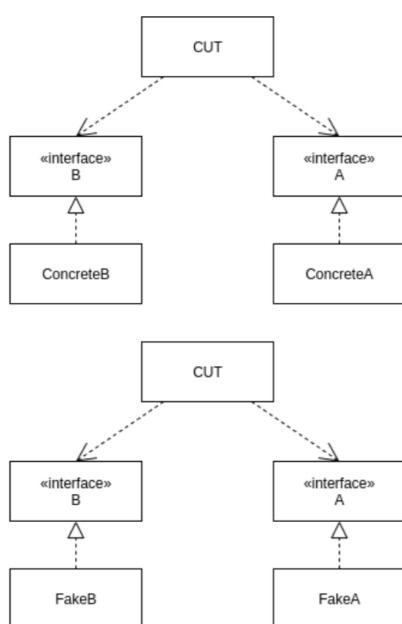
andiamo a creare oggetti finti (mock) che sostituiscono le dipendenze reali.

Per fare ciò andiamo a caricare gli oggetti finti con il minimo indispensabile per poter far funzionare il test.

Un codice testato su dipendenze reali abbiamo il rischio che deve istanziare tutto il programma e che potremmo comunque non capire dove sia il problema.

Per sostituire oggetto finti al posto delle dipendenze dobbiamo andare a fare un collegamento tra le classi e le interfacce invece di collegarle direttamente con una classe. Una classe per essere testabile deve essere associata ad un'interfaccia e non direttamente collegata con una classe.

Dal punto dell'ingegneria del software l'uso del testing va a migliorare la qualità del codice stesso



## Xunit

Realizzare un test vuol dire creare veri e propri programmi e come tali devono seguire tutto il processo di vita dei programmi.

per avere un sistema che mi gestisca tutti i test andiamo ad utilizzare un approccio su un frame di tipo XUnit.

XUnit: ambiente per definire:

- test runner
- test case
- test fixtures: contesti in cui si trovano
- test suites:
- test execution: tracce di esecuzione dei test

## Design pattern

La definizione pattern è una soluzione che si sa che funziona per tutti i problemi che conosciamo

I pattern solitamente sono raggruppati in strutture coerenti (cataloghi) che presentano un proprio vocabolario, propria sintassi e grammatica.

## Cataloghi dei pattern

Esistono diversi cataloghi dei pattern e che possono essere diversi oltre alle differenze strutturali anche quelle di design (dove cambia la strutturazione dei pattern)

## Documentazione

comprende una serie di elementi:

- nome del pattern
- intento
- eventuali nomi alternativi
- forze che ti spingono all'adozione del pattern
- applicabilità: quando è applicabile
- struttura: diagramma che rappresenta le classi
- partecipanti: che ruolo svolgono gli elementi
- collaborazione: come interagiscono
- conseguenze
- implementazione:
- esempi di codice
- usi conosciuti: catalogo del pattern che vuole essere una conoscenza pregressa affinando la conoscenza precedente
- pattern correlati: altri pattern che possono essere correlati con quello che stiamo usando in quel particolare momento

## Tipi di pattern

- Creazionali: hanno a che fare con il problema di istanziazione di nuovi oggetti  
Il problema della creazione degli oggetti non ha soluzione banale
- Strutturali: riguardano come classi ed oggetti sono composti o scomposti per formare una struttura coerente per risolvere il problema
- Comportamentali: si occupano di algoritmi e di assegnazione di responsabilità tra oggetti

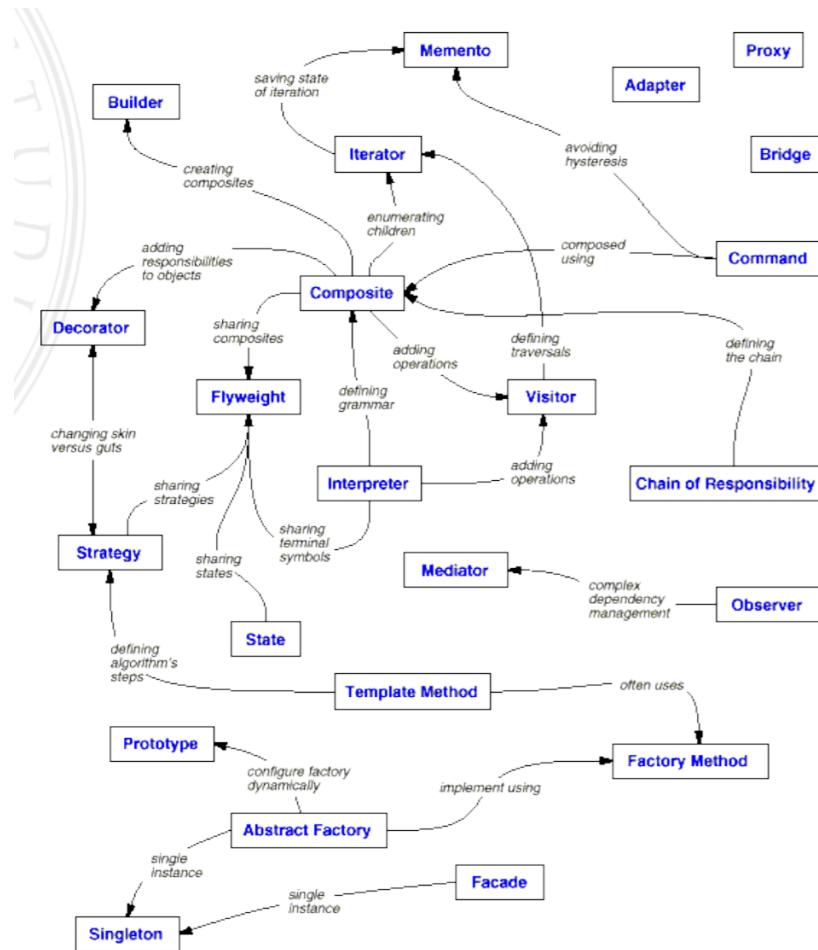
i 23 pattern fondamentali

- **Creational:** Abstract Factory, Builder, Factory Method, Prototype, Singleton
- **Structural:** Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- **Behavioral:** Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

		Purpose		
Scope	Class	Creational	Structural	Behavioral
	Object	Factory Method Builder Prototype Singleton	Adapter (class)  Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Interpreter Template Method  Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

La maggior parte dei pattern non opera al livello classe, ma opera collaborando con esse. Buona soluzione perchè vedendolo con diversi punti di vista possiamo avere visione diverse a seconda di come lo guardiamo.

Soluzioni valide perchè permettono di ottenere una soluzione che esprime un buon livello qualitativo oltre a fornire il risultato corretto.



## Composizione sull'ereditarietà

Una caratteristica comune tra tutte le soluzioni è che andando a legare le varie entità viene fuori che una delle soluzioni migliori in cui dobbiamo andare a trovare un nodo che mette in comune gli aspetti è quello di favorire la composizione a discapito dell'ereditarietà.

Le due soluzioni più immediate sono quelle di andare a creare una superclasse andando ad ereditare il metodo alle due sottoclassi, senza definirlo due volte (cosa vietata).

Altra soluzione può essere quella di andare a spostare il codice condiviso in una classe totalmente nuova che però non è gerarchicamente legata alle due classi.

**Composition** perchè avendo le classi che delegano (**delegation**) a questa classe, dobbiamo inserire delle reference che non la compongono, ma l'aggredano.

### Problema ereditarietà

Ereditare il metodo porta però a dei problemi che non ha chi usa la delega, anche se dobbiamo andare ad inserire le reference nelle due classi, nonostante sia più semplice ed immediato.

Problemi che però sono legati alla struttura ed ai costrutti del linguaggio che andiamo ad utilizzare.

Cerca di risolvere due problemi che sono distinti, ma che sembrano andare bene insieme.

Si basa sul principio di sostituibilità di Liskov che ci dice che se sono sostituibili sono sottotipi.

Il ragionamento che fa il nostro linguaggio è però il contrario ovvero che se sono sottotipi allora sono sostituibili, perché è possibile verificarlo in maniera automatica.

Dal punto di vista del linguaggio gli va bene se gli passo un sottotipo invece che un tipo diverso perché il tipo diverso può avere metodi diversi rispetto a quelli del tipo principale, quindi si trova metodi diversi da quelli che si aspetterebbe.

I tipi mi servono a garantire che vado a richiamare i metodi corretti della classe.

Questa compatibilità di tipi ci serve perché così l'oggetto ha la garanzia che gli arrivano i metodi corretti.

La struttura dell'eredità serve per consentire la sostituibilità, il linguaggio garantisce degli strumenti automatici per garantire ciò.

Il vantaggio dell'ereditarietà è il fatto che creando una sottoclasse eredita i metodi della superclasse.

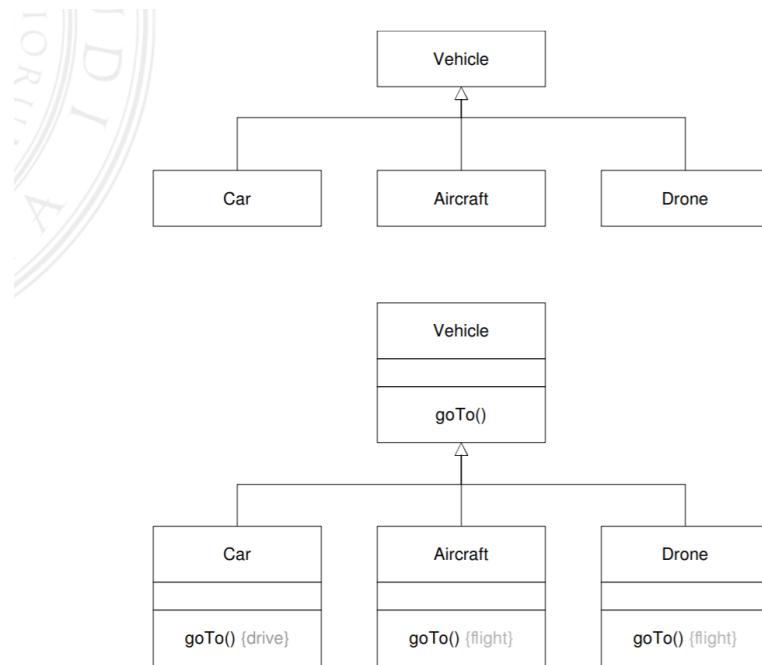
Ti forza, anche, ad ereditare il corpo di quei metodi anche se non ci sarebbe necessità di esso.

Quello che a noi interessa è il meccanismo che però è troppo generoso e si porta dietro anche il comportamento.

Noi vogliamo sostituibilità per

Ereditarietà definisce delle compatibilità portandosi dietro del codice che diventa comune da una classe all'altra, cosa che però è superfluo.

**Ereditarietà si porta dietro sostituibilità e condivisione del codice, unendo due in uno, cosa che però, come ribadito prima è superfluo.**



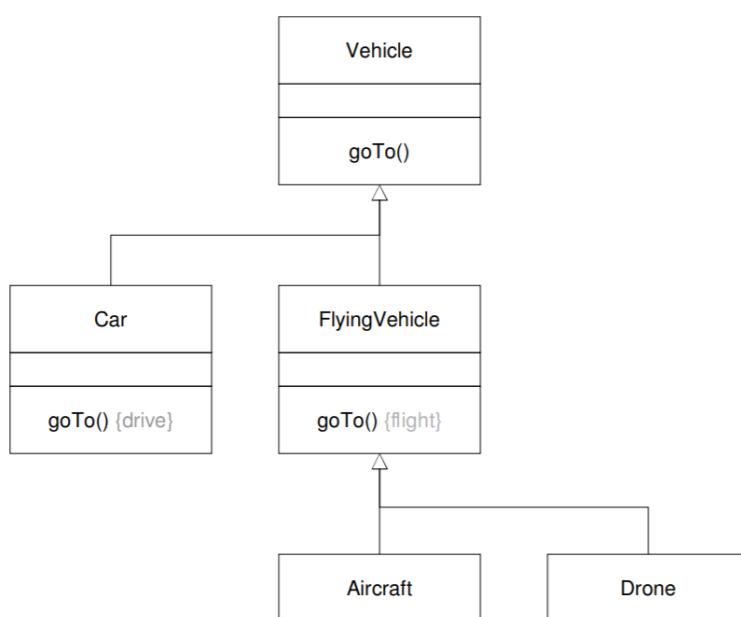
Nell'esempio indicato abbiamo che `goTo` è una caratteristica comune che speriamo sia a livello base, perchè mi aspetto che tutte le sottoclassi abbiano la stessa caratteristica (`goTo`)

Avendo però diversi modi di spostamento ci aspettiamo che il metodo venga ereditato, ma che poi bisogna andare a modificare.

Dobbiamo andare a creare un metodo virtuale (*abstract*) andando a fare *Override* del metodo.

E' nelle sottoclassi che andiamo a scrivere effettivamente il corpo del codice del metodo.

Il problema è che in *Aircraft* e *Drone* presenta lo stesso metodo che deve svolgere in ugual maniera portandoci quindi ad un problema di ripetizione del codice.



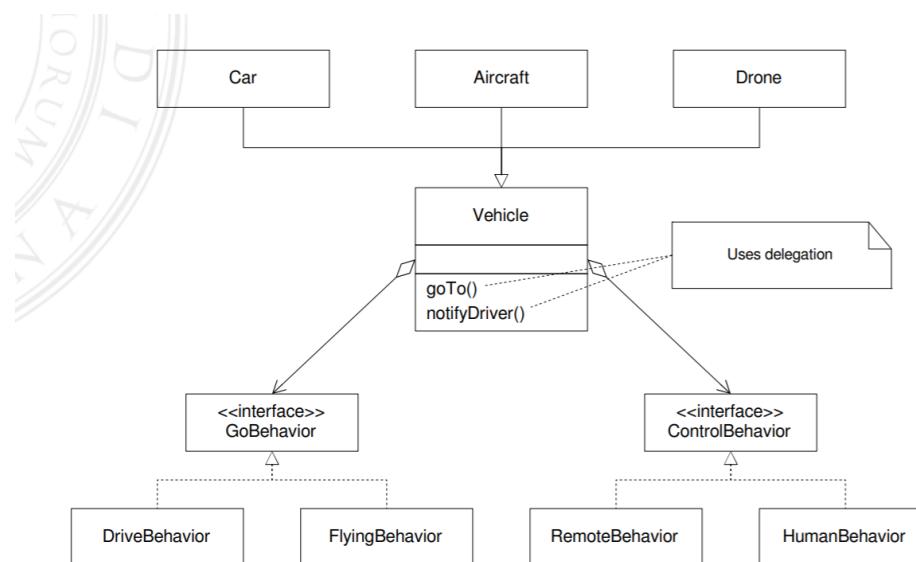
La soluzione consiste nel creare una superclasse intermedia in cui andiamo ad inserire il metodo all'interno che verrà ereditato.

Nel caso in cui dovessimo aggiungere una funzionalità come *notifyDriver* iniziano i veri problemi, perchè *Car* e *Aircraft* hanno lo stesso metodo, mentre *Drone* no e quindi siamo nello stesso problema.

Non andiamo a considerare gli aspetti Object Oriented e Protected Variation.

(Protected Variation: struttura il codice in modo che qualora ci fossere dei cambiamenti non dobbiamo andare a modificare tutto il codice)

La soluzione consiste nell'andare ad utilizzare delle interfacce che però non ci porta ad ereditare quelli che sono i metodi della classe, quindi ci serve anche una classe o più classi che ci serve per delegare.



Una struttura del genere va a far uso della delegazione ad un'altra classe e che possiede i due metodi.

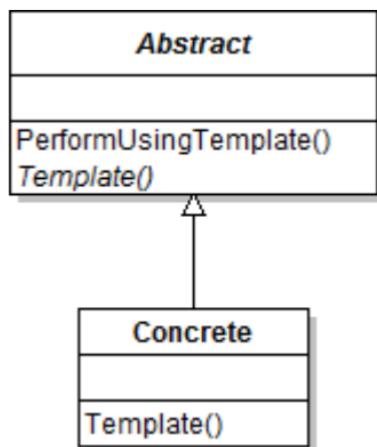
Le classi esterne (quelle delegate) sono classi concrete che devono contenere solamente il codice che ci serve.

Car richiamerà il `goTo()` all'interno della classe delegata `DriveBehavior`, che si occuperà di fare lo spostamento.

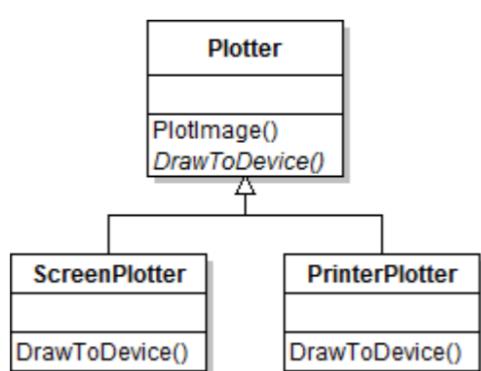
## GoF: Template Method

Definiamo lo scheletro di un algoritmo all'interno di una certa classe delegando alcuni suoi passi ad una sottoclasse.

L'idea è quello di :



Questa scelta mi viene comoda per gestire la struttura di un algoritmo che può essere comune a più classi e che presenta piccole variazioni interne a seconda dell'oggetto che viene dato in input.



Con questa soluzione andiamo ad utilizzare le parti che sono in comune tra le due sottoclassi all'interno della superclasse, mentre le due differenze le andiamo a scrivere all'interno del metodo astratto che viene ereditato.

L'idea che sposto il comportamento comune nella classe principale, anche se non definito totalmente.

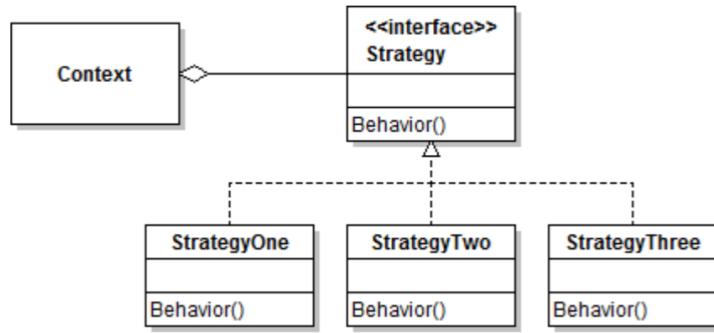
Andiamo quindi a creare un algoritmo astratto che dovrà poi essere concretizzato all'interno delle sottoclassi.

Questa soluzione ci da un vantaggio dal punto di vista qualitativo.

Le dipendenze sono dirette verso gli elementi più stabili aiutando ad aderire all'OCP (Open Close principle)

## GoF: strategy

definiamo una famiglia di algoritmi che possono essere incapsulati e scambiati.  
La strategia ci permette di variare indipendentemente dai clienti che lo utilizzano



Context (classe in cui andiamo a delegare)

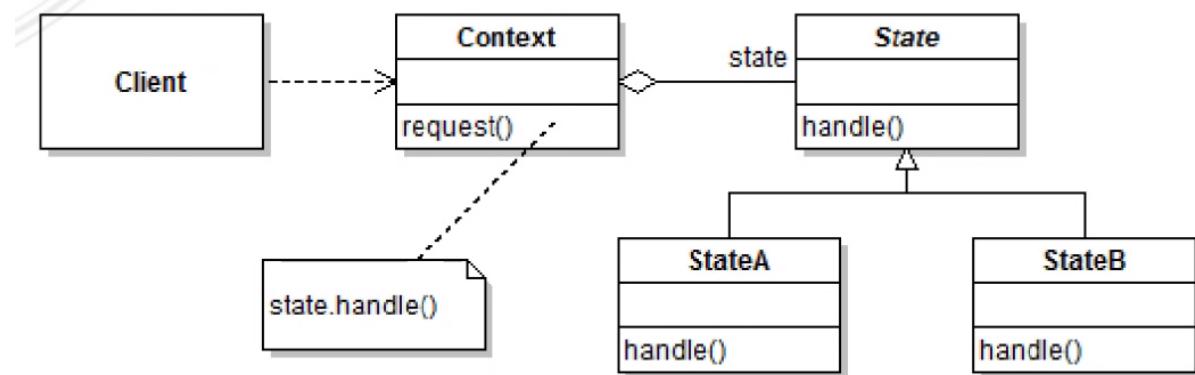
Aiuta a rispettare l'OCP, l'Obeys Protected Variation e favorisce la composizione rispetto all'ereditarietà.

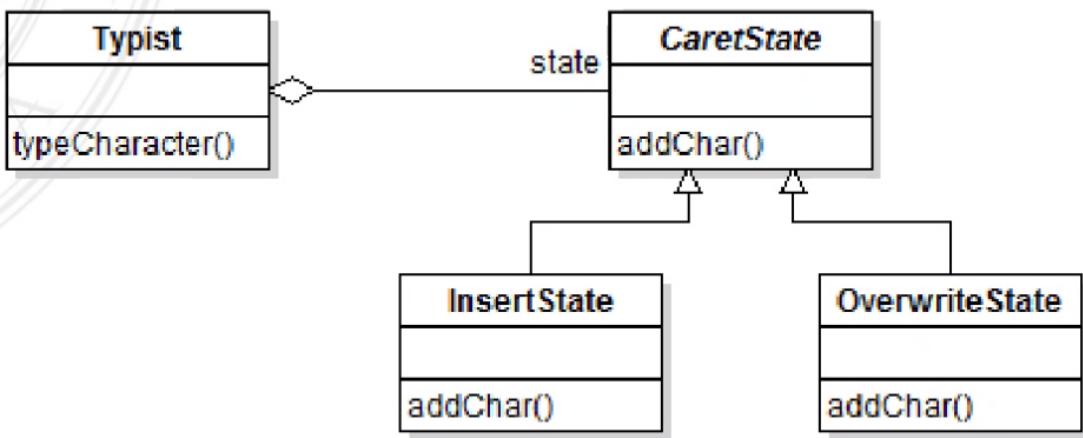
enucleare parte del comportamento ad un elemento di delega, questo ci permette di rimpiazzare il comportamento in maniera trasparente

## GoF: State

Ha la struttura identica di strategy, con delle differenze nella determinazione della classe concreta non è specificata.

A cambiare la modalità di scelta del delegato che è funzione dello stato di context.





la cosa che cambia è che la selezione del delegato dipende dallo stato dei typist

Nella classe typist c'è sempre una sola reference al delegato che può essere delle due classi.

In questo modo possiamo modificare dinamicamente la soluzione andando a modificare.

Il vantaggio è che rinchiudiamo l'oggetto in classi esterne.

e' la soluzione immediata per non scrivere comportamenti variabili all'interno della soluzione senza usare else if.

## New considerato pericoloso

New considerato pericoloso perchè potrebbe provocare un rallentamento o un perdita di efficienza

potrei creare un certo numero di oggetti allo start del programma, evitando di ricreare un nuovo oggetto per non gravare sull'efficienza e la velocità di esso.

New provoca problemi per la qualità interna del codice.

**Dependency Inversion principle:** se una classe di alto livello deve usare una di basso livello, deve andare a fare New della classe concreta che ha l'interfaccia, ma questo provoca una dipendenza diretta tra la classe di basso livello e alto livello e se la classe di basso livello cambia, deve cambiare anche quella di alto.

Quando abbiamo istanziazione diretta questa ci provoca un'istanza diretta con una classe concreta e questa cosa può essere più o meno risolta.

Le dipendenze sono critiche perché possono essere strade che diffondono il cambiamento che con un piccolo cambiamento a cascata può cambiare tutto il codice lungo tutta la linea delle dipendenze

La dipendenza peggiore è quando effettivamente si ha una dipendenza diretta con una classe e questa cambia provocandomi del cambiamento nel mio codice

Puntando ad un'interfaccia invece che ad una classe di basso livello questo ci provoca si un'istanza in più, ma evitiamo problemi di cambiamento in quanto le interfacce sono molto più stabili andando a modificare eventualmente la classe di basso livello invece dell'interfaccia.

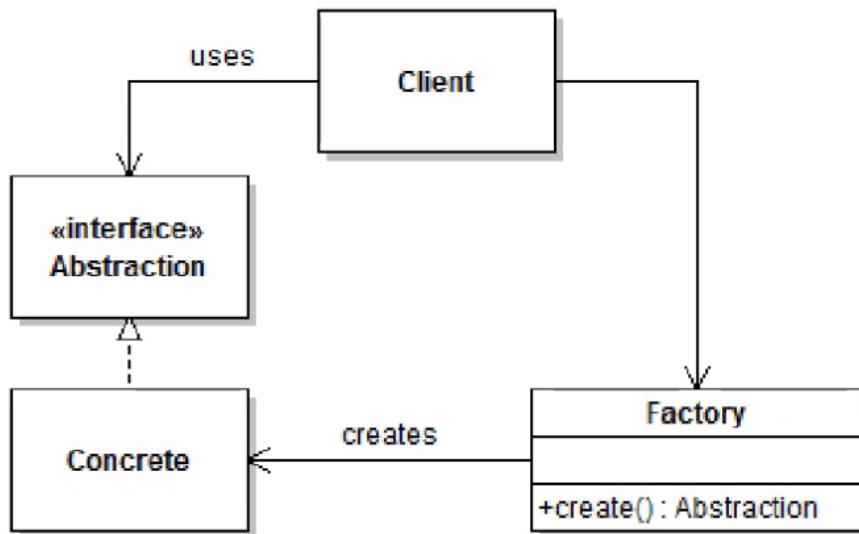
Il semplice inserimento del New per le ragioni elencate sopra può essere pericoloso.  
è inoltre facile violare OCP perchè se andiamo a modificare il costruttore dobbiamo  
modificare il codice.

Questo problema si può sistemare in diversi modi:

- factory

## Factory

La struttura è che viene proposta un'ulteriore elemento a cui noi deleghiamo l'istanziazione  
dell'oggetto



Client usa interfaccia per evitare di avere un collegamento diretto con la classe di livello basso, per evitare che un eventuale cambiamento a basso livello non ci porti anche ad un cambiamento a cascata anche alla classe di alto livello.

Noi vogliamo ottenere un oggetto di tipo interfaccia, senza dipendenza diretta. Per fare ciò delega il tutto ad una classe che si occuperà di creare un oggetto di tipo interfaccia.

è vero che Client dipende da Factory, ma invece di avere una dipendenza concreta, abbiamo una dipendenza verso una classe concreta come Factory.

Stiamo guadagnando una maggiore qualità ed efficienza a discapito però della complessità, che però aumenta, anche se ci può essere il caso in cui un uso sbagliato di questo pattern ci porta a nessun vantaggio, in quanto non andiamo a risolvere il problema che dobbiamo risolvere.

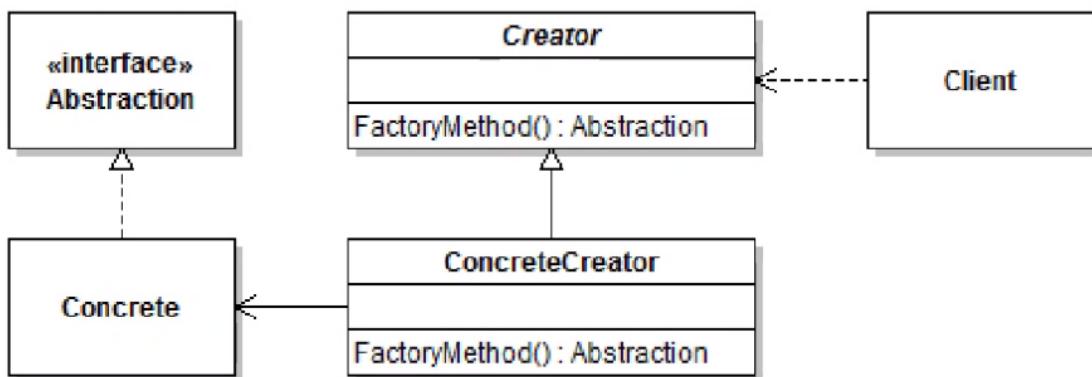
Questo funzionamento è talmente basilare da non essere considerato pattern.

Il corrispettivo pattern è Factory method

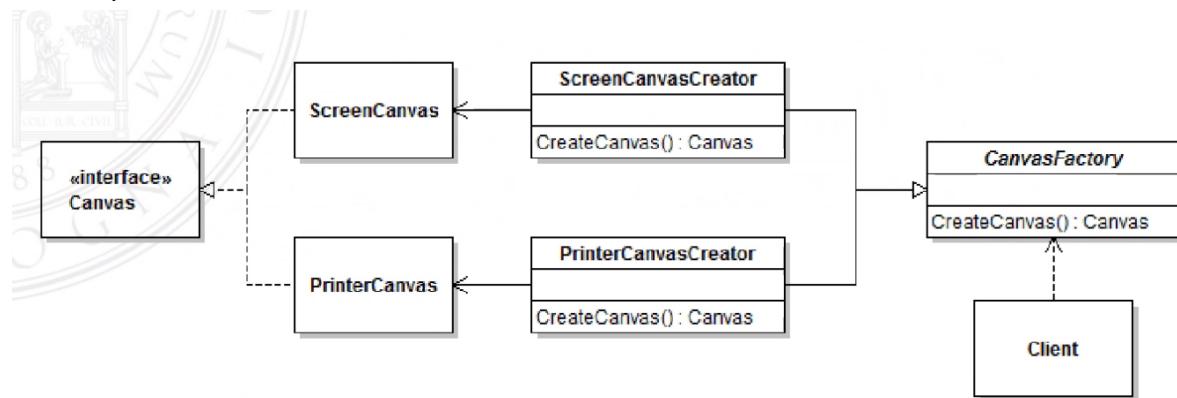
## Factory method

aggiunge un ulteriore livello alla struttura.

definisce un'interfaccia per la creazione di un oggetto, ma ciò permette alla sottoclasse di decidere quale classe istanziare



Una struttura di questo genere può essere utile quando potrei avere strategie o politiche diverse per selezionare la classe concreta



In questo caso potrebbe venirci comodo avere un *Factory method*

Altre volte invece di dove andare a lavorare su una classe diversa, andiamo ad utilizzare un particolare metodo in cui andiamo ad utilizzare un metodo statico come *Factory*.

## Esempio di codice che viola

La soluzione del codice del prf ha dei problemi soprattutto dal punto di vista dei principi in particolar modo:

Problema di troppa responsabilità della classe *Game* che oltre a gestire il campo, va anche a gestire le mosse, cosa che non dovrebbe fare.

Altri principi che sono sbagliati vengono visti, ci sono però all'interno del codice quando andiamo a fare dello stress test sul codice.

Uno di questi problemi consiste nel collegamento diretto tra classe di alto livello e classe di alto livello.

OCP è uno dei principi violato: dovevamo renderci conto che con delle modifiche ragionevoli del codice ci portano alla violazione dell'Open Closed Principle e che quindi con l'aggiunta di una variazione non devo aggiungere codice.

Per evitare di incappare nella violazione dell'OCP, cerchiamo di identificare i punti di modificabilità e ci costruiamo attorno delle astrazioni per proteggerlo.

Per fare ciò possiamo usare il meccanismo delle deleghe, prendendo un pezzo del comportamento della classe e renderlo interscambiabile (**GoF Strategy**)

Possibile anche usare Template Method spostando la parte del codice in comune in un livello superiore nella classe di ereditarietà.

Con Template Method invece di specificare un particolare metodo lo andiamo a scegliere come Template Method, facendo diventare la classe astratta.

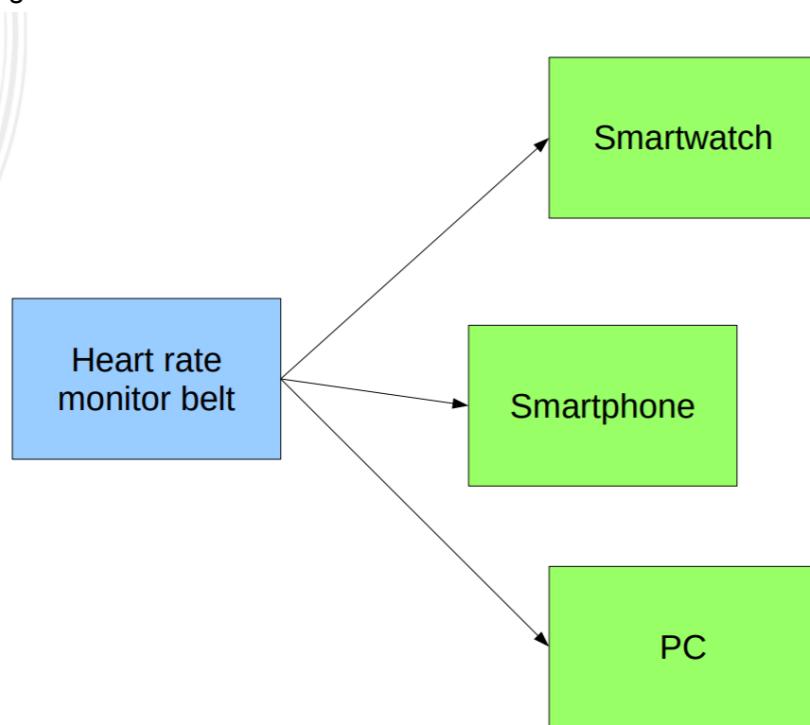
Nel planning game abbiamo due entità, una che è il cliente che vogliono funzionalità direttamente fruibili, mentre il team spinge per far funzionare meglio la parte backend che il cliente non può vedere.

Nel planning game si cerca di equilibrare queste due forze dove andiamo a chiedere una lista a livelli delle funzionalità che vuole, dall'altra parte c'è la lista del team in ordine delle cose dalle più importanti a quelle meno. Una volta fatto ciò prendiamo in maniera equilibrata dalle due liste senza cercare di sbilanciare le richieste.

Opportuno cercare di ottemperare le esigenze delle due entità.

## The notification problem

Può capitare che nella costruzione di un sistema software abbiamo un problema simile alla figura.



Vogliamo ottenere di permettere questo meccanismo di accesso allo stato attraverso un meccanismo di notifica.

Questi osservatori sono interessati a guardare i cambiamenti di stato del soggetto. Solitamente facciamo che il subject abbia tutti i riferimenti ed ad ogni cambiamento di stato mandiamo la notifica a tutti.

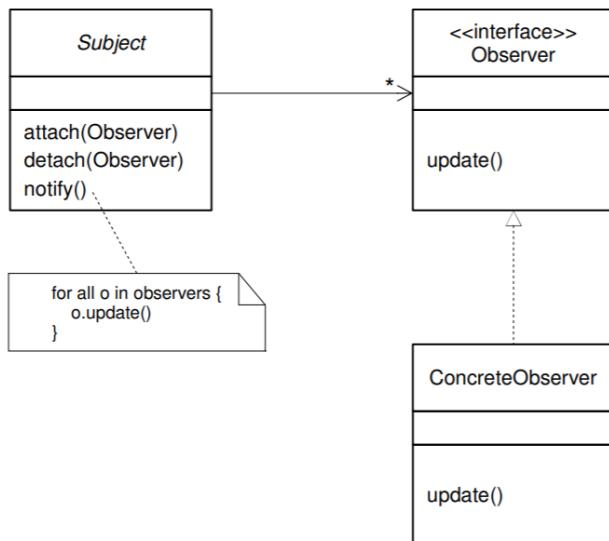
Questa soluzione però porta a due diversi problemi:

1. capire a chi notificare il cambiamento
2. problema di qualità: legato al fatto che se il soggetto dialoga direttamente con tutti gli osservatori allora diventa dipendente da esso, portandoci quindi ad altri problemi.  
Violazione di DIP(), ISP(), PV (private aggregation)

La soluzione che viene proposta è quella che ogni tipo di osservatore (anche classi completamente diverse) devono implementare una stessa interfaccia (GoF Observer)

## GoF Observer

Subject ha la lista di osservatori interessati che trova tramite delle reference e gli osservatori si possono registrare o registrare attraverso i metodi dell'interfaccia.



Questa soluzione ci porta ad eliminare le reference dirette che il subject ha al suo interno rispetto a tutti i vari controllori.

Usando questa soluzione abbiamo che non abbiamo che subject non ha reference dirette al suo interno.

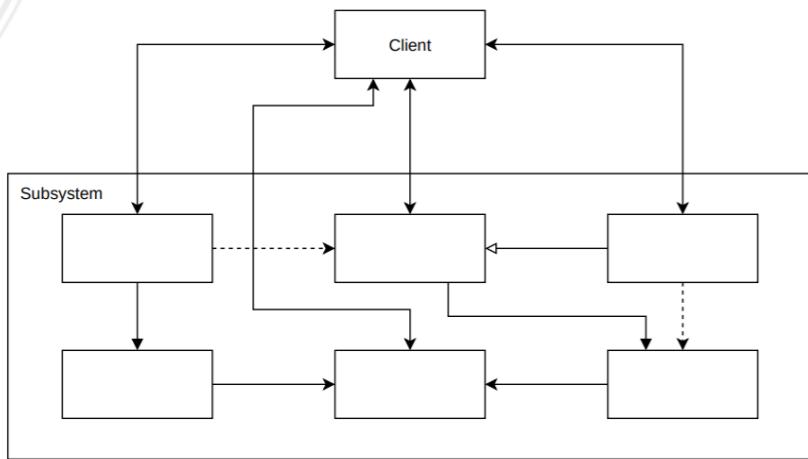
In questo caso il problema che abbiamo non è migliorare la struttura dal punto di vista qualitativo, ma dal punto di vista operativo, soprattutto per quanto riguarda la notifica agli osservatori

Osserv è talmente utile che Java stesso lo ha implementato nel pacchetto util

## GoF Facade

Pattern tipicamente strutturale.

Legato al fatto che è comune nell'organizzare le classi l'andare ad identificare dei **gruppi correlati** che si occupano di realizzare delle funzionalità che sono raggruppate per sottosistemi.

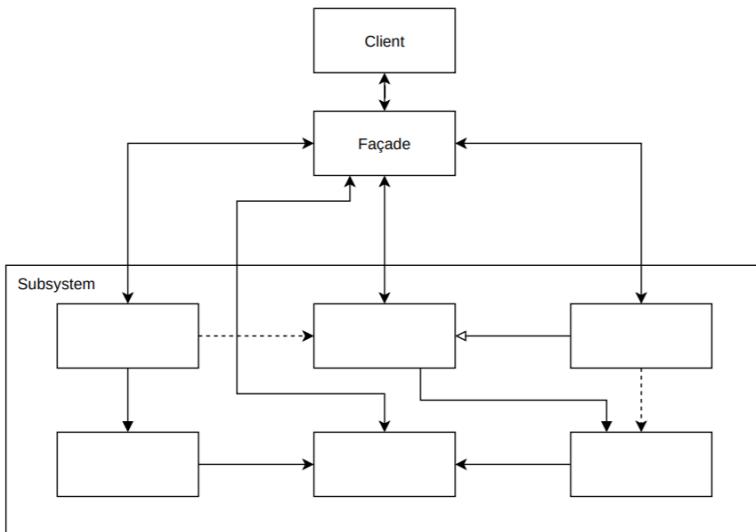


Il problema di un approccio di questo genere è che il Client ha un alto numero di dipendenze verso elementi del sottosistema esponendosi ad una eccessiva complessità

Visto che è difficile immaginare che il Client necessiti di tutte queste dipendenze possiamo anche incappare in un problema di ISP.

Per risolvere questo problema dobbiamo usare un mediatore che prende le richieste del client e le distribuisce al sottosistema.

Dal punto di vista del client, il client è come se vedesse l'intermediario come quello che fa effettivamente il lavoro avendo il collegamento diretto solo con il mediatore.



## GoF Singleton

serve a garantire che esiste un'unica istanza di una classe e che esista un meccanismo globale che mi permetta di trovare un'istanza a questa classe

Prima di usarlo dobbiamo porci delle domande, in quanto è molto comune usarlo in maniera impropria

Ci serve quando abbiamo la necessità di una sola istanza senza aver bisogno di passarla in giro.

## Singleton candidates

- **Factories**: rappresentano i singleton perché una factory può essere creata in maniera diversa rispetto ad un'altra  
Ci garantisce che ci sia una sola istanza all'interno e poi perché effettivamente non c'è ragione di dover passare la factory in giro come istanza
- **loggers**: oggetti che servono per memorizzare messaggi di log per capire cosa succede nell'applicazione  
è pratica comune che vadano a scrivere messaggi di errori mettendoli su un file o in rete.  
Se creiamo tanti e troppi log su thread diversi che cercano di aggiungere allo stesso file creando un problema di concorrenza.  
Con singleton andiamo ad eliminare il problema di creazione di troppi file di thread concorrenti
- **configuration classes**: classi che contengono parametri essenziali per il funzionamento del sistema.  
Usiamo singleton perché non vogliamo creare ogni volta un nuovo file e soprattutto non vogliamo che venga girato in giro come istanza a tutti, in quanto abbiamo un meccanismo che fa ciò.
- **resource access**
- **classi che non hanno attributi statici o associazioni che sono percorribili dalle loro istanze**

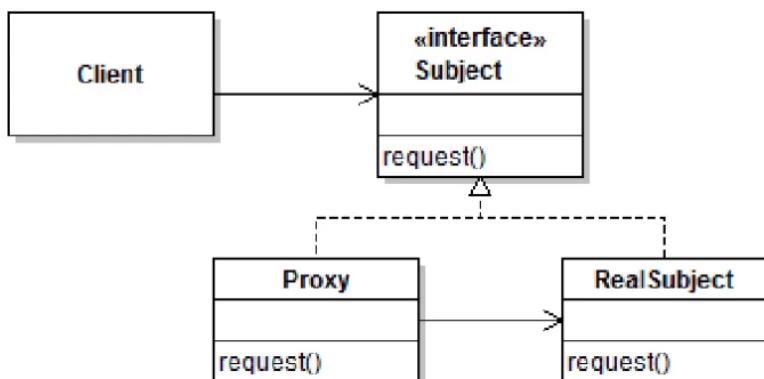
## Singleton unlikely candidates

Casi in cui il fatto che le classi per le quali una singola istanza fa parte della specifica, ma non è un problema intrinseco.

Altro problema è quando per mancanza di voglia di passare le istanze andiamo ad usare in singleton. Il problema che in un futuro potrebbero esserci dei problemi in possiamo anche a dover cambiare il funzionamento di tutto.

## GoF Proxy

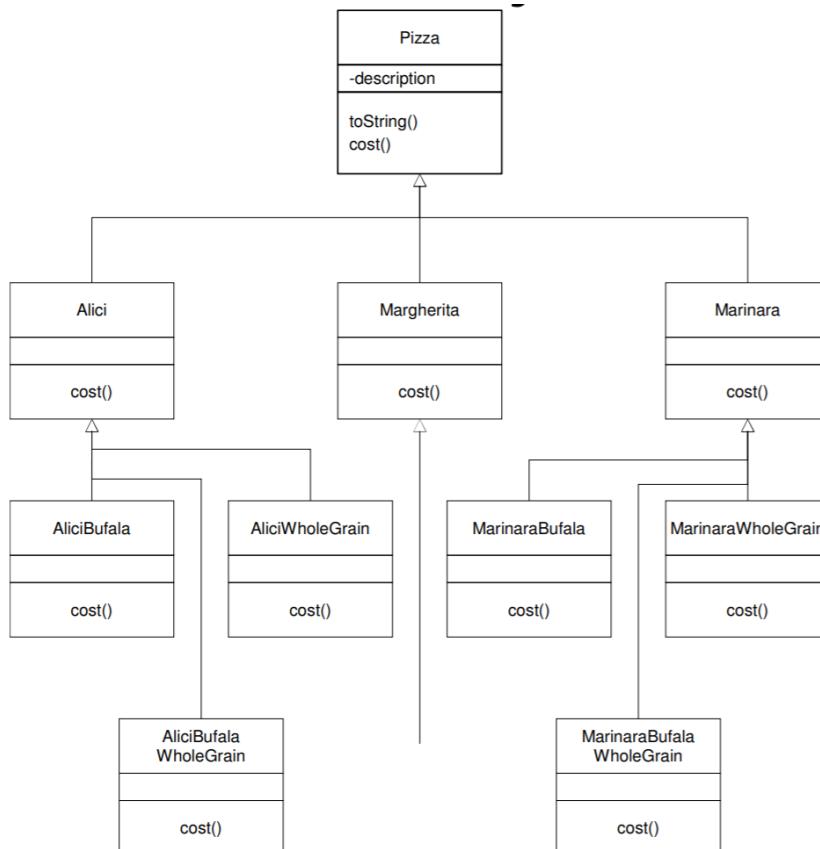
Abbiamo a che fare con un intermediario che scherma l'oggetto reale a cui ha bisogno di accedere dal client all'accesso effettivamente diretto con il client, dandogli l'impressione che effettivamente dialoga con l'oggetto reale anche se dialoga con un'interfaccia



Il proxy può aggiungere del comportamento senza aggiungere le responsabilità  
Possiamo usare per diverse ragione come:

- controllo accessi
- counter degli accessi
- log degli accessi
- accesso ad oggetti remoti (possibilmente usando un sistema di caching)
- reference intelligenti

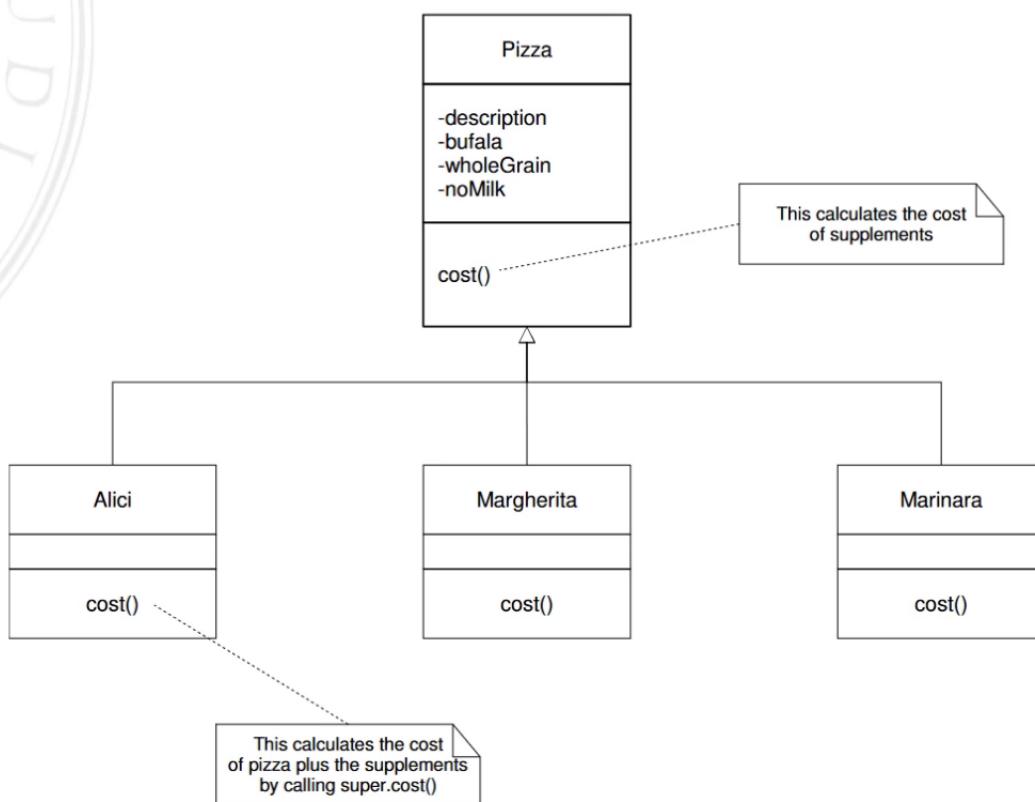
## Troppe dimensioni



in questo esempio abbiamo che una pizza marinara ha elementi e prezzo diverso rispetto alla margherita e quindi viene da pensare di poter costruire una gerarchia del genere per poter realizzare quello che è il nostro sistema di ordini.

Una soluzione del genere può avere problemi quando le pizze hanno delle versioni?

Se vado a specializzare ulteriormente la classe, abbiamo una complessità eccessiva e senza senso.



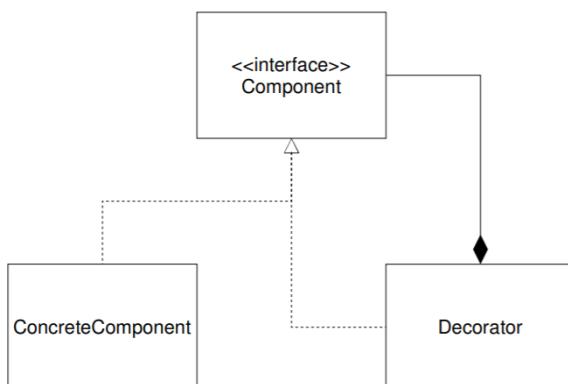
In questo caso andiamo a mettere le varie specializzazioni all'interno della classe padre **Pizza** andando a modificare il metodo andando a calcolare i supplementi che ho all'interno della **Pizza**, andando però a modificare il metodo andando a cambiare il calcolo dei supplementi.

Questa soluzione ci porta ad eliminare la complessità precedente, però non è ottimale in quanto abbiamo problemi in caso di cambiamento, come aggiunta di un nuovo supplemento perchè ridobbiamo mettere mani al codice non rispettando quindi il vincolo Open Close Principle

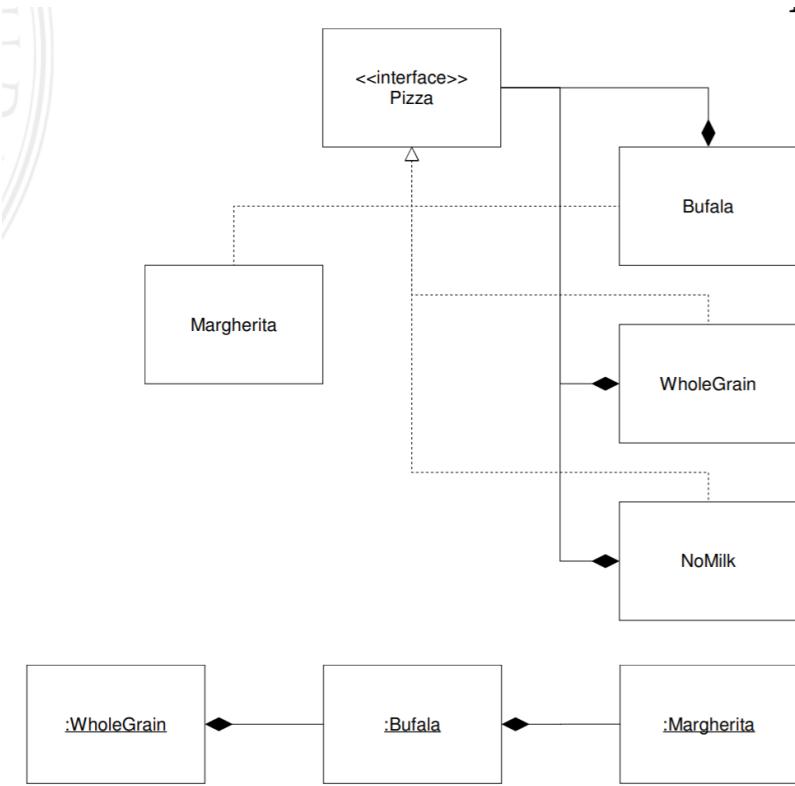
Una soluzione può esser **Decorator** che serve per aggiungere le responsabilità ad oggetto dinamico

## GoF Decorator

l'oggetto di partenza *CreateComponent* e poi andiamo a definire un decoratore che ha la stessa interfaccia dell'oggetto concreto ma che ha un'istanza dell'oggetto concreto interno ad esso



Riconsiderando il nostro esempio abbiamo che

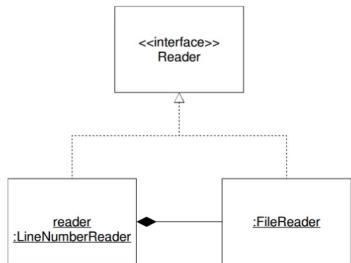


Il decorator è del livello più esterno possibile e quando dobbiamo andare a richiedere il prezzo andiamo a lavorare sul decorator che poi si ritroverà ad avere l'interfaccia dell'oggetto concreto. Così in caso di doppia richiesta di mozzarella possiamo andare a creare due oggetti e non più uno solo.

Essendo poi un meccanismo dinamico, possiamo smontare e rimontare questi oggetti come meglio vogliamo

```

java.io.Reader
Reader reader =
    new LineNumberReader(
        new FileReader("myfile"));
  
```



## GoF Adapter

A differenza di Proxy dove l'intermediario deve avere la stessa interfaccia, in questo caso l'intermediario ha un'interfaccia diversa e serve affinché il Client accedi a quella particolare classe attraverso una diversa interfaccia.

Non andiamo ad utilizzare la stessa interfaccia della classe stessa, perché in caso di cambiamento andiamo a cambiare quella che è l'interfaccia andando a violare il dependency inversion principle.

## GoF Bridge

Ha la struttura completamente uguale all'Adapter, ma quello che cambia è l'intento dietro questo pattern rispetto a quello precedente.

In questo caso vogliamo disaccoppiare l'astrazione dalla sua implementazione.

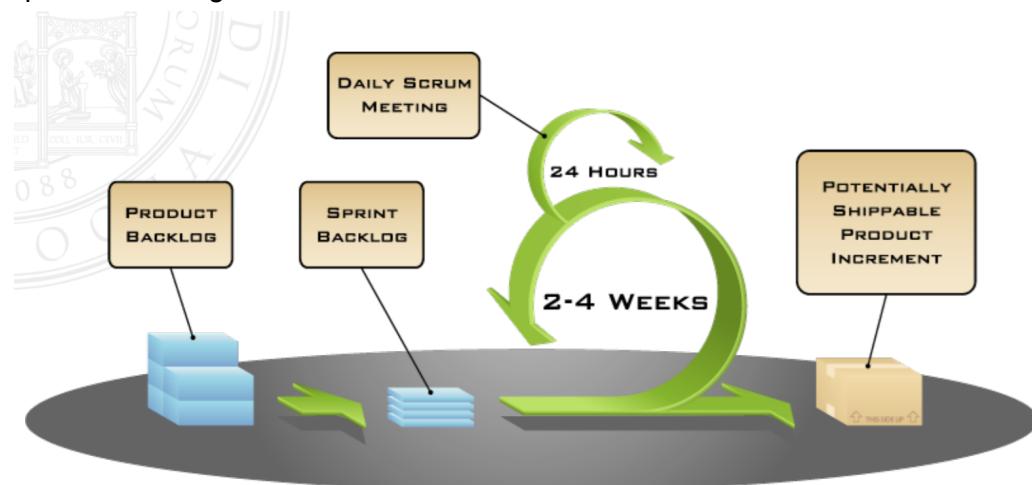
Con questo pattern andiamo a rompere la gerarchia nel quale è il cliente a decidere Adapter serve a far funzionare le cose dopo che sono state disegnate, mentre Bridge viene pensato prima ancora della creazione del modulo di basso livello.

## Scrum

framework nel quale le persone possono affrontare problemi complessi in maniera adattiva, mentre forniscono i prodotti con il più alto livello possibile attraverso produttività e creatività. Scrum fa parte del modo Agile, anche se non è una metodologia completa, focalizzandosi sugli aspetti di gestione del progetto ed è infatti comune usare Scrum con altre pratiche del mondo agile.

## Struttura di Scrum

struttura che abbiamo già visto nella parte più generale, anche se qui alcuni punti vengono specificati in miglior maniera



## Sprint

sono le interazioni che Spring possiede

Scrum tende ad essere più rigoroso e dettagliato nella gestione del progetto andando a definire ruoli, eventi ed artefatti

## Ruoli

Abbiamo i ruoli:

- **Core** (committed) sono le persone a tempo pieno sul processo
  - **product owner**: proiezione degli stakeholder all'interno del progetto  
Serve a rappresentare all'interno del progetto le richieste del cliente.  
Ha il compito di decidere le priorità, decidendo le deadline e le specifiche del progetto. Persona che è in grado, in ultima istanza, di accettare o no il lavoro che è stato svolto.
  - **scrum master**: persona che mette le proprie conoscenze a disposizione del team. Ha competenze specifiche di sviluppo software che però non è tutti i giorni a testare codice o programmare. Serve per eliminare gli impedimenti e serve come consulente per certe implementazioni.  
E' il responsabile per le corrette implementazione dei principi e delle funzionalità.

- **development team:** team di sviluppo da 5/ 9 persone che si occupa di realizzare i task (insieme di attività che servono per ogni interazione) che non devono essere chiesti dal product owner, ma devono essere autoassegnati. Team che deve essere funzionale in cui tutto il team, che deve lavorare full time sul progetto, deve avere la conoscenza del progetto senza rivolgersi a persone esterne
- **Additional (involved)**
  - customer
  - executive manager

## Artefatti

gli artefatti principali in Scrum sono:

- product backlog
- sprint backlog
- burn down chart

## Product backlog

lista ordinata il cui ordine è a carico del produttore (che rappresenta le istanze del customer) che si occupa di raffinare il product backlog.

E' la lista ordinata delle cose che devono essere fatte che possono essere requisiti funzionali, bug da sistemare, requisiti non funzionali, aspetti che riguardano le tecnologie (apparecchi particolari che il client usa ecc.) e i chores (elementi che vengono inseriti su istanze dai membri del team e che servono a produrre valore al team di sviluppo e non ai clienti, come l'aggiornare l'ambiente di sviluppo).

## Stories,epic, themes

le stories servono per descrivere features ad alto livello che solitamente sono collezionate all'inizio in maniera però generica andando a raffinarle con il progresso del progetto.

Le epics sono grandi user stories. Solitamente necessitano più di uno sprint per essere completamente sviluppate.

Temi: collezione di user stories che sono caratterizzate da coprire aspetti simili nell'applicazione

## Sprint backlog

è tutto ciò che deve essere progettato dello sprint.

Lo sprint backlog contiene dei task che si ottengono dal raffinamento degli elementi con priorità più alta all'interno del product backlog.

task che sono determinate dai membri del team e che devono essere completate nello sprint corrente

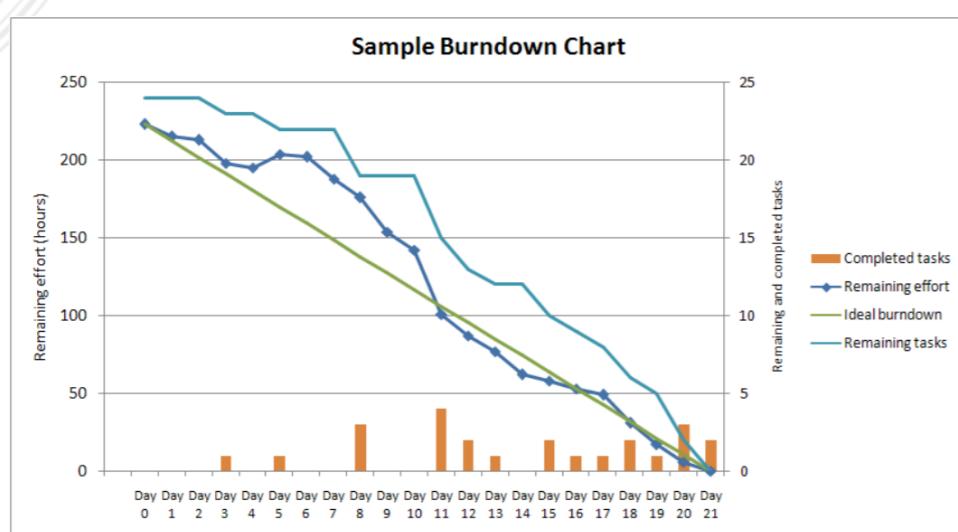
Si cerca di fare in modo che i task abbiano un tempo di durata inferiore alla giornata, e se è maggiore di una giornata lo si va a scomporre.

## Tasks

è pratica comune tenere traccia dei tasks che vengono processati.

## Burn down chart

Diagramma che fa vedere come nel tempo varino il numero di task da fare ed il monte ore disponibile



bisogna cercare di portare il nostro effort sulla retta, più ci discostiamo più risulta difficile

## Event

momenti di confronto tra membri del team o membri del team con il client sono

- sprint planning
- daily scrum
- 
- 

## Sprint planning

meccanismo per definire come riempire lo sprint backlog. Solitamente si fa una riunione che si fa in una singola giornata al massimo due e ci si organizza in due fasi:

1. product owner definisce gli obiettivi e presenta gli elementi essenziali che vuole all'interno del prodotto e per ogni elemento viene indicato gli elementi nel dettaglio e le richieste, stimando l'effort necessario
2. (Solo dal team) si selezionano gli elementi che sono rotti in tasks, popolando poi lo sprint backlog

## Scrum estimation

Quando andiamo a tirare fuori gli elementi con alta priorità, dobbiamo capire quanti di questi dobbiamo prendere

La stima dei task viene fatta con l'unità di misura comune : **le ore**

Quando andiamo a prendere elementi dentro il product backlog andiamo a valutarle in maniera diversa, attraverso le user stories.

User stories: siccome la scomposizione avviene nella seconda fase, è pratica comune valutare le user stories attraverso la complessità e gli **story points** che li utilizziamo come misura del valore prodotto.

L'assegnazione delle user stories avviene grazie al **planning poker** (ogni persona al tavolo definisce la quantità di story points che secondo lui è necessario per quella user stories e presenta la sua carta a faccia in giù, quando tutti presentano la carta viene girata e quando tutti valutano possiamo avere un punto di vista comune, se ciò non accade ci si confronta per capire quale persona ha ragione nell'indicazione della user stories, facendo poi in un nuovo ramo di votazioni, se ciò non avviene la storia può essere messa in stand-by oppure possiamo scegliere la carta dove possiamo fare una pausa in quanto stories complicata da eseguire).

A questo punto abbiamo delle stories e delle valutazioni e dobbiamo capire quante ne dobbiamo prendere per completare lo sprint backlog possiamo ragionare in diversi modi:

- **capacity driven planning**: per team che hanno poco chiaro la velocity o che non sono molto legati tra loro.  
Invece di caratterizzare le stories sotto forma di story point le stories vengono analizzate sotto forma di effort portandoci ad una duplice analisi.
- **velocity driven planning**: non ragioniamo sotto forma di effort, ma lavoriamo sul livello di complessità delle storie che mette insieme anche una valutazione del lavoro che dell'effort.  
per sapere quanti story points possiamo elaborare tenendo conto che non sono solo associate all'effort, ma anche della complessità, attraverso l'esperienza.  
Dimostra di essere una stima ottima con i team che conoscono la loro velocity (metrica di avanzamento che ci indica quanti story point macina il team di sviluppo). Team diversi macinano diversi story points, che però possono dare stime corrette ed azzeccate

## Daily Scrum

riunione informale, abbastanza breve la cui gestione è in mano allo scrum master e durante questa riunione diversi membri devono rispondere a diverse domande:

- cosa ho fatto ieri che ha aiutato il team raggiungere l'obiettivo
- cosa ho fatto oggi per aiutare il team a raggiungere l'obiettivo
- C'è qualche impedimento che ci impedisce di raggiungere l'obiettivo

## Sprint review

evento che vede coinvolto tutti i membri del team e gli stakeholder dove viene mostrato ciò che si sta facendo discutendo dei problemi e di eventuali soluzioni e discutendo anche delle tempistiche

## Retrospecting

parte dello Scrum master e development team dove si discutono delle problematiche riscontrate durante lo sviluppo cercando di identificare miglioramenti prima del prossimo sprint.

possono essere discusse cose anche diverse tra di loro parlando anche dell'uso di diverse pratiche o di aspetti tecnologici

## Scaling

Lo scaling può essere ottenuto tramite gerarchia nello scrum del team  
Eventi specifici sono programmati per garantire il processo generale

## Scrum considerato pericoloso/dannoso

In caso di nessuna presenza dello scrum master (persona qualificata che però non scrive riga di codice) oppure lo svolge un membro del team che programma o più membri del team a rotazione, questo però potrebbe anche essere un vantaggio che però deve essere dimostrato praticamente.

## Kanban

Kanban è un metodo di pianificazione snello per controllare a filiera la produzione just in time.

Introdotto nel dominio dello sviluppo software di David Anderson. Open Kanban è un open versione sorgente di Kanban per lo sviluppo di software che comprendono lo sviluppo Agile. Non esiste il concetto di ciclo come per le "call" che si fanno con Agile ogni tot periodo.

Abbiamo un flusso continuo e quindi abbiamo una pipeline che è composta di diversi passi. L'idea è che gli artefatti vengano raffinati in questi stadi per fornire una versione sempre più aggiornata dell'applicazione.

La lavagna di Kanban è l'artefatto principale dell'applicazione.

Una caratteristica importante è che per evitare colli di bottiglia, prevede che i diversi stadi abbiano un limite nel work in progress.

Se una pipeline è piena non possiamo prendere un elemento dallo stadio precedente se non svuotiamo la pipeline.

Gli stadi vengono divisi in quelli da fare e quelli già fatto.

La metrica di misura in questo caso è il cycle time (tempo medio per completare tutta la pipeline di quel particolare elemento)

Il WIP serve a capire il numero di elementi in quel particolare stadio.

Workflow =>	Inbox	Specification		Ready for Development	Development (e.g. using Scrum and XP)			Code Review		Test on Local System		Test on Pre-Production System	
WIP Limit =>	5	2		2	3			2		2		2	
Feature		In progress	Done		Planned	In Progress	Done	In progress	Done	In progress	Done	In progress	Done
Login	User Story 567 User Story 214		User Story 857				User Story 654			User Story 75			
Register				User Story 244		User Story 751							
Password Recovery	User Story 624				User Story 245			User Story 782					
...	...	...	...	...	...	...	...	...	...	...	...	...	...
Billing			User Story 657	User Story 38				User Story 858					

Con *done* indichiamo quegli stadi che presentano elementi che possono andare alla fase successiva, se la fase successiva è piena allora rimane in stato di *done* fino a che non si svuota la fase successiva.

Se qualcosa non funziona è sbagliato l'allocamento delle risorse su quella pipeline.

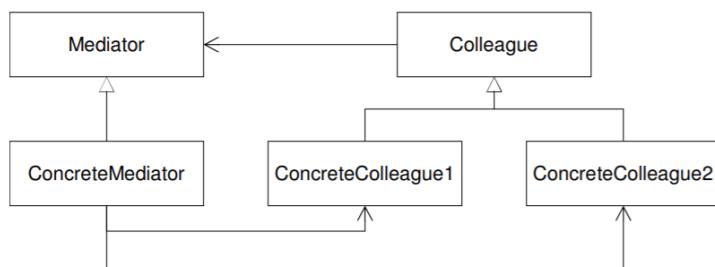
## GoF Mediator

Definisce come un oggetto encapsulato interagisce con un set di oggetti  
Pattern che si propone di definire una soluzione ad un problema che emerge quando  
all'interno abbiamo identificato una serie di oggetti che devono interagire tra di loro per  
funzionare.

Ciascun oggetto deve essere funzionale e deve dialogare con altri oggetti che ci porta ad  
una rete molto fitta di dipendenze.

La soluzione consiste nell'uso di un intermediario in modo che il dialogo tra questi oggetti  
avvenga tramite intermediazione.

Si cerca di fare in modo che i diversi oggetti siano una sottoclasse di una classe comune o  
interfacce comuni.



Nel caso in cui ci dovesse essere un nuovo oggetto, in questo caso, dialoga direttamente  
con il mediatore che si occupa poi di smistarla al destinatario.

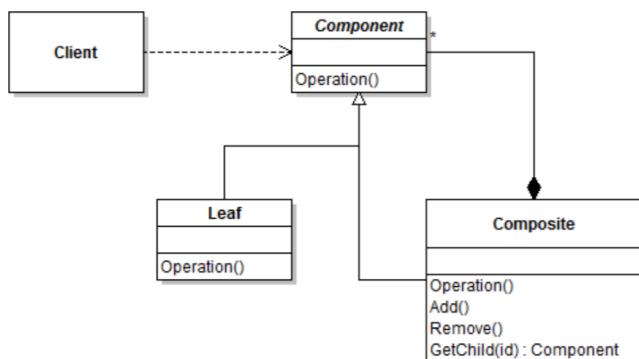
Un approccio di questo genere non è sempre utilizzato.

## GoF Composite

Pattern in cui andiamo a utilizzare strutture ad albero in cui andiamo a comporre tra di loro  
collezioni di oggetti in cui i diversi elementi svolgono due ruoli distinti: ruolo intermedio che  
rimanda ad altri elementi e ruolo terminale.

Ci suggerisce di seguire questa struttura in modo che ogni elemento (nodo) sia *Component*  
e facciamo in modo che ogni elemento che sia una *Leaf* sia una *Composite*.

la differenza tra *Leaf* e *Composite* è che *Leaf* non rimanda ad altri elementi, mentre con  
*Composite* sì.



Con strutture di questo genere è buona norma avere la stessa interfaccia e specializzarla  
poi successivamente

## GoF Memento

Serve a risolvere quei casi in cui ci serve ottenere la rappresentazione dello stato di un oggetto.

Il caso classico è la funzionalità salva.

In cui dobbiamo andare a prendere lo stato, esternalizzarlo per poterlo salvare, in modo che in caso di riapertura del documento ci ritorni quel particolare stato.

La soluzione potrebbe essere chiedere tutti gli elementi dello stato.

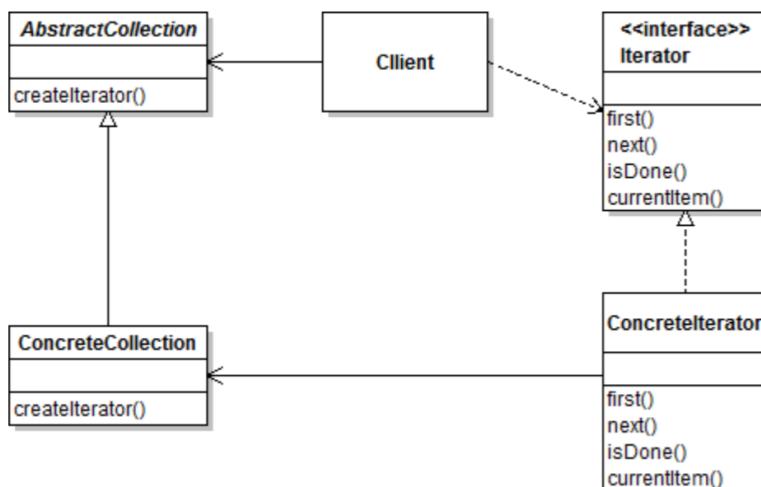
Il problema di un approccio di questo genere consiste nel fatto che potrebbe violare l'incapsulamento.

Memento richiede ad ognuno degli oggetti una versione serializzata del proprio stato sotto forma di serie opaca (in modo che non sia leggibile).

Gli oggetti coinvolti devono avere la funzionalità di estrarre in Stringa e un'altra funzionalità in cui andiamo a “convertire” la stringa nel linguaggio comprensibile alla macchina.

## Gof Iterator

fornisce un meccanismo di accesso ad una serie aggregata di oggetti (collezione) senza che si esponga i dettagli su come internamente la collezione è organizzata.



Chiede alla collezione di fornire un iteratore, dando quindi responsabilità all'aggregato di fornire un oggetto iteratore, che avrà implementazione diversa a seconda cosa debba fare all'interno del nostro programma.

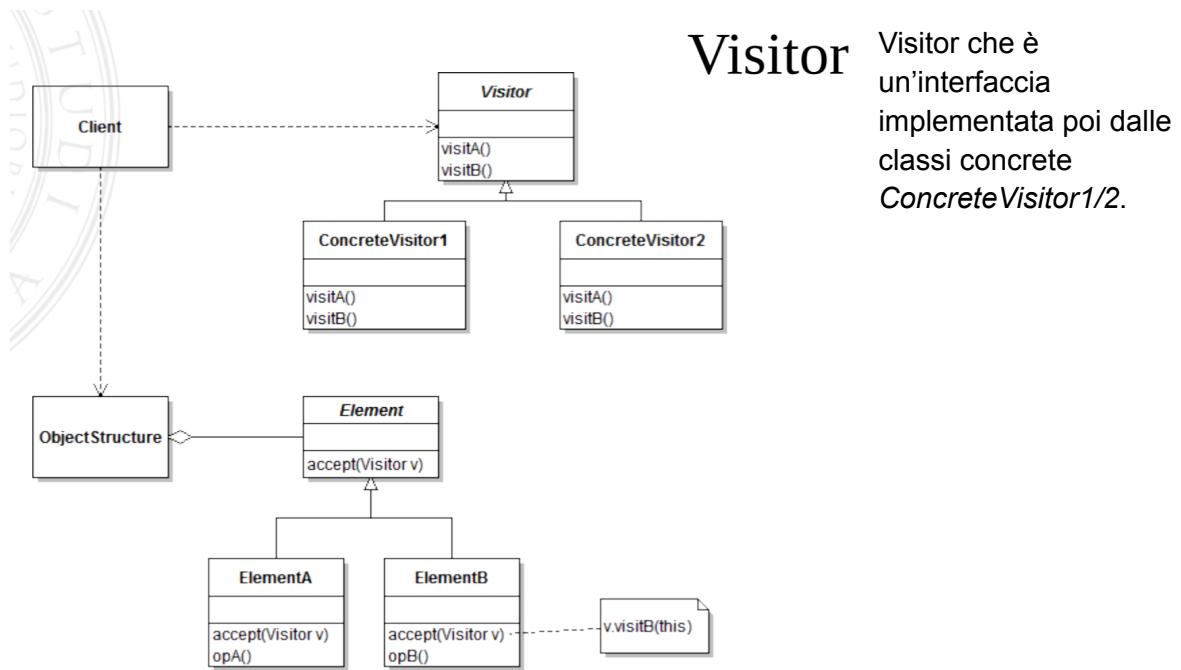
## GoF Visitor

Utilizzato per realizzare funzionalità simili invertendo il controllo.

Invece di dire alla collezione di dare l'iteratore e poi di lavorarci sopra, andiamo ad applicare a tutti gli oggetti questo particolare metodo.

Passiamo questa istanza alla collezione e poi sarà lei ad adeguarsi fornendolo a tutti gli oggetti.

Essendo la collezione a prende in mano il controllo, andiamo a dire che questo usi il **Pattern Inversion of control**.



## Visitor

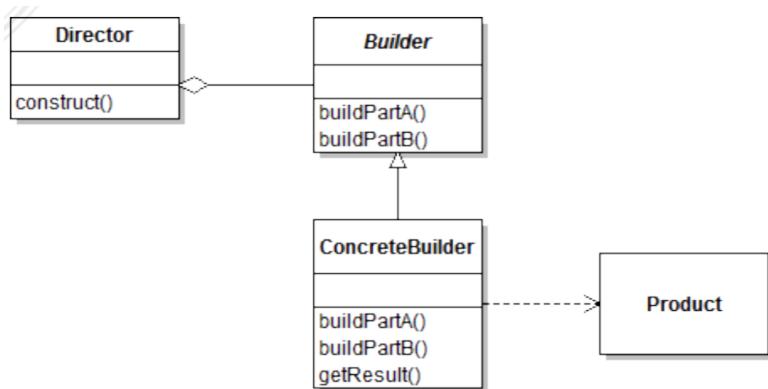
Visitor che è un'interfaccia implementata poi dalle classi concrete *ConcreteVisitor1/2*.

Con Visitor invece che fare `has.Next` come Iterator, dobbiamo indicare i metodi che la collezione deve usare su tutti gli elementi interni alla collezione

## GoF Builder

Pattern che viene utilizzato nei casi di istanziazione di oggetti particolarmente complicati. Quando abbiamo strutture di questo genere può essere complicato il processo di istanziazione.

Dobbiamo identificare il Director che delega la costruzione parti della struttura dell'oggetto a diversi *Builder* (possono essere interfacce).



La cosa rilevante è il fatto di identificare i *Builder* tramite interfaccia comune. L'idea è che tutti i *ConcreteBuilder* siano di tipi *Builder*.

Minore, ma frequente problema che *Builder* risolve: ugly constructors

Altro uso di *Builder*, molto più frequente, è che dobbiamo usare *Builder* per quei linguaggi che non hanno modo di specificare nomi per i parametri che non hanno valori (null).

Con il *null*, deve capire che non deve inserire niente perché non deve essere inizializzata oppure perché non sappiamo cosa mettere al momento dell'istanzazione.

Una soluzione consiste nell'uso di Builder partendo da un oggetto Builder che poi ha un *FactoryMethod* interno.

Abbiamo vantaggi dal punto di vista che: sappiamo cosa siano i parametri in quanto abbiamo dei metodi precisi che ci dicono. Altro vantaggio consiste nel fatto che non dobbiamo per forza inizializzare tutti i parametri.

Con questa soluzione abbiamo una migliore leggibilità risolvendo anche i problemi del *null*.

## GoF Command

Assomiglia molto a Visitor, in quanto andiamo a specificare un oggetto tramite metodo aspettandoci però che questo metodo lo attivi.

Command deriva proprio dal fatto che assomiglia ad un vera e propria attivazione dell'oggetto.

Chi riceve l'oggetto ,che ha l'interfaccia comune interno a sé, va a richiamare il metodo che andrà poi ad attivare l'oggetto.

## GoF Abstract Factory

Rende più astratto nella struttura di Factory.

Non solo abbiamo l'interfaccia Factory che possiamo poi concretizzarla.

Generalizzazione pensata per famiglie di prodotti e non solo con un prodotto

## GoF Prototype

Idea che nuove istanze siano create copiando quelle esistenti.

Abbiamo nuove istanze che sono dei veri e propri cloni di istanze già esistenti.

Questo è utile quando abbiamo una nuova istanza che dobbiamo andare a riempire, che però ci richiede di "rompere" il concetto di encapsulamento, cosa che non dobbiamo fare e cosa che Prototype va ad evitare.

## GoF Flyweight

Pattern che ci risolve un problema di generazione di numerose quantità di istanze di oggetti che condividono alcuni aspetti andando ad ottimizzare la quantità di memoria occupata.

Andiamo a prendere gli elementi di stato, li esternalizziamo, in modo da renderli disponibili a tutti.

## GoF Chain of Responsibility

serve a definire il comportamento in relazione all'esecuzione di un'operazione che può essere eseguita da diversi elementi all'interno della soluzione.

L'idea è che abbiamo una catena di oggetti che può risolvere questa soluzione andando a controllare dal primo elemento fino all'ultimo chiedendo chi può eseguire e chi no il problema.

## GoF Interpreter

Pattern che serve a dire che in certi casi serve a dire che il comportamento della nostra soluzione sia configurabile arrivando a dire che il linguaggio interno è personalizzabile

# Modern Pattern

## Null Object

serve a definire il problema delle reference non associata a delle istanze che nei vari linguaggi di programmazione corrisponde ai vari *null*.

Se una reference non è associata, in qualche maniera otteniamo errore runtime (*NullPointerException*).

Deve capire se il null è perché non dobbiamo specificare valore o perché non sappiamo cosa mettere? Questa ambiguità è un problema ricorrente dovuta ad una cattiva interpretazione dell'oggetto null al punto tale da avere costrutti specifici che ci evitano questo problema.

L'idea di questo pattern è che quando vogliamo specificare che un parametro non è inizializzato, torna un oggetto convenzionale al posto di null per far capire che l'oggetto non è istanziato.

Il vantaggio di usare questa istanza convenzionale è che possiamo specificare questo oggetto implementando delle funzionalità di default per vari metodi.

```
public interface Cat {  
    public static final Cat NONE =  
        new Cat() {  
            //“do-noting” methods  
            Cat getParent() {  
                return NONE;  
            }  
        };  
    ...  
    Cat grandParent =  
        cat.getParent().getParent();  
}
```

se avessimo usato null avremmo avuto un'eccezione in quanto non possiamo fornire null, mentre con NONE questo tipo di problema lo andiamo ad evitare.

Il vantaggio è che oltre ad avere un valore convenzionale che non ci genera eccezioni sopra.

## Hollywood principle

Per parlare di questo pattern dobbiamo introdurre l'IoC (inversion of control).

Nome che deriva dalla comune domanda che si fanno durante i provini.

**loc:** Sarà un qualcuno ad attivare un qualcosa invece che attivarlo noi stessi.

**Libreria:** insieme di classi che dobbiamo usare dove abbiamo bisogno

**Framework:** insieme di classi a cui dobbiamo passargli il controllo specificando i metodi che il framework dovrà attivare in particolari momenti (fa uso dell'inversion of control).

## Dependency injection

Quando mandiamo in esecuzione il nostro codice, vengono istanziati diversi oggetti che a catena ha bisogno di altre funzionalità creando così la nostra rete di dipendenze allocate in memoria.

Questo è il grafo degli oggetti della nostra applicazione.

Questo grafo viene prodotto dinamicamente in funzione della necessità e funzionalità e la determinazione delle diverse classi è a carico degli elementi che hanno bisogno di dialogare con queste classi.

Dependency injection usa IoC (inversion of control) per creare questo grafo andando a togliere l'istanziazione degli oggetti dalla logica, spingendola verso l'esterno (*Composition*). Significa che quando voglio operare su degli oggetti non deve essere più mia responsabilità trovare gli oggetti con cui devo dialogare andando a sfruttare il framework che mi crea la reference.

Da un lato abbiamo visto che il *new* crea dei problemi e andiamo ad utilizzare delle factory, che comunque possono essere migliorate in quanto andiamo a chiedere a noi alla factory cosa istanziare.

Il risultato dell'applicazione di questo pattern serve per migliorare due aspetti fondamentali del codice:

- semplificare tanto il **testing**: Dependency injection usato principalmente per questo motivo
- di conseguenza migliora aspetti interni del codice come **modularità**

Quando richiedo una reference il framework me la deve fornire già istanziata ed in modo corretto.

Il modo migliore è di usare il ***Constructor injection***: dipendenze fornite dai costruttori dei parametri, in quanto dipendenze costruite all'interno dei costruttori dei parametri.

Nel ***Setter injection*** o ***field injection*** andiamo a creare dei setter in cui andiamo a creare dei metodi apposta per settare delle reference in modo che sappiamo che sia corretta l'istanziazione.

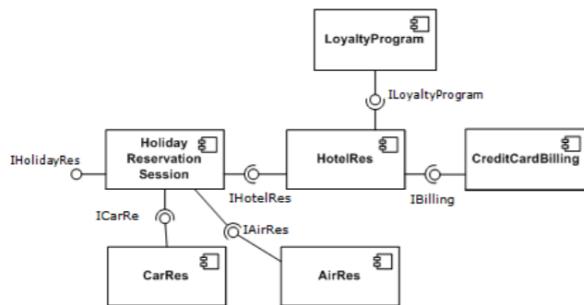
***Interface injection***: reference che viene specificata all'interno delle interfacce.

***Method injection***: reference che viene specificata all'interno dei diversi metodi. Viene attivata quando viene richiamato un certo metodo.

## Component based architectures

Nell'ingegneria dei software component-base i sistemi software sono costruiti unendo insieme componenti software sulla base di interfacce fornite e richieste.

Idea di progettazione attraverso meccanismo dove ogni componente fornisce interfacce e richiede a sua volta interfacce.



## Clean Dependency Injection

si può utilizzare attraverso meccanismi che sfruttano IoC oppure può essere realizzato con pure injection control.

Sono delle vere e proprie linee guida.

E' preferibile constructor injection in quanto andiamo a specificare dentro il costruttore delle dipendenze che abbiamo bisogno.

Il caso più problematico che con Dipendancy injection è quello per il quale dobbiamo risolvere runtime le dipendenze, in quanto a secondo dello stato in cui siamo può essere che la classe concreta può essere scelta in maniera differente.

Esempio classico può essere il gioco dello scacchi dove quando andiamo a delegare la mossa ad un'ulteriore classe, però potremmo avere però il problema in cui con un diverso livello di difficoltà, abbiamo un problema nel fatto che potremmo avere diversi tipi di difficoltà. In questo caso abbiamo un dipendenza verso l'interfaccia, ma le modifiche che implementano concretamente la classe viene effettuata specificano l'interfaccia. Facendo ciò però abbiamo il problema in cui a seconda dello stato in cui siamo possiamo avere diverse mosse che ci portano a diverse caratteristiche.

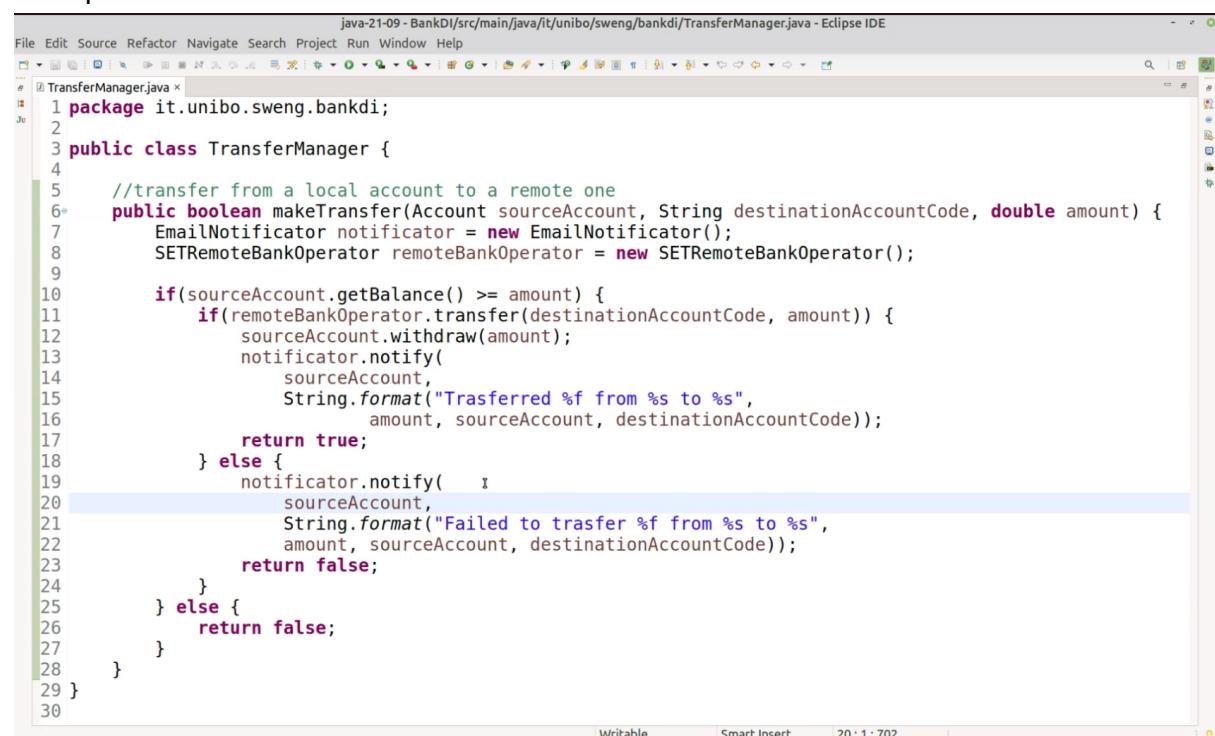
Quando l'associazione deve essere risolta al passaggio in fase di esecuzione di costrutti o metodi abbiamo che:

- cercare di isolare i punti di scelta in strutture strategy-like
- se il linguaggio lo permette la dipendenza si discosta dalla signature della factory.

Chi costruisce l'oggetto deve anche passare al costruttore le istanze collegate ad esso

Ci sono alcuni framework per java come Spring o Guice

Esempio:



The screenshot shows the Eclipse IDE interface with the TransferManager.java file open. The code implements a TransferManager class that performs a transfer between a local account and a remote one. It uses dependency injection to obtain instances of EmailNotificator and SETRemoteBankOperator. The code checks if the source account has enough balance, performs the transfer, and then notifies the notificator with a success or failure message. The code is annotated with comments explaining its purpose.

```
java-21-09 - BankDI/src/main/java/it/unibo/sweng/bankdi/TransferManager.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
TransferManager.java x
1 package it.unibo.sweng.bankdi;
2
3 public class TransferManager {
4
5     //transfer from a local account to a remote one
6     public boolean makeTransfer(Account sourceAccount, String destinationAccountCode, double amount) {
7         EmailNotificator notificator = new EmailNotificator();
8         SETRemoteBankOperator remoteBankOperator = new SETRemoteBankOperator();
9
10        if(sourceAccount.getBalance() >= amount) {
11            if(remoteBankOperator.transfer(destinationAccountCode, amount)) {
12                sourceAccount.withdraw(amount);
13                notificator.notify(
14                    sourceAccount,
15                    String.format("Trasferred %f from %s to %s",
16                                   amount, sourceAccount, destinationAccountCode));
17                return true;
18            } else {
19                notificator.notify(
20                    sourceAccount,
21                    String.format("Failed to trasfer %f from %s to %s",
22                                   amount, sourceAccount, destinationAccountCode));
23                return false;
24            }
25        } else {
26            return false;
27        }
28    }
29}
30
```

In questo caso stiamo violando il principio Solid di **DIP** (Dependency inversion principle), perché ho una dipendenza diretta tra una classe di basso livello ed una di alto livello.

Le classi di alto livello non devono dipendere da quelle di basso livello.

Il problema è che se dovessimo fare per tutto il nostro programma abbiamo tante parecchie dirette che in caso di cambiamento dobbiamo fare cambiamenti a catena che sono inutili.

Per evitare questo problema cerchiamo di implementare l'interfaccia per evitare il collegamento diretto usandola come mediatore tra le due classi.

```

1 package it.unibo.sweng.bankdi;
2
3 public class SETRemoteBankOperator implements RemoteBankOperator {
4
5     //perform a transaction with a remote bank using the SET protocol
6     @Override
7     public boolean transfer(String destinationAccountCode, double amount) {
8         System.out.println("REALLY REALLY transferring "+amount+" to "+destinationAc
9         return true;
10    }
11 }
12

```

```

TransferManager.java x EmailNotificator.java Main.java SETRemoteBankOperator.java
1 package it.unibo.sweng.bankdi;
2
3 public class TransferManager {
4
5     //transfer from a local account to a remote one
6     public boolean makeTransfer(Account sourceAccount, String destinationAccountCode
7         Notifier notificator = new EmailNotificator();
8         RemoteBankOperator remoteBankOperator = new SETRemoteBankOperator();
9
10    if(sourceAccount.getBalance() >= amount) {
11        if(remoteBankOperator.transfer(destinationAccountCode, amount)) {
12            sourceAccount.withdraw(amount);
13            notificator.notify(
14                sourceAccount,
15                String.format("Trasferred %f from %s to %s",
16                                amount, sourceAccount, destinationAccountCode));
17            return true;
18        } else {
19            notificator.notify(
20                sourceAccount

```

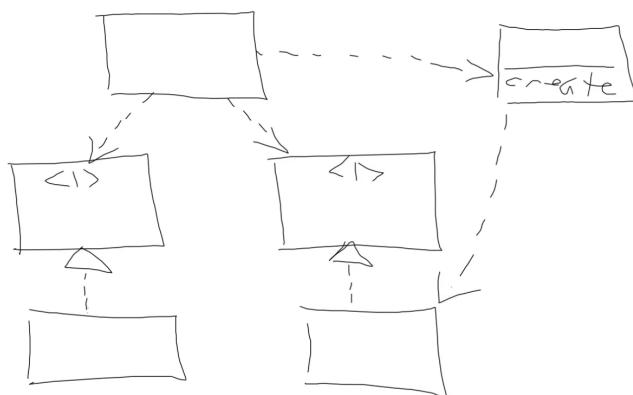
ma siccome usiamo new, che è dannoso dobbiamo staccare la dipendenza dal livello.

In questo caso possiamo fare:

1. factory e dentro la factory andiamo a fare il new lì dentro in modo da avere la dipendenza diretta dentro la factory.

Andiamo a costruire un factory method, in modo che la dipendenza sia verso la factory che ci ritorna indietro i tipi di interfacce, *Notifier*.

Abbiamo quindi che la factory dipenderà dalla classe concreta

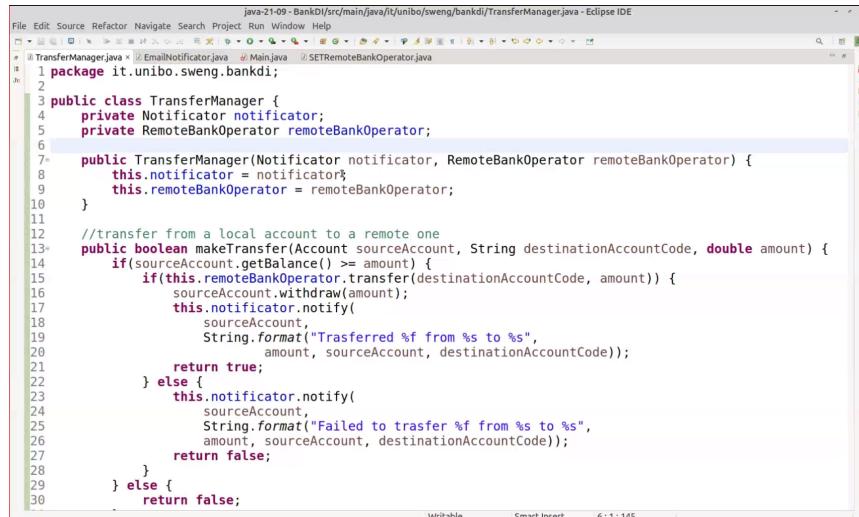


L'unica dipendenza brutta ora è il collegamento tra le factory e la classe di basso livello, invece che tra la classe di alto con quella di basso livello.

*Notifier* e *RemoteBankOperator* sono le sue variabili d'istanza e quindi hanno bisogno di essere private in modo da poter essere usate solo dalla classe.

Per rendere le classi più testabili, molti sviluppati vanno anche a migliorare la qualità del codice.

Per fare ciò dobbiamo usare delle reference dobbiamo fare uso del costruttore in modo che dobbiamo far indicare quali sono le dipendenze che necessita.



```
java-21-09 - BankDI/src/main/java/it/unibo/sweng/bankdi/TransferManager.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
TransferManager.java EmailNotificator.java Main.java SETRemoteBankOperator.java
1 package it.unibo.sweng.bankdi;
2
3 public class TransferManager {
4     private Notifier notificator;
5     private RemoteBankOperator remoteBankOperator;
6
7     public TransferManager(Notifier notificator, RemoteBankOperator remoteBankOperator) {
8         this.notificator = notificator;
9         this.remoteBankOperator = remoteBankOperator;
10    }
11
12    //transfer from a local account to a remote one
13    public boolean makeTransfer(Account sourceAccount, String destinationAccountCode, double amount) {
14        if(sourceAccount.getBalance() >= amount) {
15            if(this.remoteBankOperator.transfer(destinationAccountCode, amount)) {
16                sourceAccount.withdraw(amount);
17                this.notificator.notify(
18                    sourceAccount,
19                    String.format("Trasferred %f from %s to %s",
20                                  amount, sourceAccount, destinationAccountCode));
21                return true;
22            } else {
23                this.notificator.notify(
24                    sourceAccount,
25                    String.format("Failed to trasfer %f from %s to %s",
26                                  amount, sourceAccount, destinationAccountCode));
27                return false;
28            }
29        } else {
30            return false;
31        }
32    }
33}
```

Andiamo a costruire il costruttore in cui andiamo ad inserire le dipendenze di cui necessita andando a fare il `this.` per poterlo usare.

In questo caso nel main (root) andiamo a fare ciò di cui ha bisogno il programma.

```
public class Main {
    public static void main(String[] args) {
        //create a transfer manager
        TransferManager transferManager = new TransferManager(
            new EmailNotificator(),
            new SETRemoteBankOperator());
        //create a test account
        Account account = Account.AccountBuilder.createBuilder().
            setOwner("Superpippo").
            setEmail("super@pippo.com").
            setPhoneNumber("12345678").
            setBalance(100).
            build();
        //make a transfer
        transferManager.makeTransfer(account, "ITXYZ42", 50);
    }
}
```

All'interno del main andiamo a istanziare gli oggetti che ci servono.

Per semplificare il tutto possiamo usare framework di Dependency injection.

Andiamo ad usare l'*injector* andando ad usare l'interfaccia implementata dentro Java chiamata *Absolute Method*.

```
1 package it.unibo.sweng.bankdi;
2
3 import com.google.inject.Guice;
4 import com.google.inject.Injector;
5
6 public class Main {
7
8     public static void main(String[] args) {
9         Injector injector = Guice.createInjector(new BankModule());
10
11     //create a transfer manager
12     TransferManager transferManager = injector.getInstance(TransferManager.class);
13
14     //create a test account
15     Account account = Account.AccountBuilder.createBuilder().
16         setOwner("Superpippo").
17         setEmail("super@pippo.com").
18         setPhoneNumber("12345678").
19         setBalance(100).
20         build();
21
22     //make a transfer
23     transferManager.makeTransfer(account, "ITXYZ42", 50);
24 }
25 }
```

Questo spara tutta la classe come dipendenza senza usare il `new`. Ad ogni dipendenza associa una classe concreta.

```

java-21-09 - BankDI/src/main/java/it/unibo/sweng/bankdi/TransferManager.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer TransferManager.java EmailNotifier.java Main.java SETRemoteBankOperator.java BankModule.java
1 package it.unibo.sweng.bankdi;
2
3 import com.google.inject.Inject;
4
5 public class TransferManager {
6     private Notifier notificator;
7     private RemoteBankOperator remoteBankOperator;
8
9     @Inject
10    public TransferManager(Notifier notificator, RemoteBankOperator remoteBankOpe
11        this.notificator = notificator;
12        this.remoteBankOperator = remoteBankOperator;
13    }
14
15    //transfer from a local account to a remote one
16    public boolean makeTransfer(Account sourceAccount, String destinationAccountCode
17        if(sourceAccount.getBalance() >= amount) {
18            if(this.remoteBankOperator.transfer(destinationAccountCode, amount)) {
19                sourceAccount.withdraw(amount);
20                this.notificator.notify();
21            }
22        }
23    }

```

```

java-21-09 - BankDI/src/main/java/it/unibo/sweng/bankdi/BankModule.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
TransferManager.java EmailNotifier.java Main.java SETRemoteBankOperator.java BankModule.java
1 package it.unibo.sweng.bankdi;
2
3 import com.google.inject.AbstractModule;
4
5 public class BankModule extends AbstractModule {
6     @Override
7     protected void configure() {
8         bind(Notifier.class).to(EmailNotifier.class);
9         bind(RemoteBankOperator.class).to(SERemoteBankOperator.class);
10    }
11 }
12

```

In questo caso andiamo a fare l'effettivo Bind tra le classi e le rispettive interfacce. Questo lo possiamo considerare come un file di sistema.

In questo modo andiamo ad eliminare del tutto l'uso di New in modo da evitare i pericoli che il new porta con se.

Definiamo una factory che a seconda delle specifiche ci ritorni un metodo.

Possiamo anche usare una Factory di tipo Notifier in modo da non dire in maniera esplicita la preferenza, lasciando alla factory il ruolo di creare il *Notifier* corretto.

Un approccio di questo genere ci porta ad una alta flessibilità, ma ad una creazione di tante classi che ci porta ad un aumento del rischio di errore, portandoci quindi a dover fare una revisione, eventuale, del codice.

## Revisione per il controllo (DVCS)

Controllo della revisione degli artefatti prodotti e modificati durante il processo di sviluppo. Ci vuole un meccanismo in cui la condivisione del codice sorgente ed i relativi documenti siano facilmente condivisibili.

Siccome può far comodo tenere traccia dell'evoluzione storica del progetto abbiamo due esigenze: facile condivisione e meccanismo che gestisca il concetto di revisione degli artefatti durante il tempo.

Esistono sistemi che nascono per fare revisioni del codice per un solo utente e non per più utenti, con successive modifiche poi per aprirlo ad altri utenti tramite repository centrale (che fa da server)

### Principi base

#### Working copy

Sono i file di progetto così come sono memorizzati nel file system usato dal team di sviluppo

#### Repository

entità che va gestire e memorizzare con l'evoluzione del tempo la versione del progetto

#### The pending changeset

tutti quei file che sono stati modificati nel working copy, ma che non hanno ancora passato la revisione

#### Commit

Per fare in modo che il repository faccia la revisione dobbiamo fare il commit per caricare le modifiche all'interno del repository

#### Update

L'operazione inversa rispetto al commit, in cui andiamo ad aggiornare i file contenuti nel file system con le modifiche caricate nel repository

### Artefatto iniziale

con la creazione di diversi artefatti abbiamo la creazione di un documento vuoto, qualunque tipo esso sia

#### Add

operazione esplicita al sistema in cui andiamo ad aggiungere un nuovo artefatto

#### Edit

modifica dell'artefatto

## Delete

eliminazione dell'artefatto contenuto all'interno del repository non sincronizzandolo più

## Rename

operazione sui metadati in cui andiamo a cambiare il nome del file all'interno del sistema, non andandolo a considerare come un nuovo file, ma come una modifica dello stesso file

## Move

cambio della directory del file.

Importante segnalare che è lo stesso file ad essere spostato, altrimenti non lo trova più all'interno del progetto

## Status

operazione di interrogazione in cui andiamo a chiedere lo stato di un artefatto

Ci dice quali sono i diversi file che sono stati aggiunti, le modifiche su di essi ecc.

## Diff

operazione che ci segnala le differenze tra il file nel file system e quelle del repository e quelle che sono le varie differenze tra le varie versioni del file o dei singoli artefatti

## Revert

Operazione che ci fa ritornare indietro nel tempo andando a riutilizzare una vecchia versione dei file

## Log

opera su un singolo elemento e serve per mostrare le singole revisioni su quel particolare file

Non solo abbiamo la data, generata dal sistema quando facciamo la modifica, ma anche una descrizione per tenere traccia delle modifiche effettuate in quella particolare situazione

## Tag

nomi che diamo alle revisioni in cui andiamo a fotografare la revisione o modifica della revisione.

Serve per facilitare a quale versione della revisione voglio ritornare.

## Branch

l'evoluzione dell'artefatto su due linee temporali diverse.

Solitamente viene creato un nuovo branch quando non vogliamo incasinare l'artefatto con delle modifiche di prova dell'artefatto stesso.

## Merge

Quando vogliamo unire le due linee temporali dei branch in una singola linea temporale

In caso di modifica della stessa porzione di codice si genera un conflitto

## Resolve

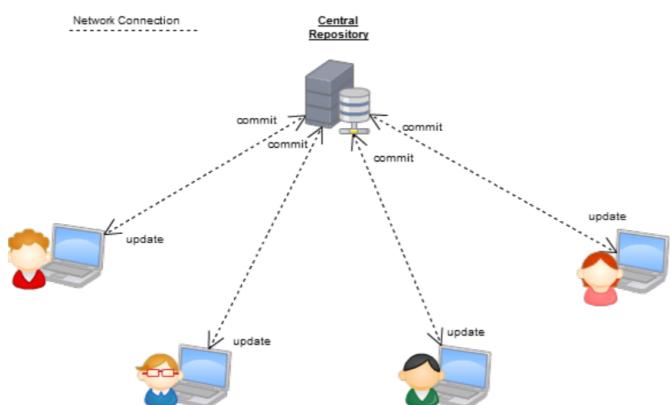
Ci permette quale versione dei due diversi Branch vogliamo scegliere

## Lock

Operazione che non rende accessibile per la sincronizzazione di un artefatto dentro il repository fino a che l'utente che ha preso il lock, non lo rilascia.

Metodo con il quale avvisiamo agli altri membri del team che non possono modificare quel particolare file per evitare problemi

## Centralized version control



ciascun utente lavora su una copia locale.  
E' il repository centrale l'unica voce della  
verità.  
Le copie di lavoro si sincronizzano con il  
repository con *update* e *commit*

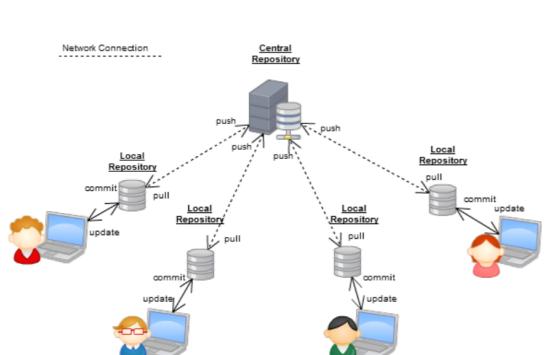
## Gestione dei conflitti

per evitare conflitti tra file possiamo usare il **locking**.

Se non usiamo locking e abbiamo conflitto possiamo ricopiare la nuova versione sul repository ricopiando le modifiche che hai fatto all'interno del nuovo file.

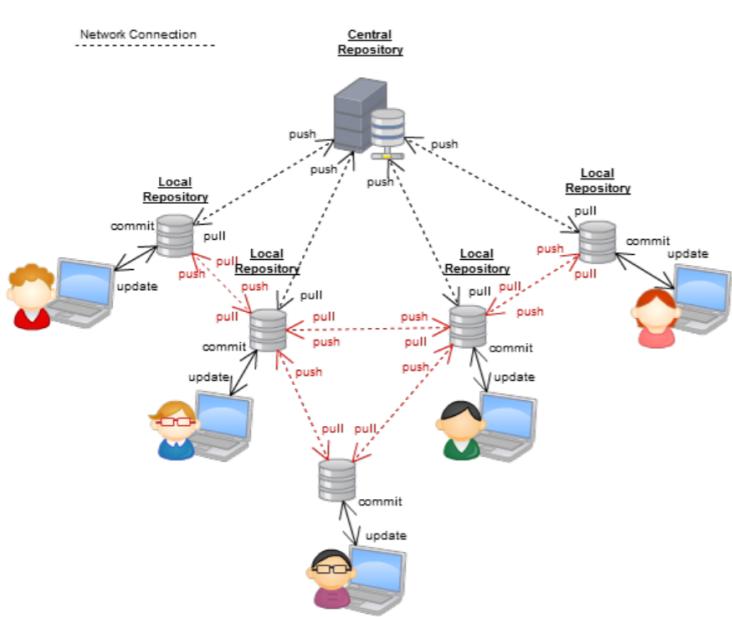
Sono due scelte che comunque sono scomode in quanto con il locking abbiamo problema che il file può essere rilasciato solo da chi l'ha messo.

## Simple distributed version control



Ogni utente ha il proprio repository dove tiene conto delle modifiche.  
Questi repository locali possono sincronizzarsi tra loro. In questo sistema abbiamo il repository centrale che gestisce il tutto.  
Tramite *push* e *pull* andiamo poi a sincronizzare il repository locale con il repository remoto.

## Fully distributed version control



In questo caso abbiamo una sincronizzazione tra i vari repository locali. Fa uso della tecnica peer-to-peer. Il repository centrale può esistere come no, in quanto non è la singola fonte della verità.

## Branches

può essere utile avere diverse linee (branche) e a seconda del sistema che stiamo utilizzando, la gestione dei Branch possono essere diversi.

Questi branch formano delle gerarchie:

- child branch: branch che si suddivide da una linea temporale
- genitore: branch padre del branch preso
- trunk: branch iniziale che non ha genitor

I branch che presentano diverse linee temporali possono essere riallineate tramite merge

## git

tra i diversi strumenti per la gestione delle revisioni abbiamo uno in particolare, che è il più moderno e diffuso, git.

Progettato per essere affidabile, molto efficiente e per funzionare in ambiente distribuito (basato sul concetto di repository locale, più repository centrale)

Supporta nella maniera più semplice possibile il branching in modo da fare una singola operazione nella maniera più semplice possibile per invitare lo sviluppatore a fare un continuo branching.

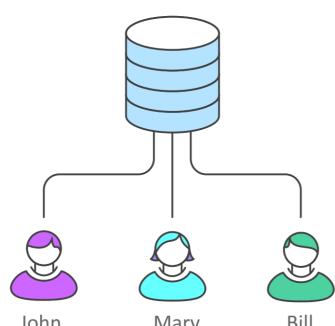
In particolare analizziamo di flusso di lavoro possibile che gestiscono per la revisione dei lavori git.

Un team di sviluppo deve capire come utilizzare git, ovvero quale workflow usare

## Workflow centralizzato

Ciascun repository locale fa riferimento ad un repository centrale.

Ogni artefatto viene rappresentato come sequenza di revisioni.



Anche nei repository locali abbiamo tutte le varie variazioni degli artefatti e delle revisioni.

Quando andiamo a sincronizzare il locale con il remoto abbiamo che i cambiamenti vengono comunicati ai Master che si occupa di aggiornare il repository aggiungendo le modifiche

Può succedere che i local repository siano disallineati.

Per evitare ciò dobbiamo:



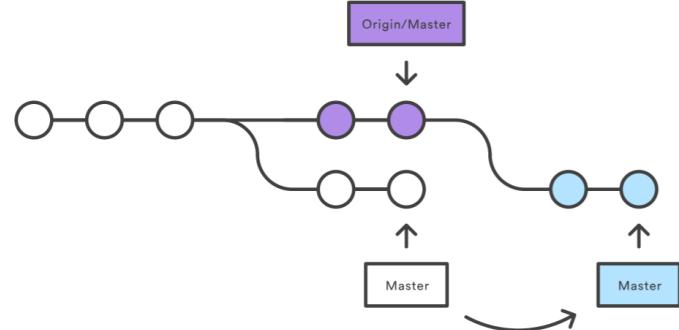
Nel repository locale del puntino viola avrà le varie versioni inclusa la sua.

Il puntino azzurro effettua delle modifiche, anche lui sullo stesso file.

Se il punto viola è il primo a caricare il file, il repository capisce che è un'evoluzione del suo e quindi si aggiorna.

Se poi anche il puntino azzurro vorrebbe fare il push, il repository riconosce che c'è un problema e non lo fa caricare.

Per risolvere questo conflitto il puntino azzurro dovrà fare un **pull** (rebase) in cui va a inserire le modifiche che il repository centrale ha nel proprio file e poi rifare, se tutto funziona nonostante il cambiamento, il **push** dell'artefatto.



Se tutto va bene e non ci sono problemi allora viene caricato, altrimenti bisogna usare degli add per segnalare eventuali problemi che ci sono all'interno del codice.

Se siamo nel caso in cui dobbiamo specificare, abbiamo che dobbiamo segnare il repository remoto e quale linea temporale considerare.

## Feature workflow branch

In questo caso, quando abbiamo troppi conflitti andiamo ad inserire tutte le feature in sviluppo dentro un branch dedicato

Una volta completata la funzione, il ramo della funzione è push e una *pull request* è aperta. *pull request*: richiesta che avviene fuori dal sistema di revisione, nel quale un membro del team che ha finito di lavorare sul branch richiede al gestore delle revisioni di accettare le modifiche che ha fatto all'interno del branch.

la pull request può essere accettata o respinta, nel caso venga accettata allora possiamo andare a fare merge e risolvere i problemi.

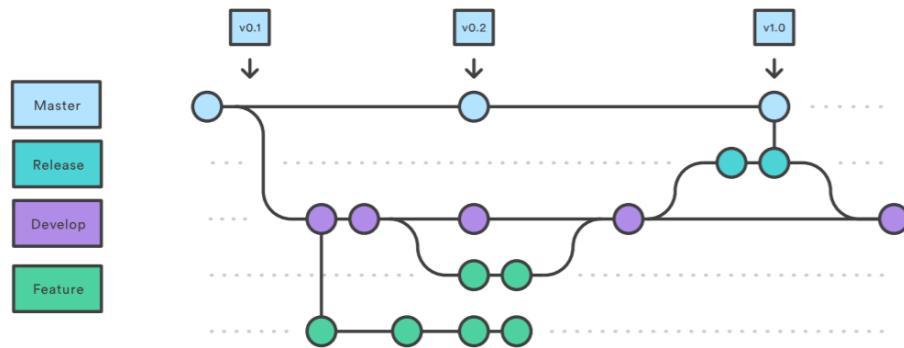
Una pull request può essere anche rifiutata per i troppi conflitti tra i file. In tal caso ci sarebbe da fare un eventuale rebase del codice.

## Gitflow workflow

Ci sono diversi branch:

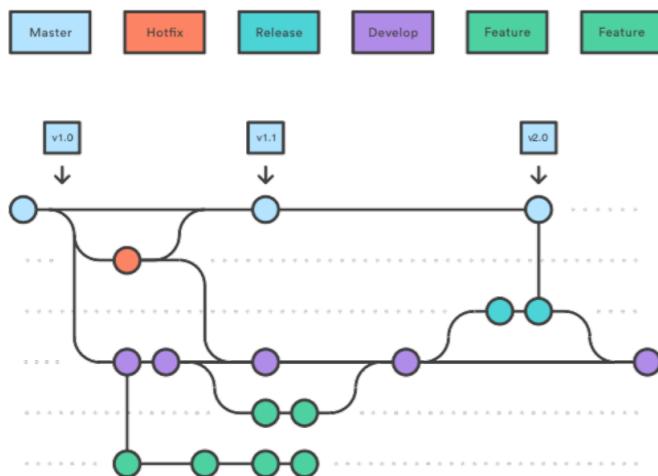
- principali: Master e Develop branch
- futuri branch che derivano da Develop

Quando siamo vicini al rilascio, allora portiamo i file da Develop branch a Release branch. Dopo che abbiamo caricato tutto, allora andiamo a creare una nuova revisione di Master andando ad inserire tutti i merge dentro.



i branch Hotfix sono gli unici branch che derivano da Master.

Soltanamente sono usati per la correzione di bug ed errori, facendo eventualmente anche un merge alla soluzione Master precedente



## Conclusion

DVCS è la colonna portante dei progetti software.

Tutti gli artefatti devono essere commissionati.

I VCS possono richiedere del tempo per essere padroneggiati ma imparare a usarli è tempo ben speso

## Tecnologie per creare una web application

lista di tecnologia di cui hai bisogno: HTML,DOM,Javascript,CSS,HTTP,XML/JSON,PHP

### GWT

progetto che nasce all'interno di Google, è una proposta che permette di organizzare applicazioni Web dinamiche senza il bisogno delle applicazioni.

E' java centrico e ciò ci permette di evitare i problemi delle tecnologie.

Trasforma automaticamente il codice Java nel corrispondente codice javascript.

Per la composizione dell'interfaccia grafica abbiamo un toolkit grafico molto simile a Swing.

Ambiente di sviluppo che fornisce librerie per trasformare il java in js per farlo girare su browser.

### Client side

Non tutto ciò che è scritto in java è supportato in js (fino a java 11).

Ci sono limitazioni:

- non possiamo fare multithreading ed annessa sincronizzazione
- no riflessione
- no serializzazione
- Limited API

### Server side

il codice che contiene la logica di business è eseguito in un server container.

nessuna conoscenza API è necessaria, in quanto automatica.

La comunicazione con il client side code è gestito dalla RPC.

### Client server interaction

### GWT RPC

è un metodo semplificato da rendere il client code interagibile con in server code.

Server code è strutturato in servizi, classi di Java, che estendono *RemoteServiceServlet* e implementa un'interfaccia che descrive le operazioni disponibili al client.

Viene anche fornita una versione asincrona dell'interfaccia.

I services sono i nostri controller, dobbiamo quindi pubblicare i controller che devono essere esportabili per i nostri services.

Dobbiamo rendere disponibili i nostri controller per i services.

Dobbiamo quindi creare un'interfaccia che deve estendere un'interfaccia base che ci viene fornita.

Nell'interfaccia finale inseriamo quei metodi che devono essere visibili dai services.

Dobbiamo scrivere il codice lato server che implementa dell'interfaccia e poi dobbiamo fornire la parte asincrona della medesima interfaccia.

Se costruiamo oggetti che vedono GWT vuol dire che abbiamo sbagliato qualcosa in quanto le uniche cose visibili dal GWT devono essere i controller.

La versione asincrona è una versione che non ritorna nessun tipo (tutti void) che prende tutti i parametri più un parametro generico che prende il tipo di ritorno del metodo iniziale (della versione non asincrona). Questo è il codice che deve essere eseguito quando si ha una risposta dal server (sempre lato client).

## Sessions

l'approccio più elementare è quella di avere una sola pagina di partenza che presenta le righe per l'istanziazione della parte server-side.

In caso di nuova interfaccia andiamo a cancellare e riscrivere le interfacce da 0.

Non è il metodo migliore (anche se per il nostro progetto va benissimo così).