



Ingegneria del

## Software

Corso di Laurea in Ingegneria Informatica e Robotica – A.A. 2022-2023

DIPARTIMENTO DI INGEGNERIA

docente

Prof. Marco MONTECHIANI

# Ingegneria del Software

studenti

**Nadi Billal** billal.nadi@studenti.unipg.it

# 0. Indice

<b>1</b>	<b>Introduzione</b>	<b>4</b>
1.1	L'importanza del software . . . . .	4
1.2	Introduzione all'ingegneria del software . . . . .	5
<b>2</b>	<b>Le qualità del software e i principi dell'ingegneria del software</b>	<b>8</b>
2.1	Le qualità del software . . . . .	8
2.1.1	Aree applicative . . . . .	11
2.2	I principi dell'ingegneria del software . . . . .	13
<b>3</b>	<b>Modelli di produzione del software</b>	<b>17</b>
3.1	I processi per la produzione di software . . . . .	17
3.2	Modello a cascata . . . . .	22
3.3	Modelli evolutivi . . . . .	23
3.4	Sviluppo agile . . . . .	25
3.5	Scrum . . . . .	29
3.6	Problemi e scalabilità dei metodi agili . . . . .	32
3.7	DevOps . . . . .	33
<b>4</b>	<b>Esercitazione 1: Controllo delle Versioni</b>	<b>41</b>
4.1	Introduzione alla gestione della configurazione . . . . .	41
4.2	Controllo delle versioni . . . . .	42
4.3	GIT . . . . .	46
4.4	Esercitazione . . . . .	48
4.4.1	Clone di un repository . . . . .	48
4.4.2	Tracciamento dei file . . . . .	49
4.4.3	Stato dei file . . . . .	50
4.4.4	Tracciare un file . . . . .	50
4.4.5	Mettere un file in stage . . . . .	51
4.4.6	Commit delle modifiche . . . . .	52
4.4.7	Rimuovere file . . . . .	52

4.4.8	Cronologia dei commit . . . . .	52
4.4.9	GITK . . . . .	53
4.4.10	Branching . . . . .	54
4.4.11	Merging . . . . .	58
4.4.12	Listare e rimuovere branch . . . . .	58
4.4.13	Esempi di merging . . . . .	59
4.4.14	Conflitti . . . . .	59
4.4.15	Scaricare le modifiche da server remoto . . . . .	60
4.4.16	Push su server remoto . . . . .	60
4.4.17	Github e Bitbucket . . . . .	60
<b>5</b>	<b>Ingegneria dei requisiti</b>	<b>61</b>
5.1	Introduzione all'ingegneria dei requisiti . . . . .	61
5.2	Deduzione dei requisiti . . . . .	65
5.3	Specificazione dei requisiti . . . . .	67
5.4	Convalida dei requisiti . . . . .	68
<b>6</b>	<b>Progettazione Architetturale</b>	<b>70</b>
6.1	Introduzione . . . . .	70
6.2	Architettura a strati . . . . .	75
6.3	Architettura repository . . . . .	78
6.4	Architettura pipe-and-filter . . . . .	79
6.5	Architettura client-server . . . . .	81
6.6	Architettura model-view-controller . . . . .	84
<b>7</b>	<b>Architetture a servizi e microservizi</b>	<b>87</b>
7.1	Architettura a servizi . . . . .	87
7.2	Architettura a microservizi . . . . .	93
<b>8</b>	<b>Esercitazione 2: Progetto di API REST</b>	<b>106</b>
<b>9</b>	<b>Programmare ad oggetti</b>	<b>115</b>
9.1	Programmazione a oggetti . . . . .	115
<b>10</b>	<b>Esercitazione 3: UML</b>	<b>120</b>
10.1	Introduzione a UML . . . . .	120
10.2	Diagramma delle classi . . . . .	122
10.3	Esercizio . . . . .	134
<b>11</b>	<b>Design patterns</b>	<b>135</b>
11.1	Introduzione ai design patterns . . . . .	135
11.2	Pattern creazionali . . . . .	136

11.3 Pattern strutturali . . . . .	140
11.4 Pattern comportamentali . . . . .	144
<b>12 Test del software</b>	<b>160</b>
12.1 Il processo di test . . . . .	160
12.2 Test di sviluppo . . . . .	161
12.3 Test di release . . . . .	169
12.4 Test degli utenti . . . . .	170

# 1. Introduzione

## 1.1 L'importanza del software

Per un azienda il Software è qualcosa di importante infatti, il lavoro con la maggiore richiesta è quella del Software Engineer. Nel 2023 si hanno milioni di visitatori in software come YouTube, Gmail, Facebook, ecc.. Purtroppo dietro a questi grandi nomi ci sono stati anche dei fallimenti. Quanto costano i fallimenti del software? Abbiamo

- Danni Economici (miliardi di euro all'anno):
  - Costi vivi di progetto;
  - Mancati Introiti;
  - Danni di immagine
- Danni Sociali:
  - Disservizi;
  - Rischi relativi alla sicurezza;
  - Sperpero di soldi pubblici

Quindi per evitare questi fallimenti occorre andare a fare l'individuazione degli errori. Infatti più tardi l'errore viene individuato e più costerà correggerlo: correggere un errore dopo che il software è stato messo in produzione può costare 100 volte di più che correggerlo durante le fasi di sviluppo. Più è complesso il software, e più sarà complicato comprendere la causa dell'errore. La correzione stessa dell'errore potrebbe avere conseguenze non previste e generare nuovi errori.

Possibili cause di fallimenti possono essere quello di trattare lo sviluppo di software come un costo piuttosto che come un investimento: infatti trattandolo così si può andare a

- Tagliare i costi;
- Velocizzare i tempi;

In realtà, investire poco nello sviluppo potrebbe portare ad una lievitazione dei costi e dei tempi, a causa di errori in fase di progettazione e sviluppo.

Un altro errore risulta essere quello da parte degli sviluppatori di compensare lo scarso tempo o le scarse risorse tagliando le fasi di test e verifica del codice.

Oltre tutto la tecnologia cambia rapidamente e spesso si cerca di usare tecnologie innovative per avere un vantaggio sui competitor e per diminuire il rischio di obsolescenza prematura del software, però se la tecnologia usata è però immatura e poco testata, può contenere errori e incompatibilità non facilmente prevedibili.

Progetti molto grandi e complessi tendono a fallire più facilmente perché è quasi impossibile testare in maniera completa sistemi software complessi, è possibile testare in maniera soddisfacente i singoli componenti, ma non tutte le loro possibili interazioni.

Avere pratiche di sviluppo superficiali renderà più facile creare errori e più difficile trovarli, il sistema potrebbe non rispettare i desideri del committente, oltretutto non sarà chiaro cosa è stato fatto e cosa andrà fatto.

A carico del project manager ci può essere una cattiva gestione dei rischi, infatti occorre saper identificare cosa può andare storto, pianificare cosa fare per mitigare le possibilità che qualcosa vada storto, e aggiornare regolarmente il piano dei rischi. Altri fattori importanti a carico del project manager sono la comunicazione, la revisione del progetto a intervalli regolari, e la gestione delle risorse umane.

Vi sono come cause di possibili fallimenti anche delle esigenze politiche che ti costringono ad anticipare i tempi di rilascio, modificare alcuni requisiti in fase già avanzata di sviluppo, oppure a porre dei tagli al budget che sono imprevisti.

## 1.2 Introduzione all'ingegneria del software

Il termine software identifica non soltanto i programmi, ma anche la documentazione associata, le librerie da cui dipendono, i siti web di supporto, i dati di configurazione, ecc.

I sistemi software sono spesso formati da vari software indipendenti che cooperano insieme.

Il software è destinato ad essere utilizzato da persone diverse da chi lo ha sviluppato, e sarà soggetto a revisione e manutenzione durante la sua vita.

La mancanza di vincoli naturali fa sì che il software possa diventare estremamente costoso, difficile da capire e costoso da modificare.

Il software è estremamente versatile, viene applicato in contesti molto diversi tra loro e tende a conformarsi agli altri elementi che compongono il sistema (tipicamente meno modificabili).

Il software è intangibile e come tale non ha rappresentazioni geometriche "naturali", ma viene rappresentato mediante diagrammi astratti e fortemente interconnessi, di difficile comprensione.

**L'ingegneria del software** è l'applicazione di un approccio sistematico, disciplinato e quantificabile allo sviluppo, l'esercizio e la manutenzione del software - ovvero, l'applicazione dell'ingegneria del software.

Cosa sa fare l'ingegnere del software? Sa programmare - conosce algoritmi, strutture dati, e linguaggi di programmazione, sa tradurre richieste e "desideri" in precise specifiche, sa comunicare anche con personale non tecnico, sa muoversi attraverso diversi livelli di un progetto e conosce vari modelli che applica in base al livello su cui ragiona, sa coordinare il lavoro di uno o più team e sa prendersi delle responsabilità.

Queste responsabilità sono spesso divise tra più specialisti con qualifiche differenti. Ad esempio:

- L'analista di sistema ricava i requisiti, interagisce col cliente e comprende l'area applicativa.
- L'analista di prestazioni analizza le prestazioni del sistema.
- Lo sviluppatore senior coordina lo sviluppo e si avvale di sviluppatori junior.

Alcune aree di conoscenza di questi specialisti sono:

- Requisiti del software
- Progettazione del software
- Sviluppo del software
- Verifica del software

- Manutenzione del software
- Gestione delle configurazioni del software
- Gestione dell'ingegneria del software
- Processi dell'ingegneria del software
- Strumenti e metodi dell'ingegneria del software
- Qualità del software

Come detto in precedenza il software è un **prodotto** invisibile, intagibile, facilmente duplicabile ma costosissimo: è un'opera dell'ingegno protetta dalle leggi. Il software è sempre il frutto di un **processo di sviluppo** che inizia con un'idea e termina quando il software viene ritirato. L'industria mondiale del software oltretutto è in forte crescita con tassi dal cinque al dieci per cento annuo. Il costo di sviluppo di un prodotto software tende a crescere in proporzione al quadrato delle sue dimensioni. Si può affermare una duttilità del software infatti modificare il software è possibile, anche senza modificarne il progetto. Questa proprietà però è spesso mal utilizzata: mentre una modifica a un ponte o a un'automobile è sempre sottoposta ad una attenta revisione del progetto, questo a volte non avviene nel software.

Gli ingegneri del software si trovano spesso di fronte a richieste di modifiche sostanziali. Gli utenti ingenuamente hanno l'impressione che sia sufficiente modificare un po' il codice per ottenere il risultato desiderato.

A differenza di altri prodotti industriali, il processo di fabbricazione del software non ha grandi costi (è sufficiente duplicarlo o renderlo disponibile in una qualche forma). Il costo del processo di produzione è invece determinato dalla progettazione e implementazione del software. Queste attività richiedono molto lavoro umano e pochi macchinari.

## 2. Le qualità del software e i principi dell'ingegneria del software

### 2.1 Le qualità del software

Sebbene il software sia intangibile, possiamo attribuirgli delle qualità:

- **Qualità Interne** del software riguardano aspetti statici, che non dipendono dall'esecuzione del software
- **Qualità Esterne** misurano invece i comportamenti del software
- **Qualità in uso** rappresentano il punto di vista dell'utente sul software e dipendono dunque sia dalle qualità esterne che interne

Le qualità interne aiutano gli sviluppatori a raggiungere le qualità esterne desiderate. Il prodotto dell'ingegneria del software è il sistema software consegnato al committente o al cliente (codice, manuali,...). Tale prodotto si realizza mediante un **processo**, attraverso il quale vengono creati diversi prodotti intermedi (documenti, dati di test, rilasci intermedi). I prodotti intermedi sono soggetti agli stessi requisiti di qualità del prodotto finale. La **qualità del processo** influenza la qualità del prodotto.

Esiste uno standard da seguire per quello che è la produzione software ossia la **ISO/IEC 25010** descritta dalle seguenti caratteristiche:

- **Correttezza:** è una qualità esterna, descrive la capacità di un prodotto software di fornire funzioni che soddisfano esigenze stabilite, necessarie per operare sotto condizioni specifiche (le specifiche fornite devono essere non ambigue, tanto più le specifiche sono scritte in maniera rigorosa, tanto più sarà possibile verificare con precisione la correttezza di un software). Il limite

di questa qualità sta nell'essere di tipo assoluto (si/no) e nel dipendere dalla correttezza delle specifiche.

- **Affidabilità:** è una qualità esterna, descrive la capacità del prodotto software di mantenere uno specificato livello di prestazioni quando usato in date condizioni per un dato periodo (I software corretti (assumendo che le specifiche siano corrette) sono anche affidabili). Software non corretti possono essere considerati comunque affidabili, qualora il difetto non abbia un impatto importante. L'affidabilità di un software può migliorare col tempo, grazie a interventi di manutenzione
- **Robustezza:** è una qualità esterna, descrive la capacità del prodotto di comportarsi in modo accettabile anche in circostanze non previste nella specifica dei requisiti. (ad esempio in caso di input non corretto o di fallimenti dell'hardware). Un programma corretto potrebbe non essere robusto, semplicemente perché la specifica dei requisiti non prevede alcune casistiche.
- **Prestazioni:** è una qualità esterna, descrive l'efficienza del prodotto nell'utilizzo delle risorse interne del computer (memoria, potenze di calcolo, rete). Possiamo valutare le prestazioni di un sistema mediante misure, analisi e simulazioni. Le prestazioni influenzano l'usabilità e la scalabilità di un software (Se un software è troppo lento diventa inusabile, un software con scarse prestazioni potrebbe essere praticamente usabile solo per dimensioni dell'input non troppo grandi (ovvero non scalabile)).
- **Usabilità:** è una qualità esterna, descrive la capacità del prodotto software di essere capito, appreso, usato e benaccetto dall'utente, quando usato sotto condizioni specificate. (Usabilità è una qualità soggettiva, può dipendere dai gusti e dall'esperienza) L'interfaccia grafica influenza molto sull'esperienza che l'utente percepisce nell'usare il software (un sistema che presenta un interfaccia coerente e prevedibile è tipicamente usabile, la standardizzazione delle interfacce è un fenomeno in crescita e tende a migliorare l'usabilità dei sistemi che adottano componenti e interfacce standard).
- **Verificabilità:** è una qualità interna, misura la facilità con cui è possibile verificare la correttezza e le prestazioni del prodotto (metodi di programmazione modulare, norme sistematiche di codifica, e uso di linguaggi di programmazione appropriati alla scrittura di codice ben strutturato aiutano a garantire tale qualità), in alcuni casi può essere vista come qualità esterna ad esempio, quando la sicurezza di un software è un aspetto critico e deve essere verificabile.
- **Manutenibilità:** è una qualità interna, descrive la capacità del software di

essere modificato, includendo correzioni, miglioramenti o adattamenti (costo manutenzione sessanta percento del costo totale). Vi sono varie tipologie di manutenzioni:

- La **manutenzione correttiva** riguarda la rimozione di errori esistenti sin dal primo rilascio del software, o introdotti successivamente
- La **manutenzione adattiva** riguarda le modifiche dell'applicazione in risposta a cambiamenti dell'ambiente (ad esempio, una nuova versione dell'hardware, del sistema operativo o del DBMS)
- La **manutenzione perfettiva** riguarda i cambiamenti necessari per migliorare alcune qualità (modifica o aggiunta di funzioni, migliorare le prestazioni o l'usabilità, ecc..)

Tipicamente, la manutenzione correttiva pesa per il venti percento dei costi di manutenzione, così come quella adattiva, mentre il restante sessanta percento è dovuto a costi di manutenzione perfettiva. Il software legacy (o software ereditato) si riferisce a software presente in un'organizzazione da lungo tempo, di valore strategico poiché incorpora molti processi importanti. Tale software spesso si poggia su tecnologia desueta, ed è pertanto difficile da modificare e manutenere. Tecniche di reverse engineering e reengineering possono essere adottate per scoprire la struttura del software legacy e per ristrutturarlo. La manutenibilità di un software può essere vista come la somma di due qualità:

- **Riparabilità:** facilità con cui si eliminano difetti dal software.
  - **Evolvibilità;** facilità con cui si apportano cambiamenti al software.
- **Portabilità:** è una qualità interna, descrive la capacità del prodotto di essere eseguito in ambienti diversi(piattaforme hardware, sistemi operativi ecc..). La portabilità può essere ottenuta isolando le dipendenze in pochi moduli del software, facilmente modificabili in base all'ambiente (ad esempio, le librerie grafiche, o l'accesso al filesystem).
  - **Interoperabilità:** è una qualità interna, descrive la capacità di coesistere e cooperare con altri sistemi(ad esempio, da un elaboratore di testi ci si aspetta la possibilità di incorporare diagrammi prodotti da un pacchetto grafico). L'utilizzo di interfacce standard semplifica l'interoperabilità.
  - **Produttività:** è una qualità di processo, nello specifico è una qualità del processo di produzione del software che ne indica l'efficienza e le prestazioni. Dipende sia dai singoli sviluppatori, che dal management, che dagli strumenti

usati. Il riuso del software favorisce la produttività, sebbene lo sviluppo di moduli riusabili è più dispendioso e quindi va valutato in prospettiva.

- **Tempestività:** è una qualità di processo, descrive la capacità di rendere disponibile un prodotto al momento giusto. Sebbene il software andrebbe consegnato solo se in possesso di tutte le altre qualità attese, consegnare versioni preliminari può aiutare a raccogliere critiche e suggerimenti. Un'attenta **pianificazione** del processo, un'accurata **stima** delle attività, e una specifica chiara degli obiettivi intermedi (milestone), sono fondamentali per conseguire la tempestività. La specifica dei requisiti è inoltre fondamentale per evitare la produzione di un software che sarà già obsoleto nel momento in cui verrà rilasciata. La **consegna incrementale** del software, ovvero la consegna incrementale di set di funzionalità parziali ma significative, può favorire la tempestività
- **Visibilità:** è una qualità di processo, infatti un processo di sviluppo del software è **visibile** o **trasparente**, se tutti i suoi passaggi sono documentati in modo chiaro. Un processo visibile consente ai vari attori di avere chiaro lo stato del progetto, potendo così sopesare le loro scelte, lavorando tutti nella stessa direzione. Un prodotto è visibile se è ben strutturato come una collezione di componenti, con funzioni ben comprensibili e con un'accurata documentazione disponibile

### 2.1.1 Aree applicative

Le principali aree applicative sono:

- **Sistemi Informativi:** lo scopo primario di un sistema informativo è quello di gestire le informazioni di una organizzazione (sistemi bancari, sistemi bibliotecari, Enterprise Resource Planning ecc..) Il cuore di un sistema informativo è una base di dati, utilizzata mediante transazioni che creano, ricercano, modificano, o cancellano i dati. Molti di questi sistemi offrono inoltre una interfaccia web per operare sulle informazioni gestite. I sistemi informativi sono applicazioni orientate alla gestione dei dati e possono dunque essere caratterizzati in base al modo con cui elaborano i dati. Di seguito alcune qualità caratteristiche:
  - Integrità dei dati: capacità di garantire la non corruzione dei dati anche a fronte di determinati malfunzionamenti
  - Sicurezza: capacità di fornire un opportuno livello di protezione rispetto all'accesso ai dati.

- Disponibilità dei dati: capacità di limitare le condizioni e gli intervalli temporali in cui i dati non sono accessibili
- Prestazioni delle transazioni: capacità di eseguire più transazioni simultaneamente per unità di tempo.

Anche l’usabilità è essenziale per un sistema informativo, poichè dovrà essere usato da utenti anche inesperti o con scarsa predisposizione all’utilizzo di strumenti tecnologici

- **Sistemi in tempo reale:** sono caratterizzati dalla necessità di dover rispondere a determinati eventi entro un tempo prefissato e limitato. In un sistema di monitoraggio industriale il software deve rispondere a cambiamenti improvvisi di temperatura, attivando certi dispositivi e inviando segnali di allarme. Il software di controllo del volo di un aereo deve monitorare le condizioni ambientali e la posizione corrente dell’aereo, e controllare la traiettoria di volo in funzione di queste. Il software che gestisce il mouse di un computer deve rispondere rapidamente in modo da distinguere tra singolo click e doppio click. I sistemi in tempo reale si dicono orientati al controllo, e si basano su un pianificatore(**scheduler**) in grado di ordinare le azioni del sistema. Il **tempo di risposta** è dunque una qualità caratterizzante per i sistemi in tempo reale. Anche l’affidabilità è importante, poichè spesso adottati in contesti critici. Spesso si parla di **safety**, ovvero la capacità del sistema di evitare rischi inaccettabili(ciò che non dovrebbe mai capitare durante l’esecuzione di un sistema)
- **Sistemi Distribuiti:** sono composti da macchine indipendenti o semi-indipendenti collegate da una rete di telecomunicazione. L’ambiente di sviluppo deve supportare lo sviluppo dell’applicazioni su molteplici computer, su cui gli utenti devono essere in grado di compilare, collegare, e testare il codice. Linguaggi interpretati come Java e Csharp sono particolarmente adatti a questi ambienti eterogenei. alcune caratteristiche importanti per questi sistemi:
  - Il **livello di distribuzione**, il quale indica se è possibile distribuire i dati, l’elaborazione, o entrambi.
  - La possibilità di **tollerare il partizionamento** (in presenza ad esempio di un collegamento di rete non funzionante che divide il sistema in più sottosistemi).
  - La possibilità di **tollerare uno o più computer non funzionanti** senza far venire meno l’operatività del sistema.

In un sistema distribuito, l'affidabilità e le prestazioni possono essere aumentate replicando i dati su più macchine. Un ulteriore beneficio in termini di prestazioni si ottiene rendendo il codice mobile, in grado cioè di migrare durante l'esecuzione, ad esempio verso il nodo che memorizza i dati da elaborare.

- **Sistemi Embedded:** sono sistemi nei quali il software è solo uno dei componenti e spesso non ha interfacce rivolte all'utente finale, ma solo verso altri componenti del sistema che esso controlla. In questo caso le interfacce tra componenti possono essere rese più complicate, se questo aiuta ad esempio a semplificare i dispositivi collegati.

Spesso i sistemi presentano più caratteristiche, ad esempio, i sistemi embedded sono spesso anche in tempo reale, i sistemi informativi spesso inglobano componenti embedded e componenti in tempo reale. Mentre alcune qualità e proprietà del software sono facilmente misurabili, es: prestazioni, per altre non esistono metriche universalmente riconosciute, es: manutenibilità. Esempi di **metriche** (unità e modalità di misura) riconosciute:

- Lines of code e functions points per la dimensione del software;
- Cyclomatic complexity per la complessità di un software;
- Mean Time To Failure e Mean Time Between Failure per l'affidabilità.

## 2.2 I principi dell'ingegneria del software

I **principi** dell'ingegneria del software descrivono proprietà desiderabili del processo e dei prodotti che riguardano lo sviluppo del software. Per poterli applicare, l'ingegnere del software deve disporre di **metodi** appropriati e di **tecniche** specifiche. Le **metodologie** coordinano un insieme di metodi e tecniche consistenti tra loro secondo un approccio comune. Gli **strumenti** supportano l'applicazione di una metodologia. Sebbene lo sviluppo del software sia un'attività creativa, non va perseguita istintivamente in maniera scarsamente strutturata. Il rigore e il formalismo non limitano la creatività ma la complementano, come in ogni attività ingegneristica. Occorre dunque andare a scegliere il livello di rigore e formalità da raggiungere, in funzione della difficoltà concettuale e della criticità del compito che si sta affrontando. Questo livello può differire a seconda delle diverse parti di uno stesso sistema. La fase di codifica invece utilizza un approccio formale, in quanto i linguaggi di programmazione hanno una sintassi e semantica definite. Per esempio i compilatori sono in grado di verificare la correttezza formale di un programma e trasformarlo in un'altra rappresentazione equivalente(ad esempio in

linguaggio macchina). Queste operazioni automatiche, rese possibili dal formalismo dei linguaggi di programmazione, migliorano la verificabilità e l'affidabilità di un programma. Rigore e formalismo possono essere applicati anche ad altre fasi lungo il ciclo di vita del software.(Ad esempio una documentazione rigorosa anche se non formale può avere effetti benefici sulla manutenibilità, riusabilità, portabilità, comprensibilità e interoperabilità. Una documentazione rigorosa può facilitare il riuso di parti del processo di sviluppo.) Il **principio di separazione degli interessi** ci consente di affrontare differenti aspetti del problema, concentrando la nostra attenzione su ciascuno di essi in maniera separata. Le decisioni da prendere durante lo sviluppo del software possono riguardare:

- Le caratteristiche del prodotto
- Il processo di sviluppo del prodotto
- Aspetti di tipo economico e finanziario

I modelli di sviluppo del software tipicamente separano le attività in diversi periodi temporali. Un altro tipo di separazione degli interessi riguarda le qualità che devono essere considerate separatamente. Un altro tipo di separazione riguarda la scomposizione del progetto in parti (moduli) disgiunti. Un'importante applicazione del principio di separazione degli interessi riguarda la separazione tra aspetti relativi al dominio del problema da quelli relativi al dominio dell'implementazione.

Un sistema complesso può essere suddiviso in parti più piccole chiamate **moduli**, infatti un sistema composto da moduli viene detto **modulare**. In questo modo è possibile applicare il principio della separazione degli interessi in due fasi:

- una fase prevede la trattazione dei dettagli di ogni singolo modulo separatamente
- un'altra fase prevede la trattazione delle relazioni tra i moduli al fine di integrarli coerentemente

L'approccio **bottom-up** prevede la progettazione del software trattando prima i singoli moduli separatamente e poi componendoli insieme; mentre l'approccio **top-down** studia prima la scomposizione del problema in moduli, e poi studia i singoli moduli separatamente.

I moduli devono essere progettati in maniera che abbiano **alta coesione** (elementi di un modulo strettamente connessi) e **basso accoppiamento** (moduli diversi devono dipendere poco l'uno dall'altro)

La modularità permea l'intero processo di sviluppo del software e dà luogo a quattro principali benefici:

- Capacità di scomporre un sistema complesso in parti più semplici (divide et impera);
- Capacità di comporre un sistema complesso a partire da moduli esistenti (Favorisce il riuso e aumenta la velocità di costruzione);
- Capacità di capire un sistema in funzione delle sue parti;
- Capacità di modificare un sistema modificando soltanto un piccolo insieme delle sue parti (favorisce la riparabilità e l'evolvibilità)

L'**astrazione** ci consente di identificare gli aspetti fondamentali di un fenomeno e di ignorare i suoi dettagli. Ciò che è importante e ciò che è un dettaglio dipende dallo scopo dell'astrazione. **Anticipare il cambiamento** significa cercare di prevedere quali possano essere i futuri cambiamenti che saranno richiesti al software, e progettare opportunamente il software al fine di rendere tali cambiamenti più semplici possibili. Anticipare il cambiamento significa rendere i singoli moduli o componenti facilmente adattabili ad eventuali cambiamenti. Anche il processo di produzione del software dovrebbe anticipare possibili cambiamenti (ricambio del personale, stimare i costi dei cambiamenti anticipabili, prevedere e articolare le fasi di manutenzione,...). Il principio di **generalità** ci dice che ogni volta che si deve risolvere un problema, si dovrebbe cercare di scoprire prima qual è il problema più generale che si nasconde dietro lo specifico problema da risolvere. Una soluzione più generale potrebbe però essere più costosa (velocità di esecuzione, occupazione di memoria, tempo di sviluppo, ecc..). Il mercato offre numerosi prodotti di uso generale (off-the shelf), molto spesso disponibili anche come servizi web. Il principio dell'**incrementalità** prevede che il processo di produzione del software proceda attraverso una serie di approssimazioni successive. Ad esempio, è possibile identificare da subito dei sottoinsiemi delle funzioni di un'applicazione che possano essere sviluppati subito e consegnati ai committenti (prototipo) al fine di ottenere dei feedback immediati.

### Esempio: compilatore

- Rigore e Formalità: un compilatore è un prodotto critico, poiché se scorretto produrrebbe applicazioni scorrette (per tale motivo è fondamentale definire la sintassi di un linguaggio di programmazione in maniera formale(teoria degli automi e linguaggi formali))
- Separazione degli interessi: è opportuno e utile separare aspetti molto importanti quali la correttezza, l'efficienza, l'amichevolezza dell'interfaccia (i quali non necessitano di essere analizzati insieme).

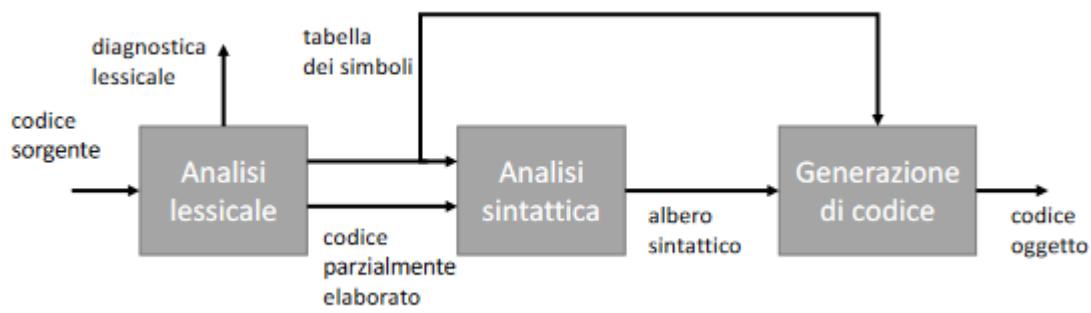


Figura 2.1: Modularità compilatore

- Modularità: ad esempio, è possibile realizzare un modulo per ogni fase principale (analisi lessicale, analisi sintattica, generazione di codice)
- Astrazione: esistono diversi tipi di astrazione in un compilatore, si distingue tra sintassi concreta e astratta al fine di ignorare alcuni dettagli sintattici ininfluenti. In alcuni casi si passa attraverso la produzione di un linguaggio intermedio che astrae dalla macchina specifica su cui avviene la compilazione, al fine di supportare la portabilità del linguaggio.
- Anticipazione del cambiamento: i compilatori sono progettati avendo in mente modifiche frequenti quali nuovi processori, nuovi dispositivi di I/O, estensione del linguaggio sorgente.
- Generalità: spesso i compilatori sono parametrici rispetto all'architettura hardware (si pensi al bytecode per il linguaggio java).
- Incrementalità: spesso i compilatori vengono rilasciati in maniera incrementale, ampliando ad ogni rilascio la copertura rispetto al linguaggio sorgente.

# 3. Modelli di produzione del software

## 3.1 I processi per la produzione di software

Un **processo software** è un insieme di attività che porta alla creazione di un prodotto software. Un **modello** di processo software è una rappresentazione **semplificata** di un processo software: il modello espone le strutture principali del processo, ma non i dettagli delle singole attività. Esistono molti tipi diversi di software e non esiste un unico modello che vada bene per tutti. Il modello giusto dipende dal tipo di software, dalle richieste del cliente, dalle capacità delle persone coinvolte. Agli inizi dell'informatica, lo sviluppo del software era principalmente un lavoro individuale, il modello usato in quei giorni, e che molto spesso i programmatore adottano per sviluppare semplici programmi, è chiamato **code and fix**. Il modello code and fix consiste nell'iterazione di due passi:

- Scrittura del codice;
- Aggiustamento del codice per correggere errori, migliorare la funzionalità, aggiungere nuove caratteristiche;

Gli svantaggi che si incontrano:

- Dopo alcune iterazioni il codice perde la sua struttura originale e le modifiche diventano difficili e costose
- Difficilmente realizza ciò che l'utente finale desidera

Oggi lo sviluppo del software è un'attività professionale che richiede una **pianificazione**. Esistono processi di produzione che sono guidati da piani, detti **plan-driven**:

- tutte le attività sono pianificate in anticipo

- l'avanzamento è misurato rispetto a quanto previsto dal piano

Esistono anche processi di produzione **agili**, in cui la **pianificazione è incrementale e continua** al fine di modificare il processo agevolmente in caso di cambiamenti. I grandi sistemi richiedono dei compromessi tra i due approcci. La produzione di software può essere scomposta in **attività specifiche**, la cui organizzazione dipende dal modello di processo adottato, possiamo raggruppare queste attività in quattro macro attività:

- **Specifico:** vengono definite le funzionalità del software e fissati i vincoli operativi.
- **Sviluppo:** viene sviluppato il software che realizza le specifiche di cui al punto precedente (vado effettivamente a codificare il software).
- **Convalida:** il software sviluppato viene convalidato per garantire ciò che il committente richiede (dobbiamo verificare se il software rispetta le aspettative desiderate)
- **Evoluzione:** il software si evolve per soddisfare le necessità e i cambiamenti dei requisiti del committente (attività dove andiamo a modificare il nostro software a seguito della manifestazione di difetti del nostro prodotto ecc..)

Lo **studio di fattibilità** è una attività preliminare che viene eseguita prima che inizi il processo di produzione vero e proprio. Serve a trovare possibili soluzioni alternative, insieme a una discussione dei compromessi in termini di costi previsti e benefici. Tipicamente si dice se sviluppare un software ex novo, comprarlo da terzi, o abbandonare il progetto perché poco realistico. Richiede dunque un'attenta analisi del problema, specialmente se il suo output è utilizzato per preparare un'offerta ad un potenziale cliente. Viene a volte affiancato da uno **studio di mercato**, che serve a capire se lo sviluppo del software è economicamente sostenibile.

### **Prima Fase:**

Nella fase di **specifico** si parla di **Ingegneria dei requisiti** che sviluppa metodi per ottenere (comprenderli), documentare (scrivere in forma che resti come documentazione), classificare, analizzarli (capire se sono ambigui ecc..). Il progettista deve capire il dominio applicativo e identificare gli **stakeholder**, ovvero tutti coloro che hanno un interesse nel sistema e che saranno responsabili della sua accettazione. Una volta analizzate queste specifiche e tradotte dunque le informazioni le vado a mettere in un **documento di specifica dei requisiti SRS** che potrebbe essere un primo draft tra cliente e venditore. Questo documento dunque dovrà essere:

- Comprensibile
- Preciso (stare attenti alle parole che usiamo, utenti ben definiti con pochi verbi)
- Completo (ossia tutto quello che inseriamo nel documento potrebbe non essere testato)
- Coerente
- Non ambiguo
- Ordinato per importanza (cosa abbastanza importante nei modelli di sviluppo incrementale con PRIORITA')
- Facilmente modificabile in futuro
- Tracciabile (devono avere degli STATI ossia si può tenere traccia di ogni evoluzione da uno stato all'altro (per stato si intende migliorie da una versione ad un'altra))

Il documento dunque può avere due livelli di dettaglio: uno di alto livello, che sarà analizzato e confermato dagli utenti interessati in modo da verificare se coglie tutte le aspettative del committente e uno più dettagliato, utilizzato dai progettisti per sviluppare una soluzione che soddisfi i requisiti. I possibili contenuti che ci devono essere nel documento sono:

- Breve descrizione del **dominio applicativo** e degli obiettivi che dovranno essere raggiunti (utenti interessati (obiettivi e aspettative), principali entità del dominio e loro relazioni)
- Elenco dei **requisiti funzionali**, ovvero cosa deve fare il prodotto, usando notazioni più o meno formali
- Elenco dei **requisiti non funzionali**, quali affidabilità, accuratezza dei risultati, prestazioni, limiti operativi, ecc..
- Elenco dei **requisiti del processo di sviluppo e manutenzione**, quali procedure per il controllo delle qualità, priorità di sviluppo, possibili cambiamenti del sistema

### **Seconda Fase:**

Nella fase di **Sviluppo** si parte dalla **progettazione** che è l'attività attraverso la quale i progettisti strutturano l'applicazione a diversi livelli di dettaglio. La progettazione dell'**architettura** del software consiste nell'identificare la struttura com-

## Table of Contents

<b>Table of Contents .....</b>	<b>ii</b>
<b>Revision History .....</b>	<b>ii</b>
<b>1. Introduction.....</b>	<b>1</b>
1.1 Purpose.....	1
1.2 Document Conventions .....	1
1.3 Intended Audience and Reading Suggestions .....	1
1.4 Product Scope .....	1
1.5 References .....	1
<b>2. Overall Description.....</b>	<b>2</b>
2.1 Product Perspective .....	2
2.2 Product Functions .....	2
2.3 User Classes and Characteristics .....	2
2.4 Operating Environment .....	2
2.5 Design and Implementation Constraints .....	2
2.6 User Documentation .....	2
2.7 Assumptions and Dependencies .....	3
<b>3. External Interface Requirements .....</b>	<b>3</b>
3.1 User Interfaces .....	3
3.2 Hardware Interfaces .....	3
3.3 Software Interfaces .....	3
3.4 Communications Interfaces .....	3
<b>4. System Features .....</b>	<b>4</b>
4.1 System Feature 1 .....	4
4.2 System Feature 2 (and so on) .....	4
<b>5. Other Nonfunctional Requirements .....</b>	<b>4</b>
5.1 Performance Requirements .....	4
5.2 Safety Requirements .....	5
5.3 Security Requirements .....	5
5.4 Software Quality Attributes .....	5
5.5 Business Rules .....	5
<b>6. Other Requirements .....</b>	<b>5</b>
<b>Appendix A: Glossary .....</b>	<b>5</b>
<b>Appendix B: Analysis Models .....</b>	<b>5</b>

Figura 3.1: Table of Contents

plessiva del sistema. Le strutture dati del sistema e le loro rappresentazioni vengono definite nella progettazione del **database**, la progettazione dell'**interfaccia** definisce le interfacce tra i componenti del sistema in maniera non ambigua, in modo che possano essere sviluppati indipendentemente. Infine, vengono ricercati i componenti riutilizzabili e, se non disponibili, vengono progettati nuovi componenti. La produzione di codice e il test dei moduli sono poi le attività che seguono la progettazione, sebbene la programmazione è tipicamente un'attività individuale, può essere soggetta a standard sulla struttura dei programmi, sui commenti, sulle convenzioni di naming, ecc.. Il test serve poi a stabilire l'esistenza di difetti, che possono poi essere risolti mediante debug del codice.

### Terza Fase:

La terza fase è la fase di **convalida** del software che è intesa a mostrare che un sistema è conforme alle sue specifiche e che soddisfa le aspettative del cliente. Questa fase avviene tipicamente in tre fasi:

1. Test dei singoli componenti da parte di chi ha sviluppato il modulo attraverso dei **test case** automatici
2. Integrazione dei componenti per formare il sistema completo, al fine di identificare i problemi con le interfacce e verificare la conformità dei requisiti
3. Esecuzione dei test col cliente, con dati reali, al fine di misurare quanto il software soddisfa le esigenze del cliente.

Gli **alpha test** sono dei test realizzati internamente alla società che sviluppa il software, in condizioni comunque realistiche (ad esempio con molti utenti attivi e con molti dati simulati). I **beta test** sono invece dei test realizzati con un insieme selezionato di utenti reali del sistema e con dati reali. Gli utenti dovrebbero essere motivati, critici e attivi nell'utilizzo del sistema. Una volta in servizio, il sistema potrà evolvere, al fine di:

- risolvere difetti non evidenziati nella fase di convalida
- aggiungere nuove funzionalità
- adattarsi a cambiamenti dell'ambiente operativo
- ecc.

## 3.2 Modello a cascata

Il **modello a cascata** fu introdotto negli anni cinquanta e divenne popolare negli anni settanta (è ancora oggi uno dei modelli di riferimento). Le attività sono strutturate come una **cascata lineare di fasi** (o stage). L'output di una fase diventa l'input della fase successiva, ogni fase è strutturata in sotto-attività che possono essere svolte contemporaneamente. Esistono varie versioni e miglioramenti del modello, ma tutte condividono alcuni aspetti essenziali:

- L'intero processo va pianificato in anticipo.
- Sono sequenziali, ovvero una fase inizia solo dopo che la precedente è terminata.
- Sono basati su documenti (documenti driven).

Alcune istanze del modello consentono dei feedback da una fase a quella strettamente precedente, o ad una fase antecedente qualsiasi

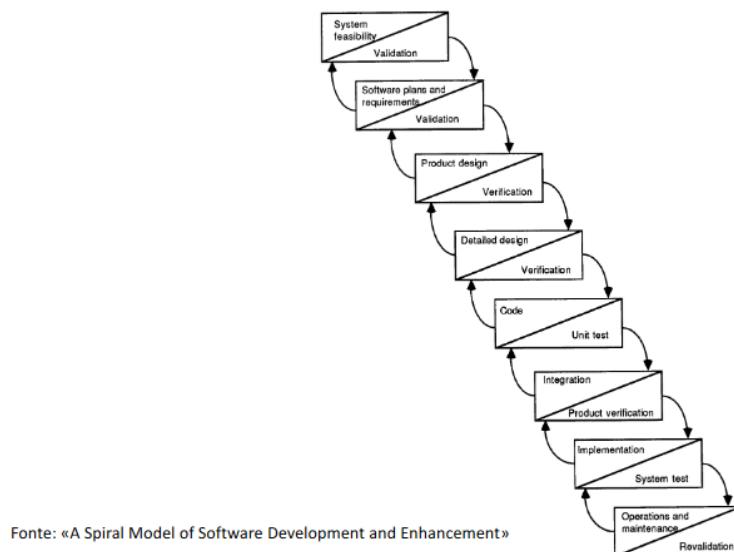


Figura 3.2: Modello a cascata

### Vantaggi:

- Impone una forte disciplina durante tutto il ciclo di vita: adatto a team di lavoro complessi e a progetti di lunghi durata.
- Permette di definire da subito dei **deliverable** associati a date di consegna.

- E' adatto per quei sistemi in cui è possibile e importante conoscere a fondo le specifiche prima di iniziare la fase di progettazione, ad esempio sistemi critici o sistemi embedded.

#### Svantaggi:

- Fasi rigide, requisiti e specifiche di progetto congelati prima della fase di sviluppo
  - interazioni intermedie tra cliente e sviluppatori non previste, dove il cliente approva il documento di specifica dei requisiti, ma questo non garantisce che il prodotto finale lo soddisfi.
  - Eventuali errori in fase di analisi potrebbero non emergere fino al rilascio del prodotto
  - Alcuni requisiti potrebbero non essere più attuali se passa molto tempo tra le prime fasi e la consegna
  - I costi di manutenzione potrebbero lievitare a causa di incomprensioni o cambiamenti non previsti
- Il modello a cascata non è applicabile in particolare a quei contesti in cui i requisiti non sono completamente noti a priori
- L'approccio document driven può risultare troppo burocratizzato e oneroso per alcuni tipi di progetti.

### 3.3 Modelli evolutivi

I modelli **evolutivi** o **incrementali**, sono tipicamente meno plan-driven e più agili. Si produce da subito una versione iniziale del software che viene poi subito esposta agli utenti al fine di ottenere dei **feedback immediati**. La versione iniziale viene poi raffinata attraverso varie versioni, fino a raggiungere una versione stabile finale. Le attività di specifica, sviluppo e convalida si intrecciano tra loro, con **feedback continui** tra una attività e l'altra. Possibili tipologie di feedback sono:

- Feedback da sviluppatori per raffinamento del progetto e dei modelli
- Feedback di processo per raffinamento stime dei tempi e dei costi
- Feedback dal cliente o dal mercato per assegnare o modificare le priorità di sviluppo

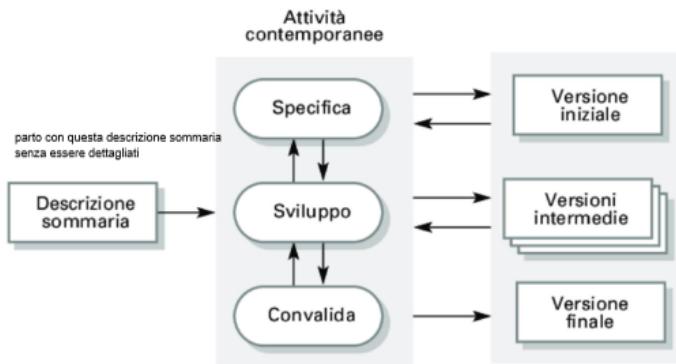


Figura 3.3: Modelli Evolutivi

La strategia di sviluppo dietro un modello evolutivo può essere espressa così:

1. Rilascio di una funzionalità all'utente.
2. Misura del valore aggiunto per il cliente
3. Aggiornamento sia del progetto che degli obiettivi, in base a quanto osservato

Nel modello a cascata il cambiamento si manifesta come un'attività post-sviluppo (manutenzione), mentre in un modello evolutivo i cambiamenti fanno parte invece dello sviluppo e tipicamente si ricorre ad una **consegna incrementale** del prodotto.



Figura 3.4: Consegnna Incrementale

**Vantaggi:**

- Il costo delle modifiche dei requisiti è ridotto con una minore probabilità di fallimento del progetto.

- Risulta più semplice ottenere dei feedback dal cliente con una maggiore probabilità di soddisfare le reali esigenze del cliente e delle parti interessate.
- Il cliente può iniziare a utilizzare il prodotto in anticipo rispetto al rilascio finale (consegna incrementale)

**Svantaggi:**

- Non è economico produrre molta documentazione per ogni incremento del prototipo, infatti il processo può risultare meno visibile da un punto di vista manageriale
- La struttura del programma tende a degradarsi e il codice si complica quando vengono aggiunti nuovi incrementi, infatti occorre una costante riorganizzazione (refactoring) del codice e la comprensibilità del codice diventa una qualità fondamentale per poter essere facilmente modificato.

### 3.4 Sviluppo agile

Oggi le aziende lavorano in un ambiente globale caratterizzato da **cambiamenti rapidi**. La rapidità (produttività) dello sviluppo diventa quindi il requisito più critico per la maggior parte dei sistemi software aziendali. I requisiti dei clienti **cambiano** perché è impossibile prevedere come un sistema influenzerà le pratiche operative, come interagirà con gli altri sistemi e quali operazioni degli utenti dovranno essere automatizzate. I processi di sviluppo plan-driven non sono adatti a questo scenario, poiché frequenti cambiamenti ai requisiti allungano i tempi di sviluppo notevolmente.

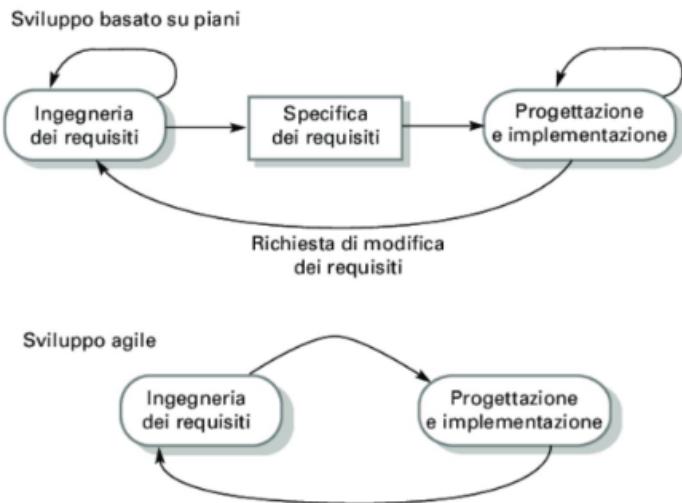


Figura 3.5: Sviluppo Agile

I metodi agili risultano particolarmente utili per:

- Prodotti di piccole o medie dimensioni sviluppati da software house
- Sviluppo personalizzato di sistemi all'interno di un'organizzazione
  - Chiaro impegno da parte del cliente di essere coinvolto nel processo di sviluppo
  - Pochi stakeholder e pochi regolamenti esterni che possono influire sul software

Principio	Descrizione
Coinvolgere il cliente	Coinvolgere i clienti in modo che siano a stretto contatto con il team di sviluppo del software. Il loro ruolo è quello di fornire nuovi requisiti e di dar loro priorità e di valutare ogni incremento del sistema.
Accogliere il cambiamento	Essere preparati al fatto che le feature del prodotto e i relativi dettagli possano cambiare man mano che il team di sviluppo e il product manager assumono maggiori informazioni sul prodotto. Adattare il software in modo che possa far fronte ai cambiamenti man mano che vengono apportati.
Sviluppare e consegnare per incrementi	Sviluppare sempre i prodotti software per incrementi. Testare e valutare ogni incremento man mano che viene sviluppato, e comunicare le modifiche necessarie al team di sviluppo.
Mantenere la semplicità	Concentrarsi sulla semplicità, sia nel software in fase di sviluppo che nel processo di sviluppo. Quando è possibile, fare di tutto per eliminare la complessità dal sistema.
Focalizzarsi sulle persone, non sul processo di sviluppo	Fidarsi del team di sviluppo e non aspettarsi che tutti facciano sempre le cose allo stesso modo. I membri del team dovrebbero essere lasciati liberi di sviluppare modi di lavorare personali, senza essere limitati da processi software prestabiliti.

Figura 3.6: I principi del « Manifesto Agile »

Nello sviluppo di un nuovo prodotto, il prodotto minimo funzionante (**Minimum Viable Product** o **MVP**) è il prodotto con il più alto ritorno sugli investimenti rispetto al rischio. E' una strategia (processo) mirata a:

- evitare di costruire prodotti che i clienti non vogliono
- massimizzare le informazioni apprese sul cliente per ogni euro speso

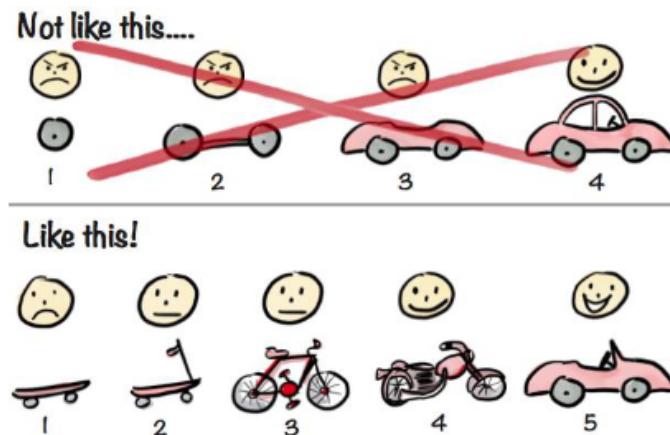


Figura 3.7: Minimal Viable Product

La tecnica di **eXtreme Programming (XP)** è uno degli approcci agili più noti. Sviluppo incrementale supportato attraverso piccole e frequenti release del sistema. Requisiti basati su semplici scenari utilizzati come base per decidere quale funzionalità deve essere inclusa in un incremento del sistema.

XP: pratiche più comunemente adottate

Pratica	Descrizione
Incremental planning (pianificazione incrementale)/ user story	Non esiste un "grande piano" per il sistema. Al contrario, ciò che deve essere implementato (i requisiti) in ogni incremento viene stabilito discutendone con un rappresentante del cliente. I requisiti sono scritti come user story. Le user story da includere in una release sono determinate dal tempo disponibile e dalle loro priorità relative.
Small release (rilasci frequenti) basato sul MVP (minimum viable product)	Per prima cosa si sviluppa l'insieme minimo di funzionalità che fornisca un valore di business. Le release del sistema sono frequenti, e aggiungono gradualmente funzionalità alla release precedente.
Test-driven development (sviluppo guidato dai test)	Anziché scrivere prima il codice e poi i relativi test, gli sviluppatori scrivono prima i test. Ciò aiuta a chiarire che cosa debba effettivamente fare il codice, e a far sì che sia sempre disponibile una versione "testata" del codice. Dopo ogni cambiamento viene utilizzato un <i>automated unit test framework</i> (framework per test di unità automatizzato). Il nuovo codice non deve compromettere il funzionamento del codice già implementato.
Continuous integration (integrazione continua)	Appena un compito è completo, il codice viene integrato nel sistema e da ciò deriva una nuova versione del sistema. Tutti i test di unità di tutti gli sviluppatori vengono eseguiti automaticamente, e devono avere esito positivo prima che la nuova versione del sistema possa essere accettata.
Refactoring (rifattorizzazione)	Refactoring significa migliorare la struttura, la leggibilità, l'efficienza e la sicurezza di programma. Tutti gli sviluppatori sono tenuti a eseguire il refactoring del codice appena vengono identificati possibili miglioramenti. In questo modo il codice resta sempre semplice e di facile manutenzione.

Figura 3.8: XP

Coinvolgimento dell'utente supportato attraverso l'impegno costante del cliente nel team di sviluppo. Persone (anzichè processo) supportate da:

- Programmazione in coppia,
- Possesso collettivo del codice sorgente,
- un processo di sviluppo sostenibile che non richiede tempi di lavoro eccessivamente lunghi

Una **storia utente** descrive in modo strutturato e dettagliato un singolo aspetto che un utente desidera da un sistema software. Le storie utente possono essere ottenute raffinando scenari di utilizzo del sistema di più alto livello. Le storie sono ordinate per priorità dal product owner, iniziando da quelle che possono fornire immediatamente un supporto utile al cliente/committente. Ogni storia utente viene suddivisa in **task**, dove per ciascun task, si fa una stima delle risorse e degli sforzi richiesti per implementarlo. (I task sono le unità principali dell'implementazione). Il problema principale delle storie utente è che potrebbero essere incomplete o poco veritieri.

Si preferisce fare tanto refactoring che architetture complesse rispetto alla quale bisogna fare molte migliorie, infatti il refactoring è una pratica che richiede al team di programmazione di ricercare possibili miglioramenti del software e di implementarli immediatamente (anche se non c'è una immediata necessità). Il refactoring migliora la **struttura** e la **leggibilità** del codice, contrastando il deterioramento strutturale che si verifica naturalmente quando si modifica il software. I test vengono scritti **prima** del codice da testare: le interfacce e le specifiche comportamentali per le funzionalità da sviluppare sono quindi ben definite. Si risolvono subito eventuali ambiguità e omissioni nelle specifiche. Ogni task genera uno o più test, i test sono automatizzati e lo sviluppo non può procedere finché tutti i test non sono stati superati con successo. Il cliente aiuta a sviluppare i **test di accettazione**, si fa uso dei dati del cliente per verificare se il sistema soddisfa le reali esigenze del cliente. Vi sono test automatici eseguiti ogni volta che viene aggiunto nuovo codice.

### 3.5 Scrum

In contrasto rispetto ai processi plan-driven, i metodi agili inizialmente supportavano una pianificazione informale e un'organizzazione autonoma dei team, i quali producevano poca documentazione e lavoravano in cicli di sviluppo molto brevi. In qualsiasi organizzazione, i manager hanno però bisogno di sapere che cosa sta accadendo, se un progetto potrà raggiungere i suoi obiettivi e se il software sarà consegnato in tempo con il budget previsto. Il metodo agile **Scrum** corrisponde ad un framework per organizzare progetti agili e ha una buona visibilità esterna su ciò che sta accadendo durante il processo di sviluppo. Al fine di integrarsi più facilmente con i metodi già esistenti in un'organizzazione, scrum non richiede necessariamente l'uso di specifiche pratiche di sviluppo, come la programmazione a coppie e lo sviluppo con test iniziali. Per questi motivi, scrum è oggi il metodo più utilizzato per la produzione software. Ogni team di sviluppo è un'organizzazione autonoma che dovrebbe avere almeno cinque elementi al suo interno, per consentire la realizzazione dei vari ruoli previsti. Per ragione di efficienza, un team non dovrebbe essere troppo grande, idealmente non dovrebbe avere più di otto persone al suo interno. Un **product backlog** è una lista di elementi (**PBI**) di cui si deve occupare il team. PBI spesso sono classificati in tre stati:

- **Ready for consideration:** PBI di alto livello non ancora raffinati
- **Ready for refinement:** PBI confermati
- **Ready for implementation:** PBI raffinato, pronto per la stima dell'impegno richiesto

1. Come docente, voglio poter configurare il gruppo di strumenti disponibile alle singole classi. (funzionalità)
2. Come genitore, voglio poter visionare i compiti dei miei figli e le valutazioni espresse dai loro insegnanti. (funzionalità)
3. Come insegnante di bambini piccoli, voglio un'interfaccia pittografica per i bambini con capacità di lettura limitata. (richiesta utente)
4. Stabilire criteri per la valutazione di software open source che possano essere usati come base per porzioni di questo sistema. (attività di sviluppo)
5. Refactoring del codice dell'interfaccia utente per migliorare comprensibilità e prestazioni. (miglioramento ingegneristico)
6. Implementare la cifratura per tutti i dati personali degli utenti. (miglioramento ingegneristico)

Figura 3.9: Product Backlog

Ogni prodotto ha un **product owner** che: identifica caratteristiche e requisiti del prodotto, stabilisce le priorità, e rivede continuamente il backlog per tenerlo allineato. Questo product owner può essere un rappresentante del cliente o un manager dell'azienda. L'input del processo scrum è il product backlog, ogni interazione del processo, detta **sprint**, genera un **incremento del prodotto** testato e in teoria pronto per essere rilasciabile al cliente. Ogni sprint ha una durata pre-definita (timeboxed), in genere 2-4 settimane, all'inizio di ogni sprint il product owner stabilisce le priorità nel product backlog, gli elementi non completati nel periodo assegnato vengono restituiti al product backlog e saranno realizzati in uno sprint successivo. Lo **story point** è una unità di misura della «dimensione» di un PBI, che stima le risorse (tempo) necessarie e il rischio associato. Dopo aver stimato la complessità del PBI, si stima la velocità dei team per avere una idea del budget e del tempo necessario.

#### **Planning Poker Game:**

Il team assegna gli story points a ciascun PBI usando la serie di Fibonacci, esempio 1, 2, 3, 5, 8, 13, 21, BIG. Si sceglie il PBI più semplice e meno costoso e lo si pesa 1 story point. Dopo di che si prende un PBI medio e lo si mostra al team descrivendolo, dove ciascun membro del team sceglie poi di nascosto i valori che pensa sia corretto per quel PBI in base all'1 scelto. Se non si riesce a valutare si sceglie "?". Una volta che tutti hanno fatto la loro scelta si girano le carte, se tutti hanno scelto lo stesso valore si assegna il valore concordato, altrimenti le persone che hanno messo il valore più basso e quello più alto si confrontano. Dopo qualche minuto si riprova a giocare e dopo qualche tentativo si dovrebbe arrivare alla convergenza. Un PBI che crea troppa discussione vuol dire che ha bisogno di approfondimenti e quindi non è pronto per gli sprint imminenti. Lo **sprint backlog** è un piano di lavoro per lo sprint (priorità degli elementi, stima risorse). Lo **scrum** è una breve riunione giornaliera in cui il team:

- Esamina l'avanzamento del lavoro

- Stabilisce le priorità del lavoro da svolgere in quel giorno
- Discute eventuali problemi riscontrati nello sviluppo.

Lo scrum è spesso svolto in piedi, per evitare che si dilunghi troppo. Le interazioni giornaliere possono essere coordinate attraverso una lavagna che riporta note sullo sprint backlog, sul lavoro svolto, sull'indisponibilità delle persone, ecc.. Alla fine di ogni sprint si tiene una riunione di verifica, utile a migliorare la qualità del processo, e a capire lo stato di avanzamento del prodotto al fine di revisionare il product backlog.

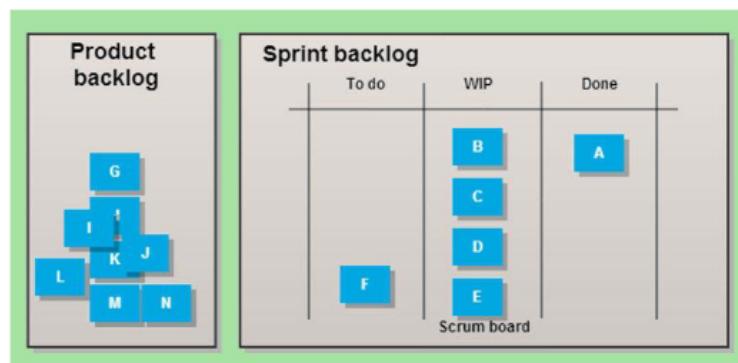


Figura 3.10: Sprint Backlog

Che significa « done » ? Tipicamente:

- Codifica delle funzionalità richiesta completata
- Test di unità scritti e superati
- Test di integrazione superato
- Test prestazionale superato
- Documentazione (minimale) scritta
- Approvato dal product owner

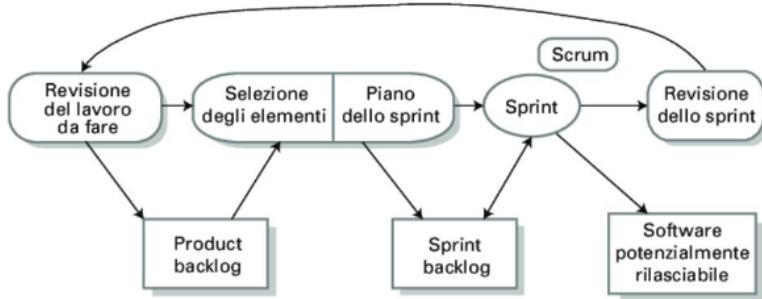


Figura 3.11: Scrum

Lo **ScrumMaster** ha invece la responsabilità di garantire che il processo di scrum sia seguito e sia efficiente, si interfaccia inoltre con il resto dell’organizzazione, riferendo sullo stato del progetto e partecipando alla sua pianificazione.

### 3.6 Problemi e scalabilità dei metodi agili

L’informalità dello sviluppo agile è spesso incompatibile con l’approccio legale alla definizione dei contratti che di solito si usano nelle grandi società. I metodi agili sono più indicati per lo sviluppo di nuovo software, mentre sono poco adatti alla manutenzione per le seguenti problematiche:

- Mancanza di documentazione
- Difficoltà nel mantenere coinvolto il cliente
- Difficoltà nel garantire continuità al team di sviluppo

I metodi agili sono pensati per piccoli team che lavorano fisicamente insieme. Vi sono delle problematiche legate alla produzione di sistemi software su vasta scala

- Team diversi spesso distanti geograficamente hanno difficoltà di comunicazione, e di condivisione della responsabilità.
- Interazione/ integrazione con altri sistemi esterni, molti requisiti riguardano questa interazione e non si prestano allo sviluppo incrementale, attività di configurazione, anch’essa poco adatta allo sviluppo incrementale e all’integrazione continua
- Regole e norme esterne che potrebbero richiedere la produzione di alcuni tipi di documenti e specifici requisiti di conformità

- Tempi lunghi di sviluppo, difficile garantire continuità ai team
- Molti stakeholder con prospettive e obiettivi diversi. Non è possibile, e spesso comunque controproducente, coinvolgerli tutti nello sviluppo.

Esistono vari approcci per adattare i metodi agili alla produzione di sistemi su vasta scala. I punti comuni invece sono:

- Approccio completamente incrementale all'ingegneria dei requisiti non sostenibile
- Un solo product owner non sufficiente (più rappresentanti del cliente)
- Meccanismi di comunicazione tra team usando vari strumenti (telefono, email, wiki, social network)
- Integrazione continua, ovvero la ricostruzione dell'intero sistema ad ogni modifica di un componente, non sostenibile (integrazione frequentemente e release regolari)

Per la produzione di software su vasta scala, il metodo scrum viene adattato per funzionare con più team, ogni team ha un suo product owner e un suo ScrumMaster. Potrebbero esserci un product owner e uno ScrumMaster anche per l'intero progetto, ogni team ha un **architetto di prodotto**, che collabora alla progettazione e al progresso dell'intera architettura. Le date del release dei singoli team vengono allineate, al fine di produrre un sistema dimostrativo completo. Ogni giorno si svolge uno **scrum di scrum**, dove i rappresentanti dei vari team si incontrano per discutere l'avanzamento dei lavori, identificare i problemi e pianificare il lavoro da fare in quel giorno.

## 3.7 DevOps

Lo sviluppo agile del software promuove la collaborazione e introduce brevi cicli di rilascio: rilasciando una versione di software ancora incompleto ma già utilizzabile, il feedback reale dell'utente o del cliente diventa subito disponibile. Questo feedback viene quindi utilizzato per meglio indirizzare lo sforzo di sviluppo, aumentando l'affidabilità del processo, anche il rendimento è aumentato, perché viene minimizzato lo sforzo messo in caratteristiche non necessarie.

Quando le attività di sviluppo, release e supporto del software sono affidate a team di sviluppo distinti, si creano dei ritardari tra la fase di sviluppo e quella di release:

- Strumenti diversi

- Abilità diverse
- Scarsa comprensione dei problemi altrui
- Collo di bottiglia nel passaggio tra sviluppo e rilascio

l’obiettivo è migliorare i livelli di condivisione e di integrazione tra sviluppatori e sistemisti per accellerare i tempi di progettazione, testing e di rilascio delle soluzioni applicative aziendali sia in ambienti tradizionali che in ambienti cloud.

Il **DevOps** è un set di pratiche e di cambiamenti di processo supportati da strumenti automatici, al fine di automatizzare il rilascio del software rispetto alla sua catena di produzione, ottenendo un software di qualità superiore e sicuro in modo estremamente più rapido (diventato noto grazie alla pubblicizzazione di Amazon). Vengono ridisegnati i team di lavoro per fondere sviluppo (development) e produzione (operation). I classici team raggruppati per specializzazioni funzionali (database, front-end, back-end,...) vengono sostituiti da team inter-funzionali normalmente piccoli (otto-dodici persone) in modo che possano mantenere un focus su un aspetto specifico del prodotto complessivo.

Secondo il modello DevOps, sviluppo e produzione sono fusi in un’unica unità in cui i tecnici sono attivi lungo tutto il ciclo di vita dell’applicazione (sviluppo-test-distribuzione-produzione), e acquisiscono una serie di competenze non limitate da una singola funzione. **Prassi DevOps:**

1. Integrazione continua
2. Delivery/Distribuzione continua
3. Microservizi
4. Infrastruttura come codice
5. Monitoraggio e accessi
6. Comunicazione e collaborazione

#### **Integrazione continua:**

Come visto in precedenza, l’integrazione continua è una pratica di sviluppo software in cui gli sviluppatori aggiungono regolarmente modifiche al codice in un repository centralizzato, quindi la creazione di build e i test vengono eseguiti automaticamente. Gli obiettivi principali dell’integrazione continua sono individuare e risolvere i bug con maggiore tempestività, migliorare la qualità del software e ridurre il tempo richiesto per convalidare e pubblicare nuovi aggiornamenti.

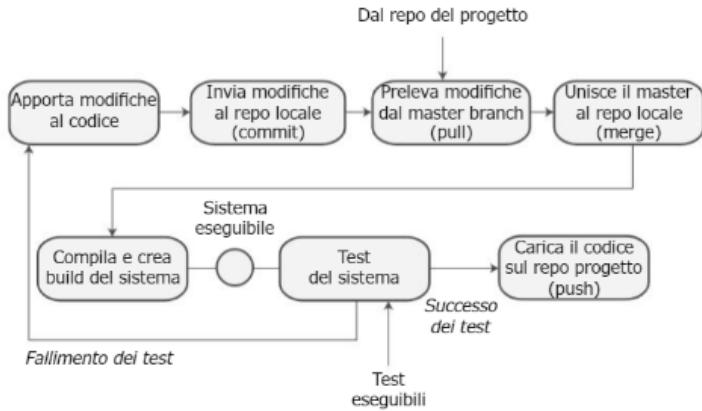


Figura 3.12: Integrazione Continua

### **Delivery continua:**

La **delivery (consegna) continua** prevede che le modifiche al codice vengano applicate a una build, testate e preparate per il rilascio in produzione in modo automatico, estende l'integrazione continua distribuendo tutte le modifiche al codice all'ambiente di testing e/o di produzione dopo la fase di creazione di build. Gli sviluppatori hanno sempre a disposizione una build temporanea pronta per la distribuzione che ha già passato un processo di testing standartizzato.

### **Deployment continuo:**

Sempre più aziende adottano inoltre un approccio di deployment continuo, in cui il software viene rilasciato ad ogni modifica, tipicamente ciò avviene in ambienti cloud, in cui gli aggiornamenti restano trasparenti agli utenti finali. Il modo migliore per semplificare il processo di consegna è quello di utilizzare dei container, infatti per applicazioni che fanno uso di moltissimi container, esistono oggi degli orchestratori (come Kubernetes) che ne facilitano il deployment e la gestione. Ogni nuova versione del codice inoltrata attiva un flusso di lavoro automatizzato che applica a una build, testa e approva temporaneamente l'aggiornamento. La decisione finale per implementare il nuovo software nell'ambiente di produzione attivo dipende poi dallo sviluppatore.

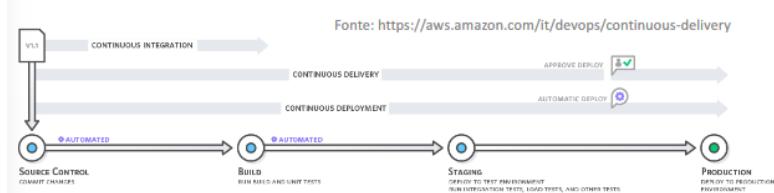


Figura 3.13: Deployment Continuo

### Microservizi:

L'architettura a microservizi è un approccio alla progettazione in cui un'applicazione si basa su un gruppo di servizi di piccole dimensioni, per la loro natura, i microservizi si adattano perfettamente alle pratiche DevOps. Infatti ciascun team full-stack è interamente responsabile di un microservizio, ciascun microservizio (o gruppo di microservizi correlati) viene rilasciato attraverso un container, che può essere così messo in produzione in maniera indipendente dagli altri.

### Infrastruttura come codice:

**Infrastruttura as Code** (IaC) è una prassi per l'automatizzazione via codice delle fasi di provisioning e gestione dell'infrastruttura. Si basa sui sistemi di containerizzazione o su modelli di gestione cloud tramite API, che permettono agli sviluppatori di interagire con l'infrastruttura in modo programmatico e su larga scala, piuttosto che tramite l'impostazione e la configurazione manuale delle risorse. Poiché possono essere definiti tramite codice, l'infrastruttura e i server possono essere distribuiti con la massima rapidità tramite modelli standardizzati, aggiornati con patch e versioni più recenti o duplicati in modo iterabile. Esempi di tecnologie e piattaforme che supportano IaC sono Docker, Amazon AWS, Google Cloud Platform,... Una **macchina virtuale** è l'emulazione di una macchina reale, incluso l'hardware.

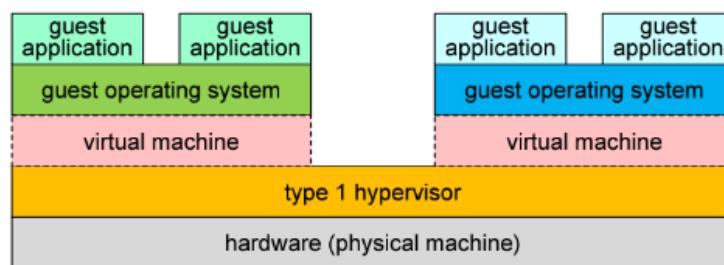


Figura 3.14: Macchine Virtuali

Un'altra forma di virtualizzazione, più efficiente, è quella a livello di OS, basata su container.

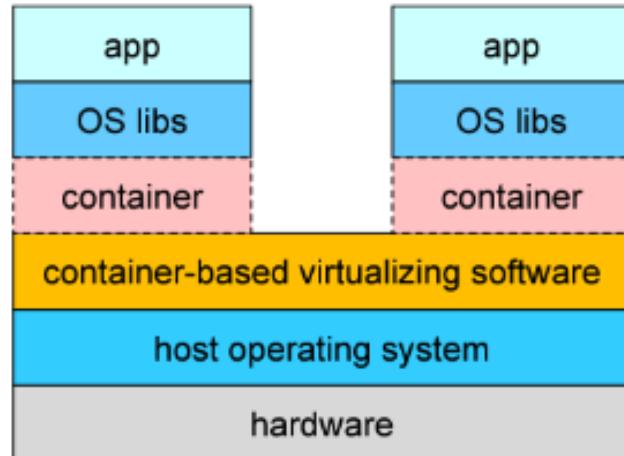


Figura 3.15: Container

Un **container** è un processo « sandboxed » in esecuzione su un computer host isolato da tutti gli altri processi in esecuzione su quel computer host. Una **sandbox** viene implementata eseguendo il software in un ambiente del sistema operativo limitato, controllando così le risorse che un processo può utilizzare. In Linux, tale isolamento sfrutta i kernel namespace e i cgroups.

**Docker** è una tecnologia che semplifica la definizione di containers: i container docker possono essere creati, avviati, arrestati, spostati o eliminati. Un container è portatile, può essere eseguito su macchine locali, macchine virtuali o distribuito nel cloud. Un container in esecuzione utilizza un file system isolato, fornito da una **immagine** del container.

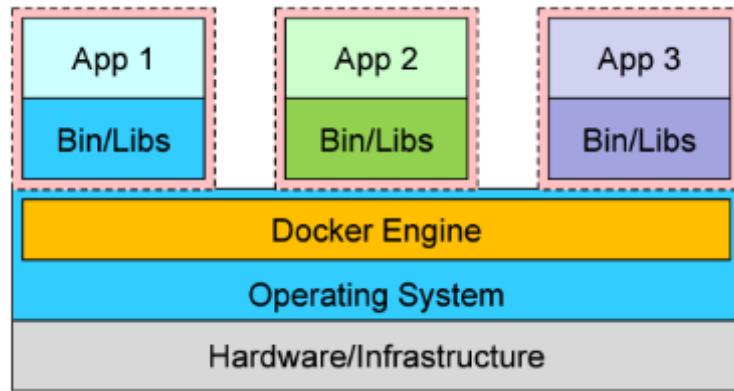


Figura 3.16: Docker

**Macchine Virtuali vs Container:** Le Virtual Machine sono molto flessibili infatti ogni VM ha un proprio OS completo e le proprie applicazioni. I container offrono invece una flessibilità minore, l'OS di un container deve essere compatibile con l'OS dell'host (che è di solito Unix o Linux). Per quanto riguarda l'isolamento, tra VM è completo mentre quello offerto dai container non è invece completo perché infatti i container offrono degli ambienti di esecuzione che sono adeguati per molte applicazioni. Una VM richiede una quantità di risorse maggiori nel sistema host infatti, può introdurre un overhead di esecuzione maggiore, e richiede un tempo maggiore per l'avvio. I container sono invece più "leggeri" delle VM, infatti le prestazioni sono quasi native, richiedendo una quantità minore di risorse (ad esempio un installazione Linux minimale richiede circa un MB, e le librerie di Ubuntu Server richiedono circa cento ottanta MB). E' possibile avere una maggior densità di container per host, ed infine i container possono essere creati e avviati più velocemente.

Ogni container incapsula un singolo servizio software (il rilascio di un'istanza di quel servizio può essere gestito, in modo semplice e affidabile, come la creazione di un container), vi è un isolamento dei guasti e sicurezza (ogni container con il relativo servizio viene eseguito in modo abbastanza isolato), inoltre i container sono leggere (è possibile allocare risorse ai container e ai relativi servizi e scalarli a grana fine, la creazione e l'avvio di un container richiedono in genere da una frazione di secondo a pochi secondi). I container possono essere rilasciati sia nel cloud che on premises, in un proprio data center privato, in particolare i container possono essere rilasciati in una piattaforma per l'orchestrazione di container (ad esempio Kubernetes), sia on premises che nel cloud.

**Kubernetes** fornisce molti strumenti per scalare, mettere in rete, proteggere e mantenere le applicazioni containerizzate.

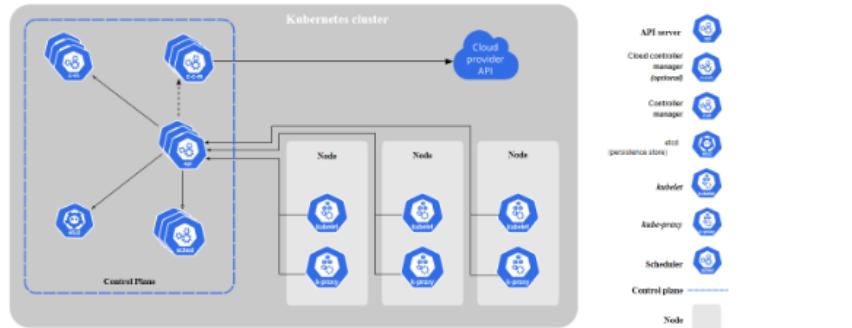


Figura 3.17: Kubernetes

### Monitoraggio e accessi:

Per migliorare costantemente il processo DevOps, sia in termini di qualità del software che di velocità di rilascio, è bene monitorare e misurare costantemente varie dimensioni:

- Misurazioni del processo (dati su sviluppo, testing, deployment)
- Misurazioni del servizio (prestazioni, affidabilità, accettazione da parte degli utenti)
- Misurazioni dell'utilizzo (modi in cui il prodotto è usato)
- Misurazione sul successo dell'azienda (dati sul contributo del prodotto al successo dell'azienda)

Diventa fondamentale monitorare i parametri e i log del sistema per scoprire in che modo le prestazioni di applicazione e infrastruttura influiscono sull'esperienza dell'utente finale. Per capire l'impatto di modifiche o aggiornamenti sugli utenti finali, si devono acquisire dati e log generati da applicazioni e infrastruttura, suddividerli in categorie e analizzarli, esaminando le possibili cause primarie dei problemi o le modifiche impreviste. Anche la creazione di allarmi o l'analisi dei dati in tempo reale sono attività che aiutano a monitorare proattivamente i propri servizi.

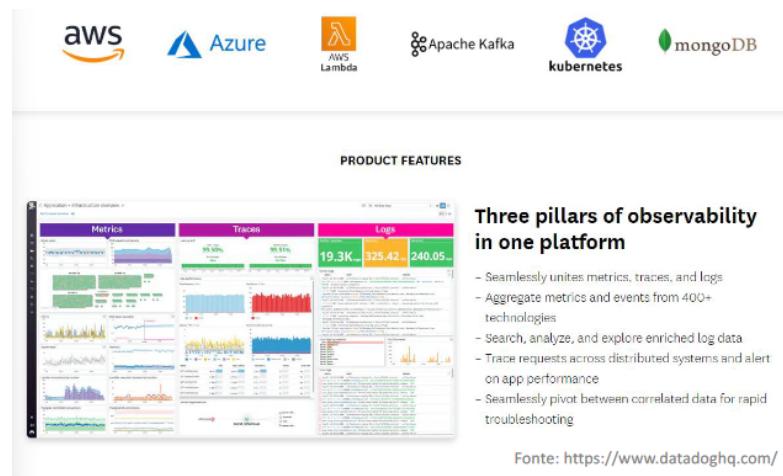


Figura 3.18: Product Features

### Comunicazione e collaborazione:

Uno degli aspetti cruciali dell'approccio DevOps è l'aumento di comunicazione e collaborazione interne all'organizzazione. L'uso di strumenti DevOps è l'automatizzazione dei processi di distribuzione software prevede maggiore collaborazione tramite l'unione di flussi di lavoro e responsabilità in genere distribuiti tra sviluppatori e produzione.

# 4. Esercitazione 1: Controllo delle Versioni

## 4.1 Introduzione alla gestione della configurazione

La **gestione della configurazione** si occupa di politiche, processi e strumenti per gestire un sistema software in evoluzione, per piccoli progetti, è utile tenere traccia delle modifiche effettuate sui vari componenti, mentre per progetti grandi, è essenziale per consentire a team di sviluppatori di lavorare contemporaneamente su più versioni diverse. La gestione della configurazione prevede quattro attività:

1. **Controllo delle versioni:** tenere traccia delle varie versioni dei componenti di un sistema e garantire che le modifiche apportate ai componenti da diversi sviluppatori non interferiscano tra loro.
2. **Costruzione del sistema:** processo che assembla i componenti, i dati e le librerie di programma, compila il codice e crea un sistema eseguibile.
3. **Gestione delle modifiche:** processo che tiene traccia delle richieste di modifica del software da parte dei clienti e degli sviluppatori, valuta i costi e l'impatto di queste modifiche, e decide se e quando implementare le modifiche.
4. **Gestione delle release:** prepara il software per le release esterne e tiene traccia delle versioni del sistema che sono state rilasciate per essere utilizzate dai clienti.



Figura 4.1: Gestione delle configurazione

Gli strumenti di gestione della configurazione possono supportare uno o più compiti rispetto a queste quattro attività, per i grandi sistemi software non c'è mai una sola versione del sistema su cui si lavora attivamente, ma ce ne sono spesso diverse in base alle varie fasi di sviluppo.

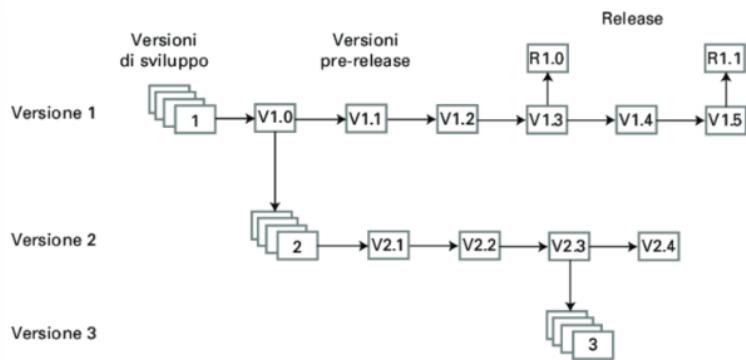


Figura 4.2: Versioni di sviluppo

## 4.2 Controllo delle versioni

La **baseline** di un sistema è la raccolta di versioni di componenti che formano un sistema, una **release** è una baseline rilasciata ai clienti esterni. La **codeline** è invece una serie di versioni di un componente software e di altri elementi di configurazione da cui dipende il componente. La **mainline** è una sequenza di baseline che rappresenta le diverse versioni di un sistema.

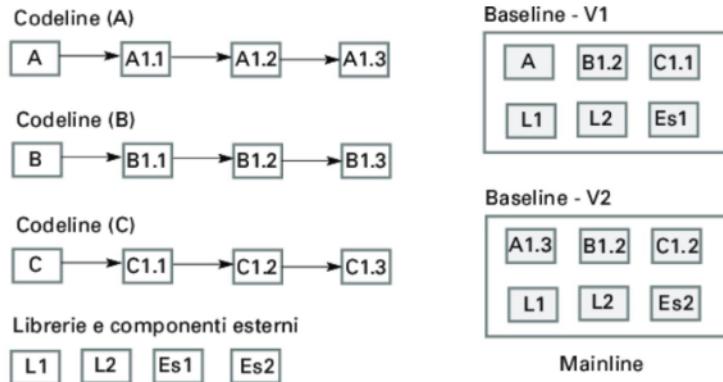


Figura 4.3: Codeline,baseline,mainline

Le varie versioni dei sistemi e dei componenti sono registrate e mantenute con identificativi unici in un database detto **repository**. Un’operazione di **branching** genera una nuova codeline indipendente da una esistente. Un’operazione di **merging** crea una nuova versione di un componente software unendo due versioni distinte in codeline diverse (ad esempio, generate da un branching precedente).

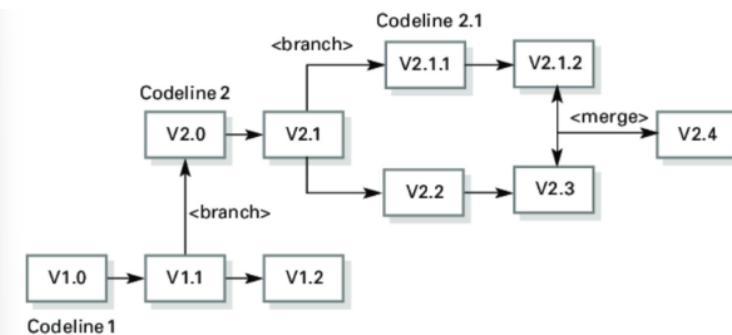


Figura 4.4: Branching e merging

I **sistemi di controllo delle versioni (CV)** identificano, registrano e controllano l’accesso alle differenti versioni dei componenti. Alcune caratteristiche chiave di questi sistemi sono:

- Assegnazione di identificatori univoci alle varie versioni dei componenti
- Registrazione della storia (log) delle modifiche, tenendo traccia di chi e quando ha generato una versione, consentendo l’associazione di commenti e parole chiave.

- Sviluppo contemporaneo e indipendente sullo stesso componente senza interferenze

I sistemi CV mantengono un **repository master**, che può essere condiviso anche da più progetti. Tale repository mantiene le versioni principali dei vari componenti (baseline). Nei sistemi CV **centralizzati** (CVS, SVN):

- Gli sviluppatori copiano i componenti di interesse (**check-out**) nella loro area di lavoro privata (**workspace**).
- Quando hanno completato le modifiche, restituiscono i componenti al repository master (**check-in o commit**)

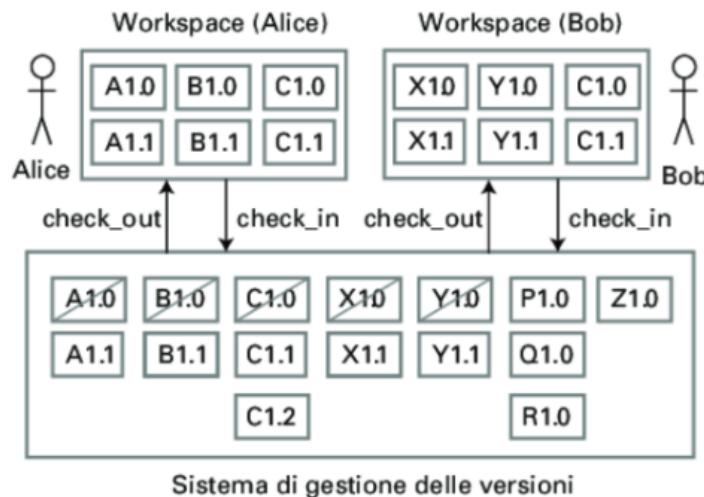


Figura 4.5: Sistemi CV centralizzati

I sistemi CV **distribuiti** mantengono più versioni contemporanee del repository remoto nel proprio workspace. Ogni sviluppatore può confermare le modifiche fatte ai file prima a livello di repository privato, e poi trasmetterle al repository remoto, il quale può accettarle o meno.

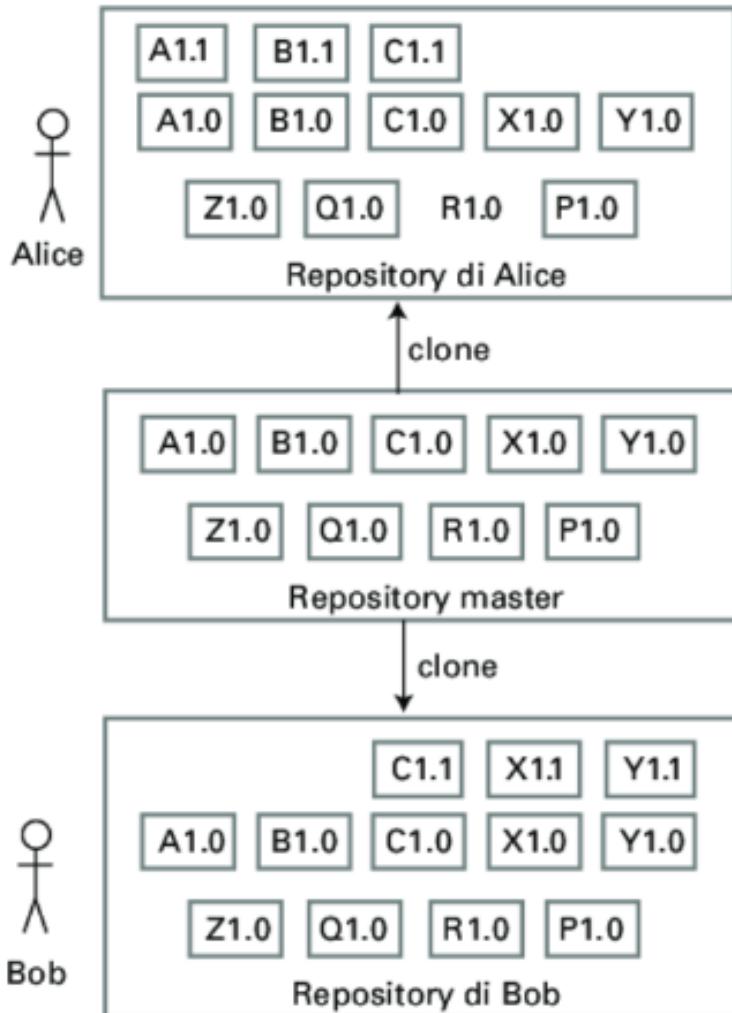


Figura 4.6: Sistemi CV distribuiti

L'approccio distribuito offre alcuni vantaggi:

- Offre un meccanismo di backup, per i dati
- Consente di lavorare anche offline, quando il repository master non è raggiungibile
- Gli sviluppatori possono compilare e testare l'intero sistema sulle loro macchine locali prima di trasmettere le modifiche al repository master.

L'utilizzo di sistemi CV distribuiti porta spesso a operazioni di merging tra versioni dello stesso file provenienti da codeline diverse. Se le modifiche interessano parti

del codice diverse possono essere fuse automaticamente dal sistema, mentre se le modifiche invece si sovrappongono causando conflitti, occorre l'intervento di uno sviluppatore per risolvere le incompatibilità.

### 4.3 GIT

Git è un software di controllo di versione (CV) distribuito (<https://git-scm.com/>). Nacque nel 2005 per lo sviluppo del kernel Linux, ed è oggi uno degli strumenti di CV più diffusi, è open-source e gratuito, può essere usato da linea di comando o tramite client visuali per varie piattaforme. Molti sistemi di CV memorizzano le varie versioni di un file come **sequenze di modifiche** da applicare su un file di partenza. Git memorizza invece delle **snapshot**, ovvero delle « istantanee » dei file. Per ragioni di efficienza, se alcuni file non sono cambiati, Git non li risalva, ma crea semplicemente un collegamento al file precedente già salvato.

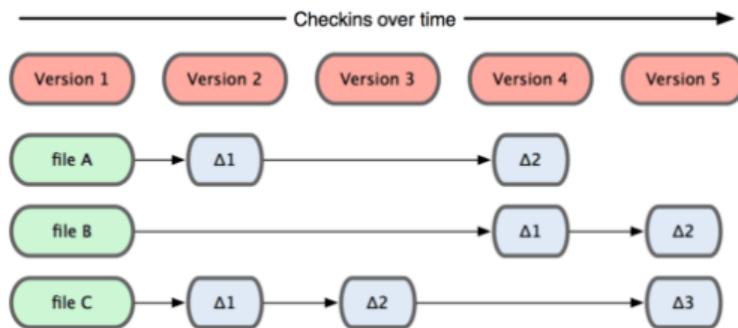


Figura 4.7: Funzionamento incrementale SVN

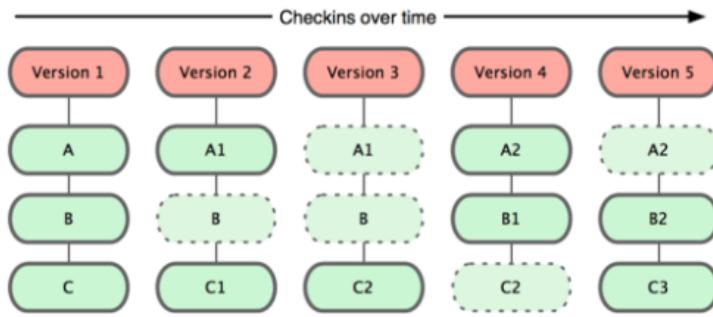


Figura 4.8: Snapshot Git

I file in Git possono essere in tre stati:

- **Committed** (committati): ovvero il file è memorizzato nella copia locale del repository
- **Modified** (modificati): ovvero il file è stato modificato ma non ancora committato nel repository locale.
- **Staged (in stage)**: ovvero un file modificato nella versione corrente e contrassegnato per essere inserito nel prossimo commit.

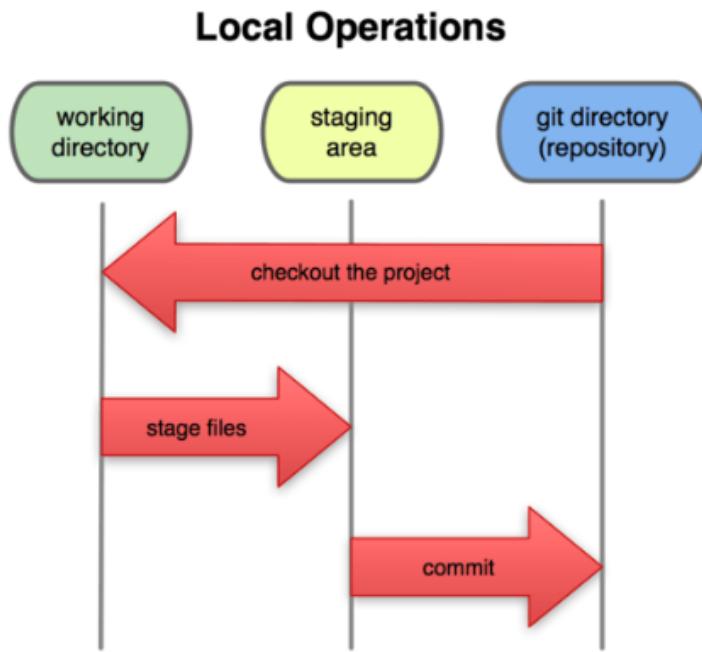


Figura 4.9: Local Operations

La **directory di Git** è dove Git salva i metadati e il database degli oggetti localmente a seguito di una operazione di clone. La **directory di lavoro** è un checkout di una versione specifica del progetto. L'**area di stage** è un file, contenuto generalmente nella directory di Git, con tutte le informazioni riguardanti la prossima operazione di commit. Lo sviluppatore:

1. modifica i file nella sua directory di lavoro
2. fa lo stage dei file modificati
3. fa il commit dei file in stage al fine di creare una snapshot permanente del sistema nella sua directory di Git.

## 4.4 Esercitazione

### 4.4.1 Clone di un repository

Spostarsi sulla directory:

`cd/home/ingsw/E1-GIT`

Clonare un repository da remoto a locale specificando l'origine:

```
$ git clone  
https://github.com/fabriziomontecchiani/ingsw.git  
test.git
```

Figura 4.10: Git clone

Questo comando:

- crea una directory «test»
- dentro questa directory inizializza una directory .git
- scarica tutti i dati dal repository e fa il checkout dell'ultima versione per poterci lavorare.

```
ingsw@ingsw-VirtualBox:~/E1-GIT$ git clone  
https://github.com/fabriziomontecchiani/ingsw.git  
test.git  
Cloning into 'test.git'...  
remote: Enumerating objects: 3, done.  
remote: Counting objects: 100% (3/3), done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-  
reused 0  
Receiving objects: 100% (3/3), done.
```

Figura 4.11: Git clone (2)

#### 4.4.2 Tracciamento dei file

Il file della directory di lavoro possono essere **tracked**, ovvero tracciati dal sistema, oppure **untracked**. I file tracked possono essere **unmodified**, **modified** o **staged**. I file untracked sono tutti gli altri: qualsiasi file nella directory di lavoro che non è presente nell'ultimo snapshot o nell'area di stage.

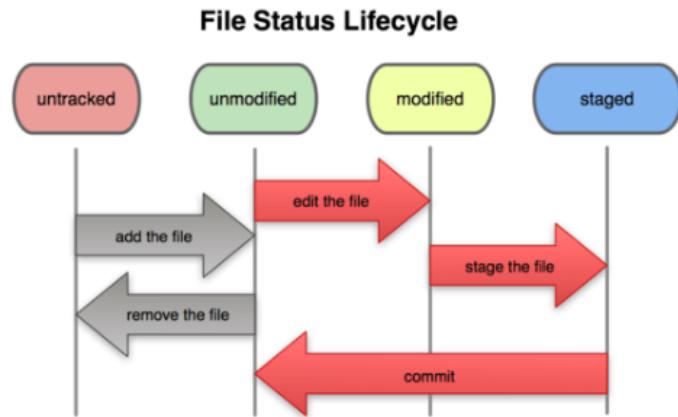


Figura 4.12: Tracciamento dei file

#### 4.4.3 Stato dei file

Spostarsi sulla directory:

`cd test.git`

Per verificare lo stato dei file:

`$ git status`

```

ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

nothing to commit, working tree clean
  
```

Figura 4.13: Stato dei file

#### 4.4.4 Tracciare un file

Creiamo un nuovo file:

`$ vi test.txt`

Per tracciare un nuovo file:

`$ git add test.txt`

```

lingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ vi test.txt
lingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git add test.txt
lingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:  test.txt

```

Figura 4.14: Tracciare un file

#### 4.4.5 Mettere un file in stage

Per vedere quali file sono modificati ma non ancora nello stage (modificare prima TestClass.java)

```

$ nano TestClass.java
$ git diff
diff --git a/TestClass.java b/TestClass.java
index 27c9355..9d6bc当地 100644
--- a/TestClass.java
+++ b/TestClass.java
@@ -1,5 +1,6 @@
 public class TestClass{
     public void testMethod(){
-        return null;
+        int i=0;
+        return null;
    }
}

```

Figura 4.15: Mettere un file in stage

Per mettere un file tracciato in stage si usa ancora add.

```

$ git add TestClass.java

lingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git status

On branch main
Your branch is up-to-date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:  TestClass.java
    new file:  test.txt

```

Figura 4.16: Mettere un file in stage

#### 4.4.6 Commit delle modifiche

Per committare i file in stage

```
$ git commit -m "Test per commit"
```

```
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git commit -m  
"Test per commit"  
[main 641fad7] Test per commit  
2 files changed, 2 insertions(+), 1 deletion(-)  
create mode 100644 test.txt
```

Figura 4.17: Commit delle modifiche

Per committare i file modificati saltando l'area di stage:

```
$ nano TestClass.java  
$ git commit -a -m "Test per commit senza stage"
```

#### 4.4.7 Rimuovere file

Per rimuovere un file va prima rimosso dall'area di stage e poi committato:

```
$ git rm test.txt  
$ git commit -m "File rimosso"
```

```
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git rm test.txt  
rm 'test.txt'  
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git commit -m "File  
rimosso"  
[master 12dc6ad] File rimosso  
1 file changed, 0 insertions(+), 0 deletions(-)  
delete mode 100644 test.txt
```

Figura 4.18: Rimozione file

#### 4.4.8 Cronologia dei commit

Per vedere la cronologia dei commit:

```
$ git log
```

```
commit bba0fbbb5d0e54d8796f3a1917de98b4d805c51c (HEAD -> main)
Author: ingsw <ingsw@IngSW2023.myguest.virtualbox.org>
Date:   Thu Aug 10 17:42:45 2023 +0200
    File rimosso

commit 457c513c282a2505f1426075088a1899e0e5b44b
Author: ingsw <ingsw@IngSW2023.myguest.virtualbox.org>
Date:   Thu Aug 10 17:40:40 2023 +0200
    Test per commit senza stage

...
```

Figura 4.19: Cronologia dei commit

#### 4.4.9 GITK

Il comando *gitk* lancia uno strumento grafico per visualizzare il log di GIT

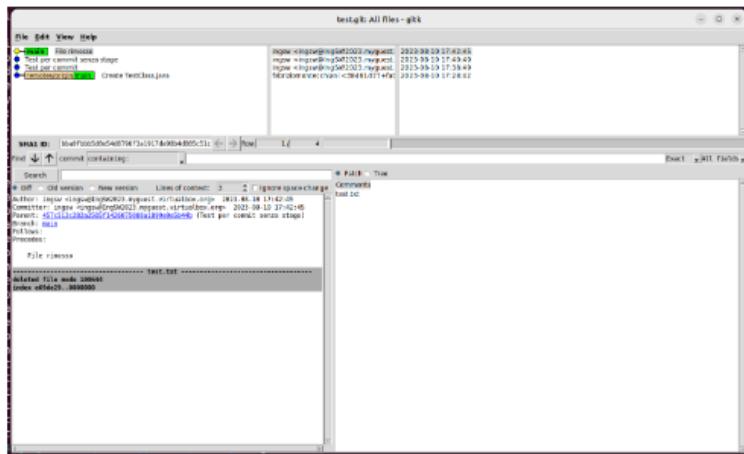


Figura 4.20: Gitk

Diff cont gitk:

```

Diff Old version New version Lines of context: 3 Ignore space change
Author: ingsw <ingsw@IngSW2023.myguest.virtualbox.org> 2023-08-10 17:38:49
Committer: ingsw <ingsw@IngSW2023.myguest.virtualbox.org> 2023-08-10 17:38:49
Parent: 2405507015352f7883a84077584718a17e29ed23 (Create TestClass.java)
Child: 457c513c282a2505f1426075008a1899e0e5b4db (Test per commit senza stage)
Branch: main
Follows:
Precedes:

Test per commit

----- TestClass.java -----
index 27c9355..9d6bc4e 100644
@@ -1,5 +1,6 @@
 public class TestClass{
     public void testMethod(){
-     return null;
+     int i=0;
+     return null;
 }

```

Figura 4.21: Gitk

Ad ogni commit, Git immagazzina un oggetto che contiene varie informazioni (autore, commenti,...) e che punta ai commit **parent**:

- zero parent per il primo commit
- un parent per un commit normale
- più parent per un commit che risulta da un merging di due o più branch

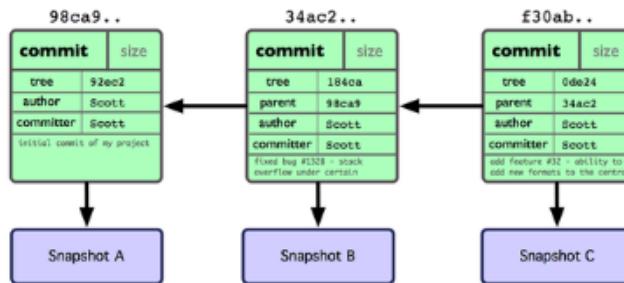


Figura 4.22: Parent nei commit

#### 4.4.10 Branching

Git incoraggia un metodo di lavoro che sfrutta le ramificazioni e le unioni frequenti, anche molte al giorno. Un ramo (branch) è semplicemente un puntatore ad un commit. Il nome del ramo principale in Git è **master**.

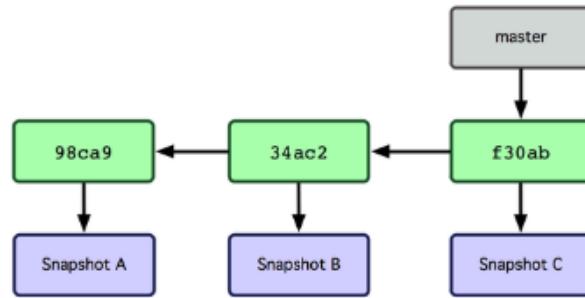


Figura 4.23: Branching

Per creare un nuovo branch:

`$ git branch testing`

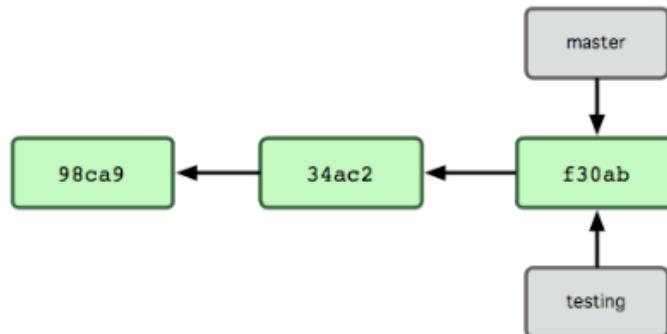


Figura 4.24: Branching

Git mantiene uno speciale puntatore al ramo locale corrente chiamato **HEAD**.

Per cambiare ramo:

`$ git checkout testing`

```

ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git branch testing
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git checkout testing
Switched to branch 'testing'

```

Figura 4.25: Branching

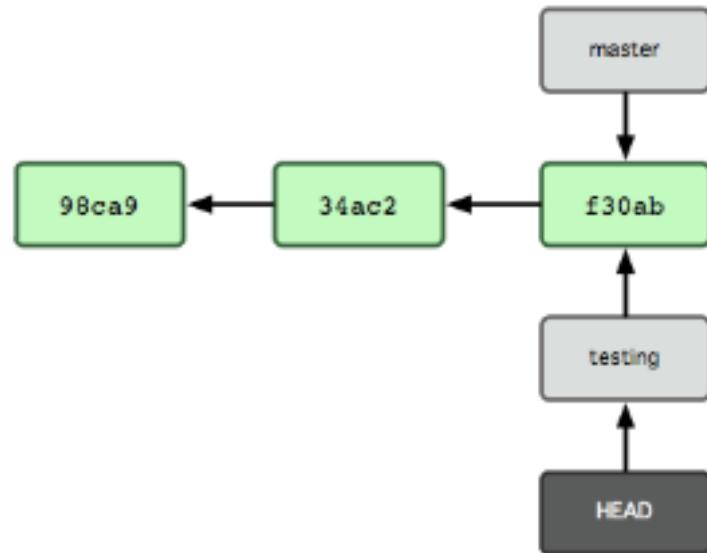


Figura 4.26: Branching

```

ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ vi file.txt
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git add file.txt
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git commit -a -m
"aggiunto un file"
[testing 6778b26] aggiunto un file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file.txt

```

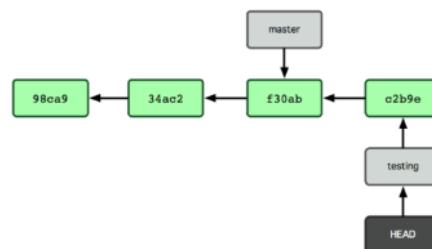


Figura 4.27: Branching

```
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)
```

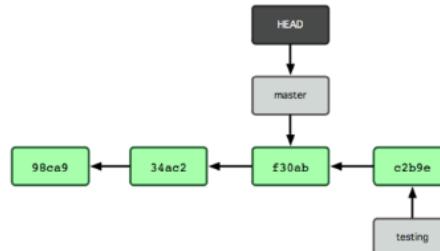


Figura 4.28: Branching

Questo comando ha fatto due cose:

1. ha spostato il puntatore HEAD indietro per puntare al ramo main
2. ha riportato i file nella directory di lavoro allo stato in cui si trovavano in quel momento

I cambiamenti fatti da questo punto in poi saranno separati da una versione più vecchia del progetto. Essenzialmente riavvolge temporaneamente il lavoro fatto nel ramo testing e permette di «muoversi» in una direzione differente.

```
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ vi newfile.txt
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git add newfile.txt
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git commit -a -m
  'fatte altre modifiche'
[master 08f3594] fatte altre modifiche
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 newfile.txt
```

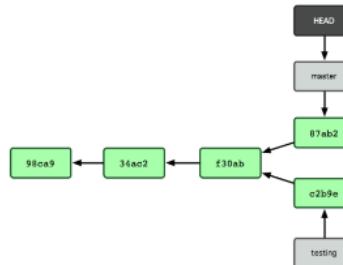


Figura 4.29: Branching

#### 4.4.11 Merging

```
$ git merge testing
```

```
ingsw@ingsw-VirtualBox:~/E1-GIT/test.git$ git merge testing
Merge made by the 'ort' strategy.
 file.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
```

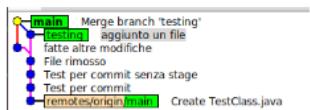


Figura 4.30: Merging

#### 4.4.12 Listare e rimuovere branch

```
$ git branch
* main
  testing
```

```
$ git branch -d testing
Ramo testing eliminato (era ed6610d).
```

```
$ git branch
* main
```

Figura 4.31: Listare e rimuovere branch

#### 4.4.13 Esempi di merging

```
$ git checkout master  
$ git merge hotfix
```

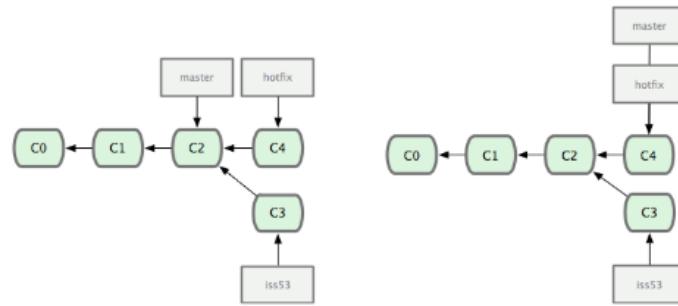


Figura 4.32: Esempi di merging

```
$ git checkout master  
$ git merge iss53
```

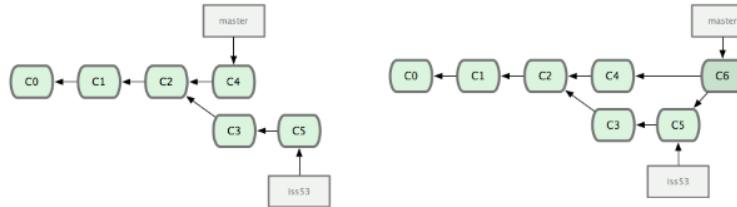


Figura 4.33: Esempi di merging

#### 4.4.14 Conflitti

Se si modifica la stessa parte di uno stesso file in modo differente in due rami che vengono fusi assieme, Git non è in grado di fonderli in modo «pulito». I conflitti possono essere risolti «manualmente» o con l'aiuto di tool grafici com *git mergetool*

#### **4.4.15 Scaricare le modifiche da server remoto**

Per scaricare le ultime modifiche dal server remoto, senza fare il merge con i dati del repository locale:

*\$ git fetch origin*

Per scaricare le ultime modifiche dal server remoto, facendo il merge con i dati del repository locale (fetch + merge):

*\$ git pull origin*

#### **4.4.16 Push su server remoto**

Per fare il push di un branch (in questo caso master) sul repository remoto principale: serve autenticazione

*\$ git push origin master*

#### **4.4.17 Github e Bitbucket**

GitHub e Bitbucket sono noti servizi di hosting per progetti software. Implementano un sistema CV basato su Git e gestiscono il controllo degli accessi. Offrono anche altri servizi quali bug tracking, feature requests, task management, wiki pages, ecc. Hanno diversi piani tariffari, sia gratuiti che a pagamento.

# 5. Ingegneria dei requisiti

## 5.1 Introduzione all'ingegneria dei requisiti

I **requisiti** di un sistema sono la descrizione dei servizi che il sistema deve fornire e dei suoi vincoli operativi. L'**ingegneria dei requisiti** è il processo di ricerca, analisi, documentazione e verifica di questi servizi e vincoli. Serve una separazione netta tra diversi livelli di descrizione dei requisiti:

- **Requisiti utente:** dichiarano, nel linguaggio naturale e con diagrammi, quali servizi il sistema dovrebbe fornire e i vincoli sotto cui deve operare, il livello di dettaglio può essere più o meno astratto, in base alle esigenze del cliente
- **Requisiti sistema:** descrizioni più dettagliate delle funzioni, dei servizi e dei vincoli operativi del sistema software, sono raccolti in un documento formale, spesso chiamato **specifiche funzionale**.

*Definizione dei requisiti dell'utente*

*1. Il sistema Mentcare genererà mensilmente dei rapporti che riportano il costo dei farmaci prescritti da ciascuna clinica durante il mese.*

*Specifiche dei requisiti del sistema*

*1.1 Nell'ultimo giorno lavorativo di ogni mese, dovrà essere generata una sintesi dei farmaci prescritti, il loro costo e le cliniche che li hanno prescritti.*

*1.2 Il sistema genererà il rapporto per la stampa dopo le 17:30 dell'ultimo giorno lavorativo di ciascun mese.*

*1.3 Sarà creato un rapporto per ciascuna clinica; dovranno essere elencati i nomi dei singoli farmaci, il numero totale di prescrizioni, il numero delle dosi prescritte e il costo totale dei farmaci prescritti.*

*1.4 Se i farmaci sono disponibili in dosi unitarie differenti (per esempio, 10 mg, 20 mg ecc.), dovranno essere creati dei rapporti separati per ciascuna dose unitaria.*

*1.5 L'accesso ai rapporti dei costi dei farmaci dovrà essere limitato agli utenti autorizzati, come indicato nella lista di controllo degli accessi.*

Figura 5.1: Esempio

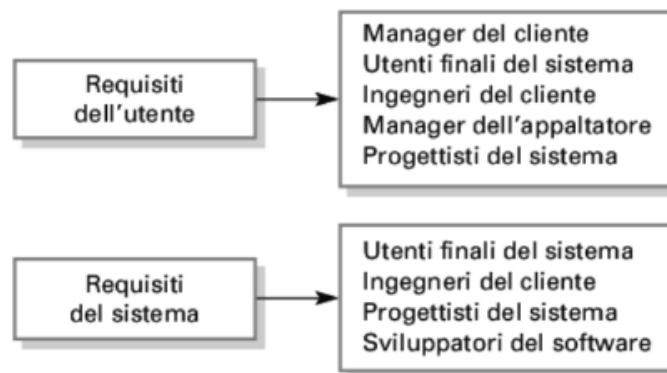


Figura 5.2: Tipologie di lettori

I requisiti dei sistemi software possono essere divisi in due gruppi:

- **Requisiti funzionali:** definiscono i servizi che il sistema deve offrire, ossia come il sistema reagisce a determinati input e come si comporta in certe situazioni e cosa il sistema non dovrebbe fare. La specifica dei requisiti funzionali dovrebbe essere **completa e coerente**.

- **Requisiti non funzionali:** definiscono dei vincoli sulle funzioni o sui servizi offerti dal sistema. Sono definiti sull'intero sistema anziché sui singoli componenti, le proprietà del sistema sono: affidabilità, tempi di risposta e uso della memoria ecc..., mentre per i vincoli di implementazione: capacità di dispositivi di I/O, rappresentazione di dati per interfacce con altri sistemi. Un requisito non funzionale può influire sull'intera architettura del sistema (Esempio: vincoli sulle prestazioni potrebbero influire sull'architettura del sistema richiedendo la minimizzazione della comunicazione tra componenti). Un requisito non funzionale può generare vari requisiti funzionali (Esempio: un requisito legato alla protezione dei dati potrebbe aggiungere requisiti legati all'autenticazione degli utenti).
  - **Requisiti di prodotto:** specificano o limitano il comportamento del software (es. prestazioni, affidabilità)
  - **Requisiti organizzativi:** requisiti generali del sistema che derivano da politiche e procedure del cliente o del produttore (es. standard da rispettare, strumenti di sviluppo da utilizzare).
  - **Requisiti esterni:** derivati da fattori esterni al sistema e al suo processo di sviluppo (es. normativa, etica)

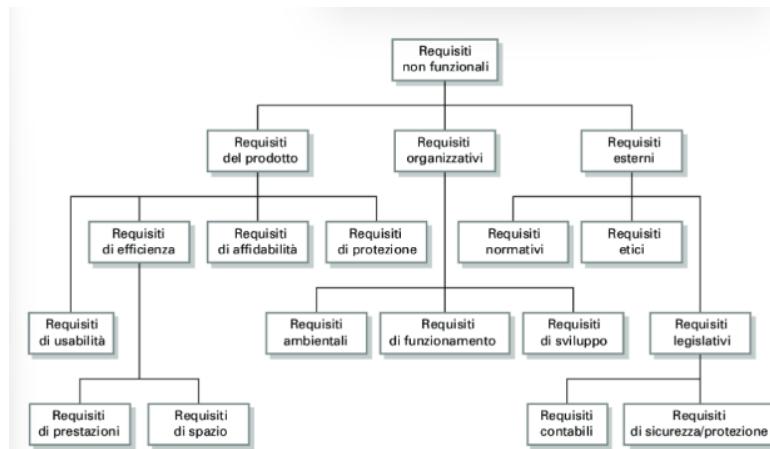


Figura 5.3: Requisiti non funzionali

#### Requisiti del prodotto

Il sistema Mentcare dovrà essere disponibile a tutte le cliniche durante l'orario normale di lavoro (lun-ven, 08:30-17:30). Il periodo di inattività all'interno dell'orario normale di lavoro non dovrà superare 5 secondi durante la giornata.

#### Requisiti organizzativi

Gli utenti del sistema Mentcare dovranno identificarsi utilizzando la carta di identità rilasciata dall'autorità del servizio sanitario.

#### Requisiti esterni

Il sistema dovrà implementare le disposizioni sulla privacy dei pazienti definite nello standard HStan-03-2006-priv.

Figura 5.4: Esempio requisiti non funzionali

I requisiti non funzionali andrebbero descritti in modo che possano essere verificati in maniera oggettiva, ossia servono misure per testare se il sistema soddisfa o no i requisiti funzionali.

L'attività chiave dell'ingegneria dei requisiti sono:

- **Deduzione e analisi:** scoperta dei requisiti attraverso l'interazione e con gli stakeholder
- **Specificazione:** conversione dei requisiti in una forma standard
- **Convalida:** controllo che i requisiti definiscano realmente il sistema voluto dal cliente

Attività interacciate e spesso organizzate secondo un processo iterativo attorno a una spirale (modello a spirale), il cui output è un documento dei requisiti del sistema:

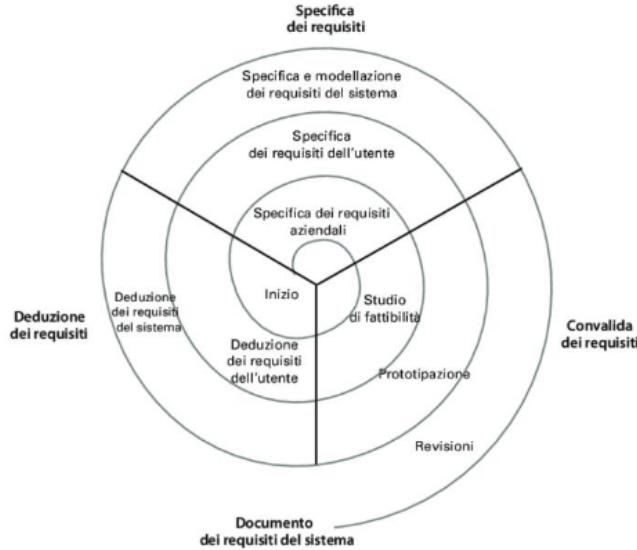


Figura 5.5: Modello a spirale

Il modello a spirale può essere adattato modificando il numero di « giri » sulla spirale, ovvero uscendo eventualmente prima di aver raccolto tutti i requisiti. La fase di prototipazione può inoltre essere sostituita con un metodo di sviluppo agile, in modo da parallelizzare lo sviluppo del sistema alla raccolta dei requisiti.

## 5.2 Deduzione dei requisiti

Durante la fase di deduzione dei requisiti gli ingegneri lavorano con gli stakeholder per scoprire informazioni su

- Dominio di applicazione
- Attività, servizi e caratteristiche che il sistema dovrebbe fornire
- Prestazioni richieste dal sistema
- Vincoli hardware
- Altro

Per quanto riguarda invece gli stakeholder, possiamo affermare che spesso

- non sanno cosa vogliono

- lo sanno solo in termini generici (difficoltà di comunicazione, richieste irrealizzabili, ecc...),
- Stakeholder diversi hanno spesso esigenze diverse che vanno conciliate.

Bisogna dunque andare a distinguere tra « desideri » e « bisogni ». I requisiti del sistema potrebbero essere influenzati da fattori politici, per esempio i manager potrebbero inserire requisiti che aumentano il loro potere. L'ambiente economico e aziendale è dinamico e i requisiti possono cambiare rapidamente.



Figura 5.6: Deduzione dei requisiti

Le **interviste** richiedono di parlare con gli stakeholder per sapere che cosa fanno:

- Intervista **chiusa**: set di domande predefinite
- Intervista **aperta**: non vincolata ad un insieme di domande prestabilito

Le interviste infatti vanno integrate con documenti esistenti che descrivono i processi e i sistemi esistenti, e le osservazioni sul campo raccolte dagli ingegneri. Però gli stakeholder spesso usano una terminologia e un gergo specifici del loro dominio, potrebbero anche dare per scontate alcune loro conoscenze, o non riuscire a spiegarle correttamente, inoltre potrebbero non dire tutta la verità, specialmente per quanto riguarda i reali ruoli e rapporti all'interno dell'organizzazione. L'**etnografia** è una tecnica di osservazione utilizzata per capire i processi operativi e per derivare i requisiti del software a supporto di tali processi. Valutare come i fattori sociali e organizzativi influenzano il funzionamento pratico di un sistema

è fondamentale per aumentare le probabilità di accettazione e di reale utilizzo di un software. Un analista si immerge nell’ambiente di lavoro in cui il sistema sarà usato, osserva il lavoro quotidiano e prende nota dei compiti in cui i partecipanti sono coinvolti. Utile per scoprire requisiti che derivano dal modo in cui le persone lavorano realmente, anziché dal modo in cui le definizioni di processo dicono che dovrebbero lavorare. Porta alla luce requisiti che derivano dalla cooperazione e dalla consapevolezza delle attività di altre persone.

### 5.3 Specifica dei requisiti

L’obiettivo di questa fase è scrivere i requisiti dell’utente e del sistema in un documento, descrivere soltanto il comportamento esterno del sistema e i suoi vincoli operativi, senza toccare aspetti di progettazione o implementazione. Le specifiche possono essere scritte in linguaggio naturale, per ridurre il rischio di incomprensioni possiamo:

- Usare un formato di scrittura standard e coerente
- Formattare il testo usando grassetto e corsivo
- Evitare acronimi e gerghi tecnici
- Associare una logica ad ogni requisito che spiega perché il requisito è utile

3.2 Il sistema deve misurare il livello degli zuccheri nel sangue e rilasciare l’insulina, ogni 10 minuti (variazioni degli zuccheri nel sangue sono relativamente lente, quindi non sono necessarie misure più frequenti; misure meno frequenti potrebbero portare a livelli di zuccheri inutilmente elevati).

3.6 Il sistema deve eseguire una routine di auto-test ogni minuto con le condizioni da provare e le azioni associate definite nella Tabella 1 (una routine di auto-test può scoprire problemi nel software e nell’hardware e avvisare l’utente che la pompa potrebbe non funzionare correttamente).

Figura 5.7: Esempio di specifiche dei requisiti in linguaggio naturale

Le specifiche possono anche essere scritte usando dei linguaggi strutturati (ad es: creare una scheda per ogni requisito, ogni scheda può contenere vari campi fissi che vanno compilati, come la descrizione della funzione, l’input, l’output, eventuali dipendenze da altri requisiti. I linguaggi strutturati sono utili per mitigare le fonti di ambiguità derivanti dall’uso del linguaggio naturale. I modelli agili fanno uso di **storie utente** per descrivere le specifiche, infatti sono una forma di linguaggio strutturato che tipicamente riporta le seguenti informazioni:

- As a <RUOLO>
- I want <FUNZIONE>

- so that <VALORE>

I **casi d'uso** utilizzano una notazione grafica e un testo strutturato per descrivere le interazioni tra utenti e sistema. Sono parte del linguaggio **UML** (Unified Modeling Language), ogni caso d'uso rappresenta uno **scenario di interazione** e identifica gli **attori** coinvolti (persone o sistemi). L'interazione è poi descritta con testi o altri diagrammi.



Figura 5.8: Esempio di singolo caso d'uso

Il documento dei requisiti del software (software requirement specification, o SRS) include i requisiti dell'utente e può includere anche i requisiti di sistema. Nei metodi di sviluppo agile il documento dei requisiti è spesso assente, e sostituito con schede o storie utente raccolte incrementalmente.

## 5.4 Convalida dei requisiti

In questa fase si verifica che se i requisiti definiscono il sistema realmente voluto dal cliente, I **controlli di validità** verificano se i requisiti riflettono le reali esigenze degli utenti del sistema, i **controlli di consistenza** verificano che non ci siano conflitti o contraddizioni tra i requisiti, i **controlli di completezza** verificano che il documento dei requisiti contenga tutte le funzioni e i vincoli operativi richiesti dagli utenti del sistema. I **controlli di现实ismo** verificano che i requisiti siano compatibili con le tecnologie disponibili e rispettando le tempistiche e il budget. I **controlli di verificabilità** cercano di stabilire se i requisiti sono scritti in modo da essere verificabili attraverso dei test. Esistono varie tecniche di validazione:

- La **revisione** prevede che un team di revisori esamini tutti i requisiti in modo sistematico per capire se sono consistenti
- La **prototipazione** prevede lo sviluppo di un modello eseguibile del sistema per vedere se esso soddisfa le loro necessità e aspettative, al fine di capire se i requisiti sono validi e completi.

- La generazione di **test case** serve a capire se i requisiti sono verificabili.

# 6. Progettazione Architetturale

## 6.1 Introduzione

La **progettazione architetturale** è il primo passo nel processo di progettazione del software. Rappresenta un passaggio importante tra l'ingegneria dei requisiti e la progettazione. Identifica i principali componenti strutturali di un sistema e le loro relazioni. L'output di questa fase è un **modello dell'architettura** complessiva del sistema.

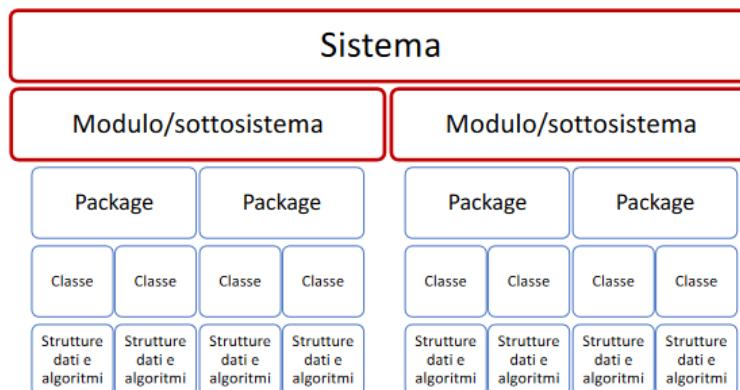


Figura 6.1: Progettazione architetturale

In un processo di sviluppo agile, la progettazione architetturale è spesso svolta nelle fasi iniziali (ad esempio, può essere inclusa in uno sprint Scrum iniziale). Documentare chiaramente l'architettura:

- Favorisce la discussione tra gli stakeholder

- Anticipa l'analisi di alcuni problemi
  - Supporta il riuso di schemi e componenti

Le architetture sono spesso modellate in modo informale utilizzando semplici diagrammi a blocchi:

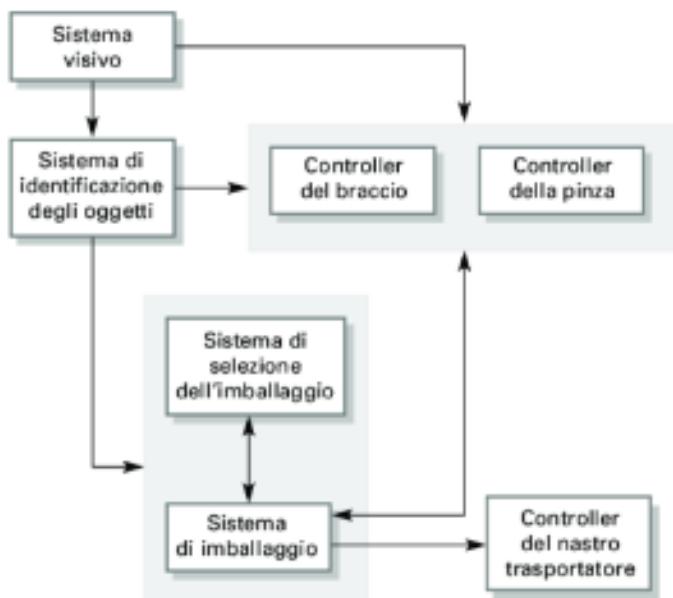


Figura 6.2: Progettazione architetturale (2)

Esistono notazioni più rigorose che consentono di:

- mostrare tutti i componenti,
  - mostrare tutte le interfacce dei componenti,
  - mostrare le relazioni tra componenti.

I primi sono che favoriscono rigore e formalismo, mentre per quanto riguarda i controlli sono spesso troppo complicate per persone non esperte ed economicamente poco vantaggiose. L'architettura di un sistema dipende molto dai requisiti non funzionali, per esempio se le **prestazioni** sono un requisito critico, l'architettura potrebbe essere progettata per localizzare le operazioni critiche all'interno di pochi componenti, riducendo la comunicazione. Oppure se la **manutenibilità** è un requisito critico, l'architettura potrebbe prevedere molti componenti piccoli e autonomi, facili da modificare.

Per esempio, la seguente architettura privilegia la manutenibilità:

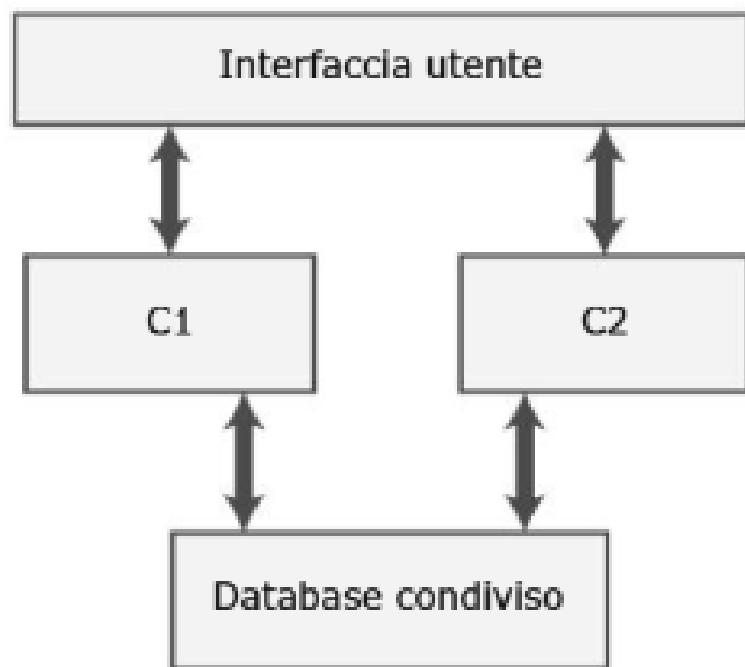


Figura 6.3: Esempio di architettura che privilegia la manutenibilità

Mentre la seguente architettura privilegia invece le prestazioni e la resilienza:

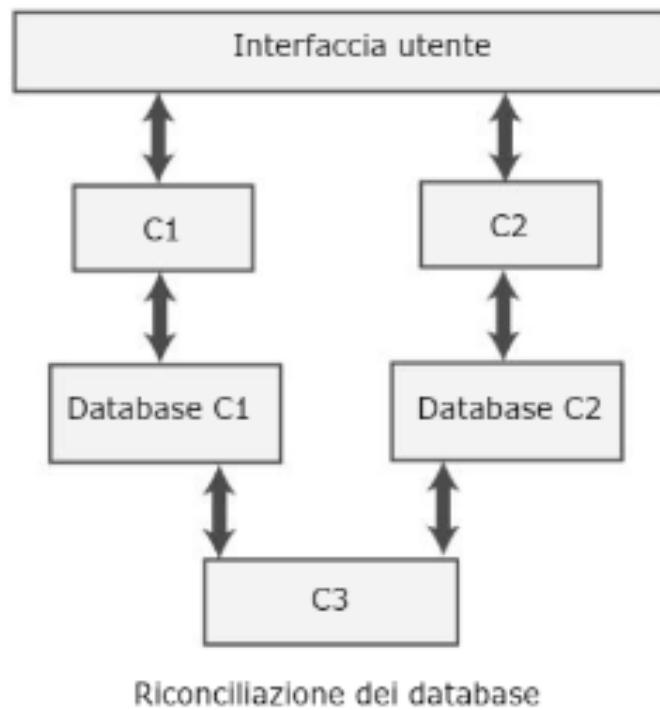


Figura 6.4: Esempio di architettura che privilegia le prestazioni

I due esempi (prestazioni e manutenibilità) mostrano come spesso requisiti diversi richiedano scelte architettoniche in **conflitto** tra loro. Il progettista deve trovare dei giusti **compromessi**, come ad esempio adottare scelte architettoniche diverse per parti differenti del sistema. L’architettura di un sistema software può basarsi su uno **stile** o **schema architetturale** noto, ovvero su modelli di architetture provate e verificate in vari sistemi, di cui si conosce pregi e difetti.

Questione	Importanza per l'architettura
Caratteristiche non funzionali del prodotto	Le caratteristiche non funzionali del prodotto, quali sicurezza e prestazioni, riguardano tutti gli utenti. Se non sono ben realizzate, il successo del prodotto è compromesso. Purtroppo alcune caratteristiche sono antitetiche, quindi è possibile ottimizzare solo le più importanti.
Ciclo di vita del prodotto	Se si prevede che il prodotto avrà un ciclo di vita lungo, è necessario creare revisioni del prodotto a intervalli regolari. È quindi necessaria un'architettura che possa evolversi e sia adattabile all'aggiunta di nuove feature e alle nuove tecnologie.
Riuso del software	Riusare componenti di grandi dimensioni di altri prodotti o di software open-source consente di risparmiare tempo e fatica. Vengono però ridotte le possibili scelte architettoniche, perché la progettazione deve basarsi sul software riusato.
Numero di utenti	Se si sviluppa software consumer distribuito attraverso Internet, il numero di utenti può cambiare molto rapidamente. Questo può portare a un forte degrado delle prestazioni, a meno che l'architettura sia progettata per consentire una rapida scalabilità del sistema.
Compatibilità del software	Per alcuni prodotti è importante mantenere la compatibilità con altri software, così che gli utenti possano adottare il prodotto e utilizzare dati predisposti da un altro sistema. Questo aspetto può imporre dei limiti alle scelte architettoniche, come ad esempio il software utilizzabile per il database.

Figura 6.5: Esempio di architettura

**Nota:** l'architettura non è l'obiettivo ma uno strumento del progetto, infatti bisogna evitare di realizzare architetture molto complesse quando non necessario e occorre tenere a mente che la tempestività è spesso una delle qualità più importanti del progetto.

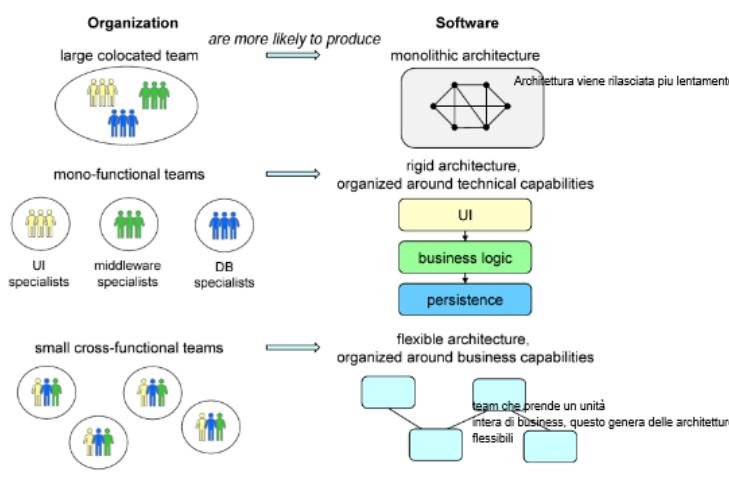


Figura 6.6: Organizzazione architetturale

Per scomporre il sistema si adottano maggiormente i principi dell’astrazione, della modularità, e della separazione degli interessi. Alcuni termini ricorrenti:

- **Servizio:** unità coesa di funzionalità. Può avere significati diversi in base al livello del sistema(esempio: servizio di posta elettronica, servizio per l’invio di messaggi).
- **Componente:** unità software che offre uno o più servizi verso altri componenti (tramite API) o verso gli utenti
- **Modulo:** insieme di componenti correlati, che offrono ad esempio funzioni tra loro collegate.

Nello scomporre il sistema in moduli e i moduli in componenti, si cerca di:

- Localizzare le relazioni tra componenti nello stesso modulo
- Ridurre le dipendenze condivise tra componenti diversi
- Mappare ciascuna responsabilità del sistema (esempi: autenticazione, gestione database, interazione...) in un componente o modulo preciso
- Evitare duplicazione di funzionalità in componenti diversi

Le tecnologie scelte influenzano l’architettura complessiva del sistema e possono limitarla.

Tecnologia	Decisione progettuale
Database	Meglio usare un database relazionale SQL o uno di tipo NoSQL?
Piattaforma	Il prodotto dovrebbe essere rilasciato su app mobile e/o su una piattaforma web?
Server	Meglio usare server dedicati in sede o progettare il sistema per l’esecuzione su un cloud pubblico? Nel caso del cloud, meglio usare Amazon, Google, Microsoft o altre opzioni?
Open source	Esistono componenti open-source che possono essere utilizzati?
Strumenti di sviluppo	Gli strumenti di sviluppo utilizzati limitano la libertà delle scelte architettoniche?

Figura 6.7: Decisione progettuale

## 6.2 Architettura a strati

Lo schema di **architettura a strati** (o **multi-tier**) organizza logicamente il sistema in vari strati (**layer** o **livelli**), ognuno con le sue funzionalità associate. Basato sul principio di separazione degli interessi, vi sono funzionalità del software

logicamente separate e suddivise tra gli strati. Dunque vi è un favorimento dell'astrazione, nello specifico ogni strato rappresenta un livello di astrazione diverso e vi è anche una possibilità di classificare i problemi in alto e basso e livello. Ogni strato sfrutta i servizi dello strato più basso e offre servizi allo strato più alto, come per esempio il modello ISO/OSI per le reti di calcolatori sfrutta un'architettura a strati.

**Modello di Riferimento  
Open Systems Interconnect  
(International Standards Organization)**



Figura 6.8: Modello ISO/OSI

Un client fa una richiesta allo strato più alto, la richiesta attraversa gli strati fino a raggiungere quello che può gestirla. Una richiesta allo strato J può trasformarsi in molteplici richieste (più primitive) allo strato J-1. La richiesta risale poi fino allo strato più alto. Uno strato di basso livello notifica lo stato superiore di un evento: la notifica attraversa gli strati fino a raggiungere quello più interessato. Una notifica allo J può condensare molteplici notifiche (più primitive) pervenute allo strato J-1.

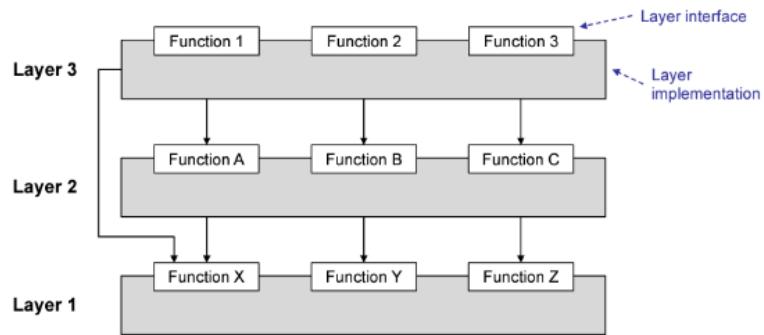


Figura 6.9: Architettura a strati

La gestione degli errori può essere complicata. Con questa architettura a strati vi è la possibilità di:

- gestire errori nello strato in cui sono generati.
- propagare errori verso gli strati superiori

Ogni strato può implementare altri schemi architetturali al suo interno, uno strato può essere un modulo o un processo. Nel primo caso gli strati comunicano tramite chiamate di metodi/procedure, mentre nel secondo caso comunicano mediante protocolli/middleware.



Figura 6.10: Esempio iLearn

### Vantaggi:

- Ogni strato ha un insieme di funzioni chiaramente identificabili, dove gli strati sono rilasciabili incrementalmente (**sviluppo incrementale**)
- Ogni strato può essere facilmente sostituito, modificato, o esteso senza impatto sul resto del sistema, si favorisce **mautenibilità e portabilità**, è sufficiente rispettare (o ampliare) la sua interfaccia.
- Alcune funzioni critiche possono essere ridondante sui vari strati, ad esempio, l'autenticazione, per aumentare la sicurezza del sistema.
- Lo sviluppo del sistema si può distribuire tra più team.
- Può essere sfruttato per aggiungere funzionalità ad un sistema esistente.
- È possibile progettare strati **generici** e **riusabili** per alcune funzioni (ad esempio, per l'autenticazione o per la crittografia dei dati).

### Svantaggi:

- Le interfacce tra strati devono essere chiare e stabilite a priori (a volte sono dettate da standard esterni)
- Non sempre è semplice dividere le funzioni in strati chiaramente distinti e uno strato potrebbe avere bisogno di comunicare con vari strati inferiori
- L'elaborazione di una richiesta potrebbe dover attraversare tutti gli strati e creare problemi a livello di performance.

## 6.3 Architettura repository

In uno schema di **architettura repository** tutti i dati del sistema vengono gestiti mediante una base dati (repository) centrale, accessibile da tutti i componenti del sistema. I componenti non interagiscono direttamente tra loro, ma soltanto attraverso il repository. Risulta utile quando si devono condividere grandi quantità di dati senza trasmetterli da un componente all'altro (come accade invece nell'architettura a strati). Risulta utile usare questo schema quando l'interazione tra i vari componenti non è guidata da processi chiari e definiti ma dai dati (esempio IDE di sviluppo basato su plug-in)



Figura 6.11: Architettura repository: esempio (IDE)

### Vantaggi:

- A livello di performance, può essere utile per condividere grandi moli di dati,
- i componenti sono relativamente indipendenti e possono essere modificati ed estesi liberamente
- I dati vengono gestiti in modo coerente( sicurezza, backup, ecc..) e le modifiche fatte da un componente sono note anche agli altri poichè la base dati è unica e condivisa
- Risulta facile aggiungere un nuovo componente

### Svantaggi:

- Il modello di dati del repository deve andar bene a tutti i componenti
- Modifiche al repository potrebbero essere molto difficili e costose
- Il repository è un punto comune di malfunzionamento, se non funziona lui non funziona niente
- Potrebbe essere inefficiente dirigere tutte le comunicazioni verso il repository, il quale potrebbe non essere facilmente distribuibile su più calcolatori

## 6.4 Architettura pipe-and-filter

Lo schema di **architettura pipe-and-filter** fornisce una struttura per sistemi che devono elaborare flussi dati, decomponendo l'elaborazione in filtri che scambiano

dati mediante «tubi», ovvero buffer di dati intermedi. Un insieme di filtri connessi da tubi è anche detto **pipeline di processamento**. E' possibile ottenere pipeline diverse ricombinando opportunamente i filtri.

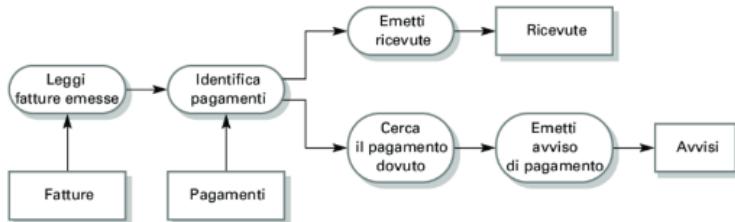


Figura 6.12: Architettura pipe-and-filter: esempio(sistema di fatturazione)

Ogni filtro processa i dati:

- arricchendoli
- filtrandoli
- trasformandoli

L'elaborazione di un **filtro passivo** può essere attività da un filtro precedente/-successivo che invia (push)/richiede (pull) dati. Un **filtro attivo** invece richiede e invia dati attivamente all'interno della pipeline. I tubi sono dei collegamenti tra filtri, possono essere realizzati mediante:

- buffer di tipo FIFO (ad esempio per collegare filtri attivi,
- collegamenti diretti tra filtri (chiamate dirette).

Le **sorgenti dati** di una pipeline possono essere multiple. (Esempi: file, sensori, o altri dati acquisiti da uno stream). Sorgenti dati **passive**: attendono che un filtro richieda i dati (es. file) Sorgenti dati **attive**: inviano dati ai filtri attivamente.(es. sensori) I risultati della pipeline fluiscono in dei **collettori di dati**, anch'essi possono essere passivi o attivi, se attendono i dati o li richiedono attivamente a filtri precedenti. **Vantaggi**:

- Le varie trasformazioni possono essere riusate e ricombinate in modi diversi (pipeline diverse)
- Lo stile del flusso spesso rispecchia la struttura dei processi aziendali che implementa, quindi è facile da capire.

- La sua evoluzione è semplice perché permette di aggiungere facilmente nuovi filtri o rimpiazzando un filtro con un altro.
- Può essere implementato come sistema sequenziale o parallelo, con istanze multiple degli stessi filtri al fine di migliorare le performance

**Svantaggi:**

- E' complicato condividere molte informazioni tra i filtri
- Il formato di trasferimento dati deve essere concordato fra le varie trasformazioni
- Questo può richiedere conversioni di dati dal formato standard a quello interno dei componenti, con conseguente overhead di tempi e risorse.
- La gestione degli errori può essere complicata: gli errori non dovrebbero bloccare la pipeline; un errore potrebbe richiedere la ri-esecuzione della pipeline per un sottoinsieme dei dati.

## 6.5 Architettura client-server

Lo schema di **architettura client-server** riguarda l'**architettura a runtime (o di distribuzione)** del sistema, anziché la sua architettura statica (come invece nei precedenti schemi). I singoli componenti possono avere architetture statiche diverse tra loro. Nel modello client-server, il sistema è organizzato a runtime mediante:

- Un insieme di servizi associati a dei server
- Un insieme di client che accedono e usano tali servizi

Nel caso di sistemi distribuiti su più calcolatori, i vari client comunicano coi server mediante protocolli di rete.

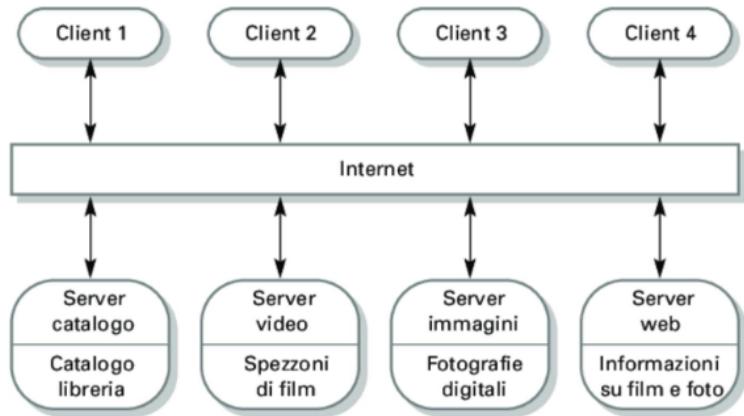


Figura 6.13: Architettura client-server: esempio (catalogo online di film e foto)

### Vantaggi:

- Separazione chiara tra i vari servizi e i client che possono essere modificati in maniera indipendente
- Utile se occorre distribuire l'accesso ai dati da più postazioni
- I server possono essere replicati e distribuiti in rete, al fine di migliorare le prestazioni e aumentare il carico di traffico gestibile.

### Svantaggi:

- I server sono dei punti comuni di malfunzionamento: ad esempio, attacchi DoS o DDOS potrebbero renderli non disponibili
- Le prestazioni e la disponibilità potrebbero dipendere dalla rete e non solo dal sistema
- Le architetture distribuite sono più complesse da gestire

Lo schema di **architettura three-tier** utilizza una architettura di distribuzione di tipo client-server e uno schema statico a tre strati. L'interfaccia utente, i processi logico-funzionali (**business logico**), e l'archiviazione/accesso ai dati sono sviluppate e mantenute come moduli indipendenti, tipicamente processi separati.

- Il **livello di presentazione** (il più alto) mostra le informazioni relative a servizi, per esempio: merce online, carrello

- Il **livello applicativo** processa i comandi del livello di presentazione, implementando regole e logiche specifiche, e attingendo al livello inferiore per recuperare i dati di cui ha bisogno.
- Il **livello dati** è tipicamente un database attraverso il quale le informazioni vengono memorizzate e recuperate

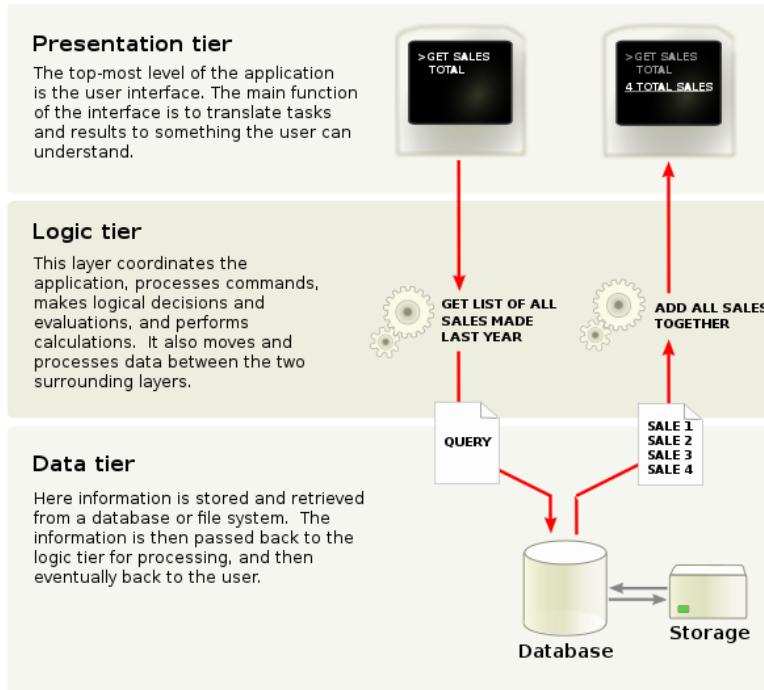


Figura 6.14: Architettura three-tier

Due strategie per associare i livelli ai client:

- **Fat client:** presentazione + applicazione
- **Thin client:** solo presentazione

Il three-tier è spesso utilizzato per lo sviluppo di applicazioni web e mobile (thin clients)

- Livello di presentazione:
  - interfaccia grafica web visualizzata attraverso un browser, e/o
  - app per dispositivi mobili
- Livello applicativo:

- back-end lato server (può essere distribuito su più server) bastato su API web
- Livello dati:
  - database che può risiedere o meno nello stesso server del livello applicativo

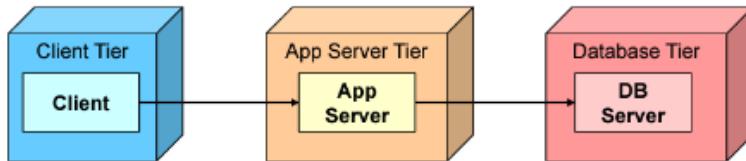


Figura 6.15: Architettura three-tier applicazioni web e mobile

## 6.6 Architettura model-view-controller

Lo schema di **architettura Model-View-Controller (MVC)** struttura un'applicazione interattiva in tre componenti logiche:

- Il **Model** gestisce i dati del sistema e le operazioni associate
- Il **View** definisce e gestisce il modo in cui i dati sono presentati all'utente.
- Il **Controller** gestisce l'interazione degli utenti (ad esempio, pressione di tasti, click del mouse, ecc.) e passa queste interazioni alla View e al Model.

L'MVC rappresenta le interazioni di molti sistemi web ed è supportato da molti linguaggi di programmazione e framework di sviluppo. Risulta particolarmente utile quando ci sono più modi di visualizzare e interagire con i dati, e non sono noti i requisiti futuri di interazione e presentazione dei dati (evolvibilità). Ogni coppia View-Controller rappresenta un'interfaccia utente distinta.

Quando l'utente stimola una View con un input:

1. Il Controller associato viene sollecitato dall'evento, lo elabora e richiede al Model di aggiornarsi
2. Il Model esegue la procedura di aggiornamento e notifica le View e i Controller che il suo stato è cambiato
3. View e Controller si aggiornano di conseguenza

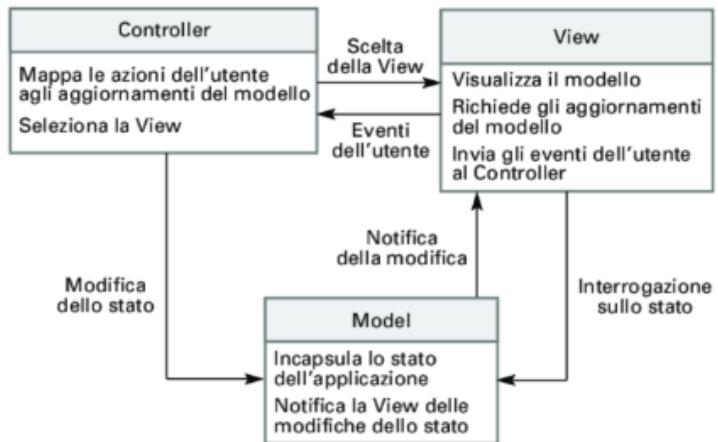


Figura 6.16: Architettura MVC

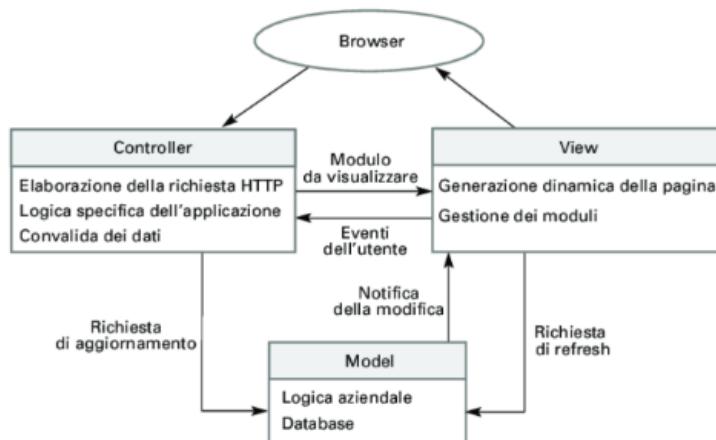


Figura 6.17: Architettura MVC: esempio (architettura WEB)

### Vantaggi:

- Possibili View multiple e sincronizzate sullo stesso modello
- Componenti View e Controller facilmente aggiungibili (plug-and-play)
- Componenti riutilizzabili (tipicamente View e Controller)

### Svantaggi:

- La decomposizione può risultare complessa e laboriosa anche per sistemi relativamente semplici
- Per evitare troppi update, solo le View interessate ad uno specifico cambiamento del Model andrebbero notificate
- Non semplice da comprendere e realizzare correttamente per gli sviluppatori

L'MVC è uno schema che può essere applicato al livello di presentazione di una architettura a strati. In un'architettura three-tier:

- View e Controller nel livello di presentazione
- Model rappresentato dal livello applicazione o da un servizio che fa da ponte tra il livello di presentazione e livello di applicazione

In un'architettura single-tier il model può essere invece direttamente rappresentato dal database o più genericamente dallo stato del sistema.

# 7. Architetture a servizi e microservizi

## 7.1 Architettura a servizi

UN **servizio** è una rappresentazione standard di **risorse** che possono essere utilizzate da altri **programmi** (non da utenti umani). Risorse di vario tipo:

- Informazioni (esempio: catalogo di prodotti)
- Calcolo (esempio: algoritmo)
- Memoria (esempio: storage di immagini)

Si fa riferimento a **servizi web**, ovvero servizi utilizzabili mediante i protocolli standard del web. I servizi hanno **interfacce pubbliche** ben definite e possono essere inclusi in più applicazioni (esempio: portali web e app mobile). Ciascun servizio incapsula una ben precisa funzionalità di business. Un'applicazione può decidere dinamicamente quale servizio utilizzare, se disponibile da più fornitori (esempio: previsione meteo).

L'interoperabilità tra sistemi diversi si ottiene mediante tecnologie **middleware** che mirano a risolvere eventuali eterogeneità tecnologiche. I servizi web possono essere scoperti e invocati, mediante la loro interfaccia e tramite tecnologie web standard. Coloro che richiedono un servizio (**client**) accedono all'interfaccia del servizio e ne localizzano il fornitore tramite un **registro**. La comunicazione tra client e servizio (**legame**) avviene mediante protocolli specifici.

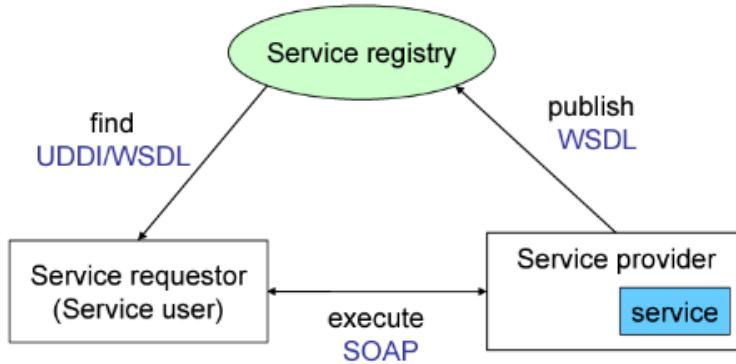


Figura 7.1: Servizi: schema di utilizzo (tecnologia SOAP)

Un'**architettura orientata ai servizi (SOA)** si compone di servizi **indipendenti e debolmente accoppiati**. Alcuni componenti forniscono servizi, altri li utilizzano. I servizi possono essere distribuiti su vari computer dislocati anche su scala mondiale (sistemi distribuiti) e offerto a un pubblico esterno. I servizi possono essere riusati in più applicazioni (esempio: autenticazione) e integrati in applicazioni più complesse (esempio: ricerca di un prodotto). I servizi possono essere composti in maniera flessibile sulla base delle loro interfacce, la composizione può riguardare servizi di più sistemi o organizzazioni anche eterogenei tra loro. La composizione può avvenire sia staticamente (prima del runtime) che dinamicamente (a runtime). Il risultato della composizione di un insieme di servizi è un nuovo servizio, anch'esso componibile.

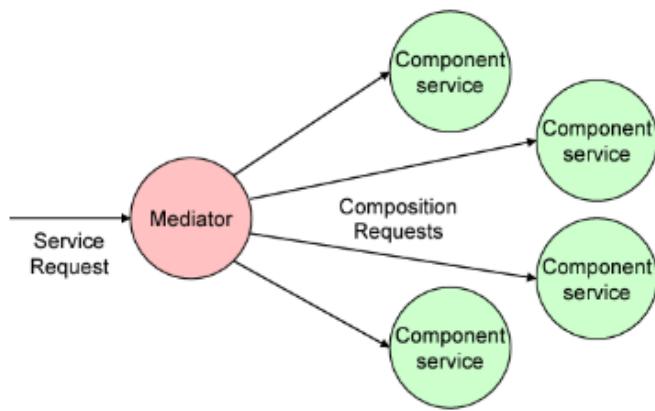


Figura 7.2: Architettura a servizi: composizione con orchestratore

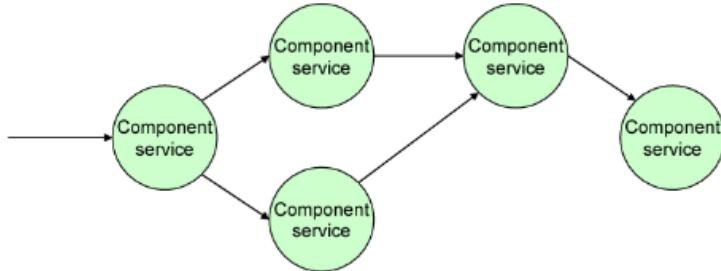


Figura 7.3: Architettura a servizi: composizione con coreografia

Le sei logiche di una architettura a servizi sono:

1. **Identificazione dei servizi candidati:** identificazione servizi da implementare e dei loro requisiti
2. **Progettazione dei servizi:** sviluppo delle interfacce dei servizi (SOAP / REST)
3. **Implementazione e consegna dei servizi**

Il cloud consente l'accesso a risorse sotto forma di servizi, si basa su tre modelli di servizio: **IaaS**, **PaaS**, **SaaS**. Nel modello IaaS (Infrastructure as a Service):

- Il consumatore usa servizi infrastrutturali come server (CPU, RAM e sistemi operativi), storage e connettività

- Il consumatore accede a questi servizi in rete, mediante una loro interfaccia contrattuale e in modo astratto dalla loro implementazione
- Il consumatore può comporre questi servizi, per definire degli ambienti di esecuzione complessi
- Il fornitore dei servizi ha autonomia sull'implementazione e sulla locazione dei servizi.

Le tecnologie a servizi sono basate su un insieme di standard tecnologici per l'interoperabilità indipendenti dalle piattaforme. **SOAP** e **REST** sono le due famiglie principali di standard per i servizi (ad esempio: CORBA, Java RMI, .NET remoting)

### Standard fondamentali SOAP:

- **SOAP** (Simple Object Access Protocol): standard basato su XML per la comunicazione (scambio di messaggi) con i servizi.
- **WSDL** (Web Service Definition Language): standard per definire l'interfaccia di un servizio (operazioni e legami tra operazioni)
- **WS-BPEL** (Web Service Business Process Execution Language): standard per definire dei workflow (processi) che coinvolgono più servizi.
- **UDDI** (Universal Discovery, Description and Integration): standard per la descrizione di registri per la scoperta e l'accesso ai servizi.



Figura 7.4: Standard fondamentali SOAP

## Servizi REST

Limiti della tecnologia SOAP:

- Significativo overhead nell'utilizzo dovuto a tanti protocolli basati su XML
- Molto spesso non si è interessati a funzionalità quali il binding dinamico e all'uso di più fornitori diversi

Lo stile **REST** (Representational State Transfer) è più semplice e leggero, non si utilizza la fase di scoperta del servizio, l'interfaccia e il fornitore devono essere noti. L'elemento centrale è sempre la **risorsa** (informativa, di calcolo, o di memoria), una risorsa può avere più **rappresentazioni** (tipicamente JSON o XML). Ogni risorsa ha un **identificatore** unico, ovvero il proprio **URL**. Le risorse sono manipolabili mediante quattro operazioni: creazione, lettura, aggiornamento e cancellazione. Il protocollo web **HTTP(S)** offre quattro metodi che si mappano bene con le quattro operazioni:

- **POST** -> creazione di una risorsa
- **GET** -> lettura di una risorsa
- **PUT** -> aggiornamento di una risorsa
- **DELETE** -> eliminazione di una risorsa

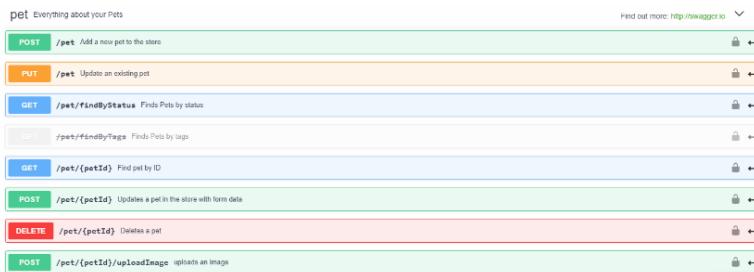


Figura 7.5: Esempio servizi REST

```

curl
-X GET "https://virtserver.swaggerhub.com/fmon/test/1.0.0/pet/0"
-H "accept: application/json"
{
  "id": 0,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": ["string"],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}

```

Figura 7.6: Esempio servizi REST

The screenshot shows a REST API testing interface. The request section has a Method of POST and a Request URL of <https://api.babymat.it/1.0.0/user>. The Headers tab shows two entries: 'content-type' set to 'text/plain; charset=UTF-8' and 'authorization' set to 'Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9eyJ1c2VyX2lkjoYmFieW1hdEBiYWJ5bWF0Lml0In0.YX2qyuYynWf'. The Body tab contains a JSON object representing a user. The response section shows a green bar indicating a '200 OK' status with a response time of 416.25 ms. The Response headers tab lists standard HTTP headers like Server, Date, Content-Type, Content-Length, Connection, and Strict-Transport-Security.

Figura 7.7: Esempio servizi REST

I servizi REST devono essere **stateless** (senza stato), dove in una sessione interattiva, nessun contesto client deve essere memorizzato sul server. I client possono fare **caching** delle risposte, ossia le risposte devono definirsi implicitamente o esplicitamente cacheable o no, in modo da prevenire che i client possano riusare stati vecchi o dati errati. D'altro canto i server possono trasferire **codice eseguibile**, ad esempio script JavaScript.

## 7.2 Architettura a microservizi

L'idea dei microservizi nasce dall'applicazione di alcune idee dell'architettura a servizi, insieme a varie idee e pratiche agili, anche al fine di sfruttare le possibilità offerte dalle tecnologie di virtualizzazione e dal cloud.

I **microservizi** sono dei servizi "piccoli" e autonomi, eseguiti come processi distinti, che lavorano insieme comunicando mediante meccanismi leggeri, tipicamente REST.

L'architettura a microservizi è pensata per una singola applicazione, a differenza della SOA che è pensata per tutte le applicazioni e i sistemi di un'organizzazione. I microservizi sono "piccoli", adatti quindi all'utilizzo di metodi e pratiche agili per il loro sviluppo e alla virtualizzazione e all'automazione dell'infrastruttura (DevOps).

I microservizi sono stateless, quindi facilmente replicabili, eseguibili in parallelo, portabili su ambienti (tipicamente cloud) diversi. Questo favorisce la scalabilità e disponibilità dei servizi offerti (per esempio: applicazioni con milioni di utenti in tutto il mondo, un'interruzione di servizio dell'applicazione è anche un'interruzione del business).

In un'**architettura a microservizi**, l'applicazione viene definita come un insieme di microservizi. Ogni microservizio rappresenta una specifica unità autocontenuta, in esecuzione in un proprio processo (o VM o contenitore). I microservizi comunicano sulla base di meccanismi leggeri, spesso mediante delle API (Application Programming Interface) basate su HTTP(S).

I microservizi sono **autonomi**:

- ciascun microservizio può essere scritto in un linguaggio di programmazione distinto da quello degli altri microservizi
- ciascun microservizio può essere basato su uno stack software diverso, e può anche utilizzare una tecnologia differente per la gestione dei dati
- ciascun microservizio può essere rilasciato indipendentemente dagli altri

I microservizi sono comunque servizi che :

- hanno un interfaccia opportuna, separata dalla loro implementazione, mediante la quale è possibile interagire con essi
- hanno accoppiamento debole e autonomia tra loro
- possono essere composti in modo semplice

- devono essere semplici da scoprire, comprendere ed invocare correttamente
- sono (preferibilmente) stateless

Ciascun microservizio è focalizzato su una singola capacità di business, che deve svolgere bene. La modularità dei microservizi è una caratteristica molto più importante della « dimensione ». Non c'è una dimensione « corretta » per i microservizi. Come regola generale, si può immaginare che un microservizio debba essere sviluppatibile, testabile e rilasciabile in circa 2 settimane (in linea con i tempi di uno sprint). Un'intera applicazione comprende tanti microservizi, per esempio Spotify è basato su oltre 800 microservizi, sviluppati da circa 700 sviluppatori, ogni team di sviluppo (di al più 8 persone) gestisce una decina di microservizi (dati: 2016).

Criterio comune di decomposizione: ciascun microservizio rappresenta una singola capacità di business in modo autocontenuto. Ciascun microservizio può occuparsi di più aspetti del software: UI, logica applicativa e dati e relativi a una specifica capacità business.

I team di sviluppo ideali per i microservizi sono team cross-funzionali (o full stack), in linea con le pratiche DevOps. Una decomposizione basata su capacità di business favorisce la scalabilità (rispetto ad una decomposizione basata su capacità tecniche).

In un'architettura a microservizi, l'interfaccia utente (UI) può essere un componente separato (esempio: app mobile nativa) che sfrutta le funzionalità offerte dai microservizi. In applicazioni complesse, la UI può essere scomposta in varie parti, ciascuna gestita da un microservizio diverso.

Alcuni problemi legati ai microservizi sono:

- La granularità delle API fornite dai microservizi è spesso diversa da quella di cui la UI ha bisogno
- Spesso client diversi necessitano di dati diversi (esempio: desktop vs mobile)
- Il numero di istanze del servizio e le relative posizioni (host + porta) cambia in modo dinamico
- Il partizionamento in servizi può cambiare nel tempo e deve essere nascosto ai client
- I servizi potrebbero utilizzare una serie diversificata di protocolli, alcuni dei quali potrebbero non essere compatibili con il Web

### API gateway:

Per superare tali problemi si può implementare un **API gateway** che funge da punto di ingresso singolo per tutti i client. L'API gateway gestisce le richieste in due modi:

- inoltro diretto ( mapping uno a uno)
- composizione di più microservizi (mapping uno a molti)

L'API gateway può esporre un'API diversa per ciascun client, ad esempio il gateway API Netflix esegue codice adattore specifico per client che fornisce a ciascun client un'API più adatta alle sue esigenze.

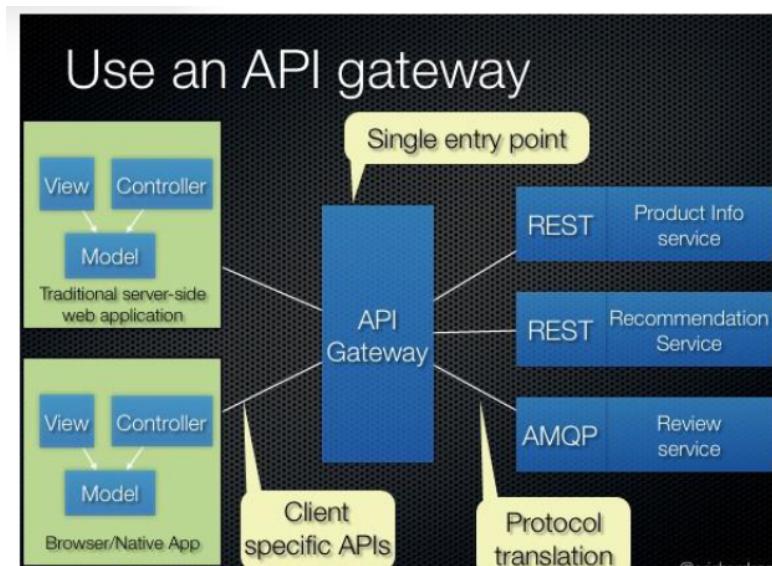


Figura 7.8: Use an API gateway

L'API gateway può inoltre implementare funzionalità di :

- autenticazione
- monitoraggio e log delle richieste
- gestione del versionamento delle API
- ...

In alternativa, è possibile definire un API gateway per ogni tipo di client.

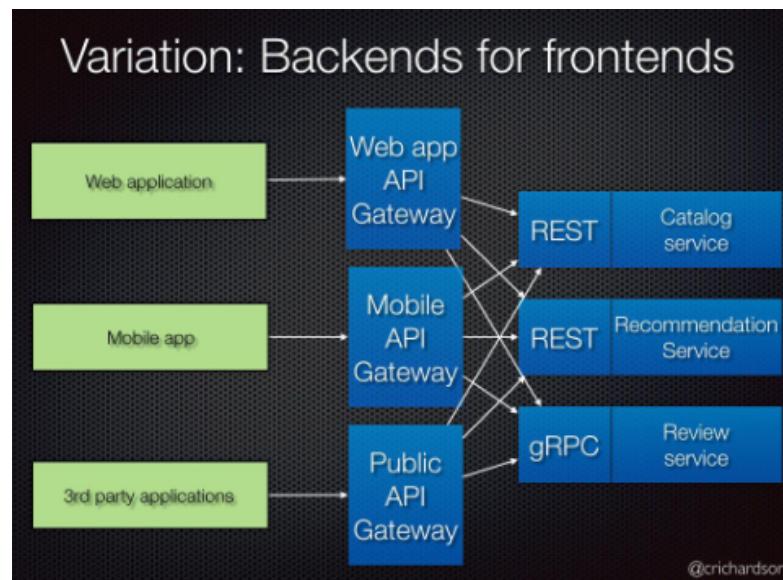
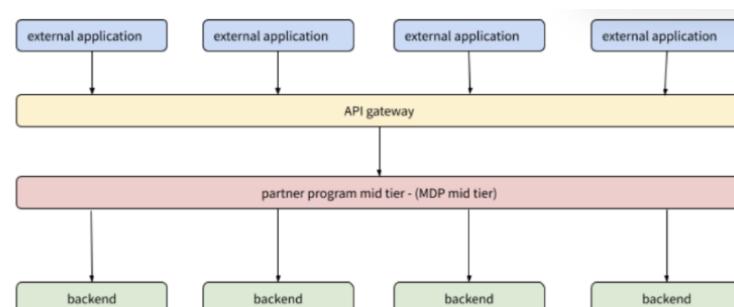


Figura 7.9: Variation: Backends for frontends

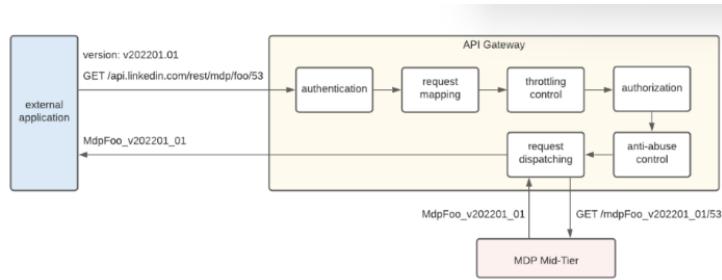
#### Esempio di API gateway:

Obiettivo: disaccoppiare API interne ed esterne (offerte ai partner)



Source: <https://engineering.linkedin.com/blog/2022/-under-the-hood--how-we-built-api-versioning-for-linkedin-market>

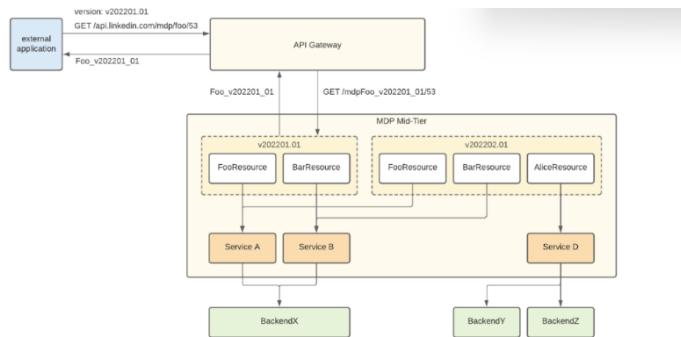
Figura 7.10: Esempio API gateway



Source: <https://engineering.linkedin.com/blog/2022/-under-the-hood--how-we-built-api-versioning-for-linkedin-market>

Figura 7.11: Esempio API gateway

- Autenticazione: Verifica che la richiesta abbia un token di autenticazione valido,
- Mappatura (e validazione) della richiesta: Determina la risorsa interna e il metodo da invocare per soddisfare la richiesta esterna, che include un'intestazione di versione
- Controllo del throttling: Controlla che il richiedente non violi alcuna regola di limitazione della velocità
- Autorizzazione: Verifica che il richiedente sia autorizzato a eseguire la richiesta specificata per chiamare la risorsa interna e il metodo corrispondente
- Controllo antiabuso: Verifica che il richiedente non violi alcuna regola di prevenzione degli abusi
- Invio della richiesta: Invia la richiesta al servizio interno, attende la risposta e la inoltra al richiedente



Source: <https://engineering.linkedin.com/blog/2022/-under-the-hood--how-we-built-api-versioning-for-linkedin-market>

Figura 7.12: Esempio API gateway

### Svantaggi:

L'API gateway è un altro servizio che deve essere sviluppato, distribuito e gestito (maggiore complessità della soluzione). Maggiori tempi di risposta a causa dell'ulteriore roundtrip di rete attraverso l'API gateway (spesso insignificante).

Per ragioni di tempestività ed efficienza, molte organizzazioni sviluppano la prima versione del proprio software mediante un'architettura monolitica. Infatti l'applicazione può essere modulare ma per **monolite** si intende che i diversi moduli sono dispiegati in un unico pacchetto software. Spesso si parte da una architettura di distribuzione di tipo client-server multi-tier. Quando il software diventa più complesso e necessita scalabilità, si può effettuare una decomposizione (di solito iterativa) del monolito in microservizi. La granularità dei microservizi va bilanciata rispetto alle prestazioni richieste, dunque si cerca di evitare una sovrapposizione tra dati usati da microservizi diversi.

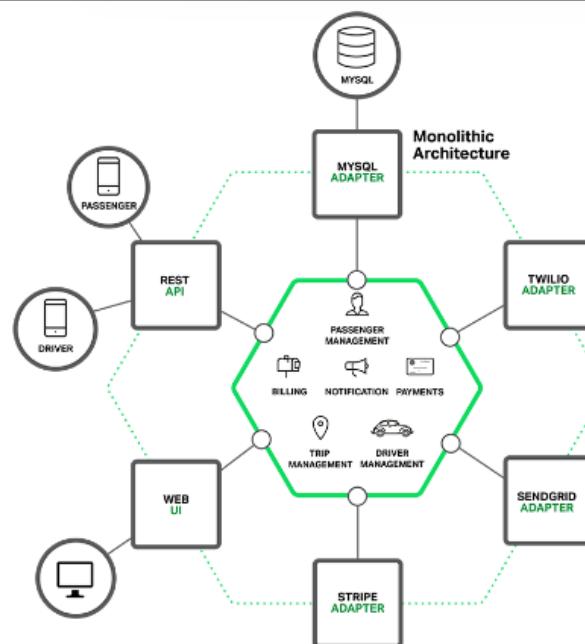


Figura 7.13: Dal monolitico ai microservizi

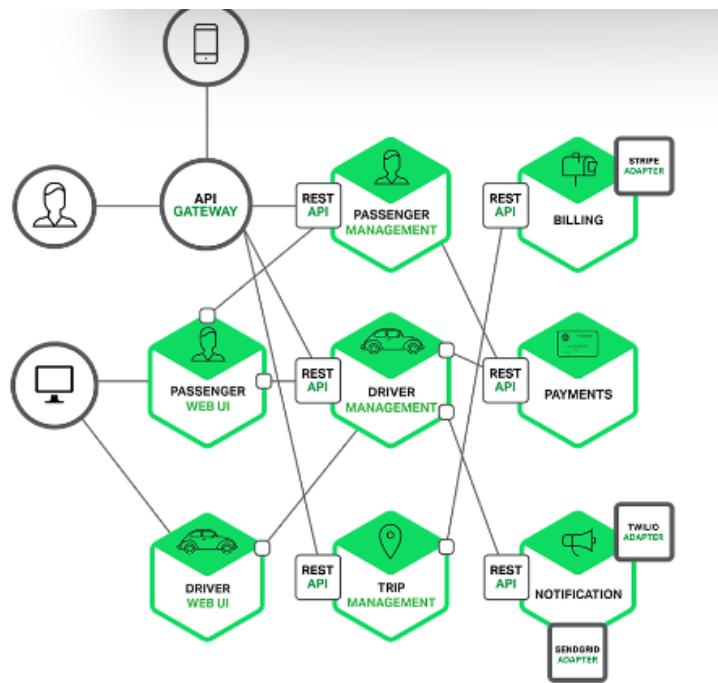


Figura 7.14: Dal monolitico ai microservizi

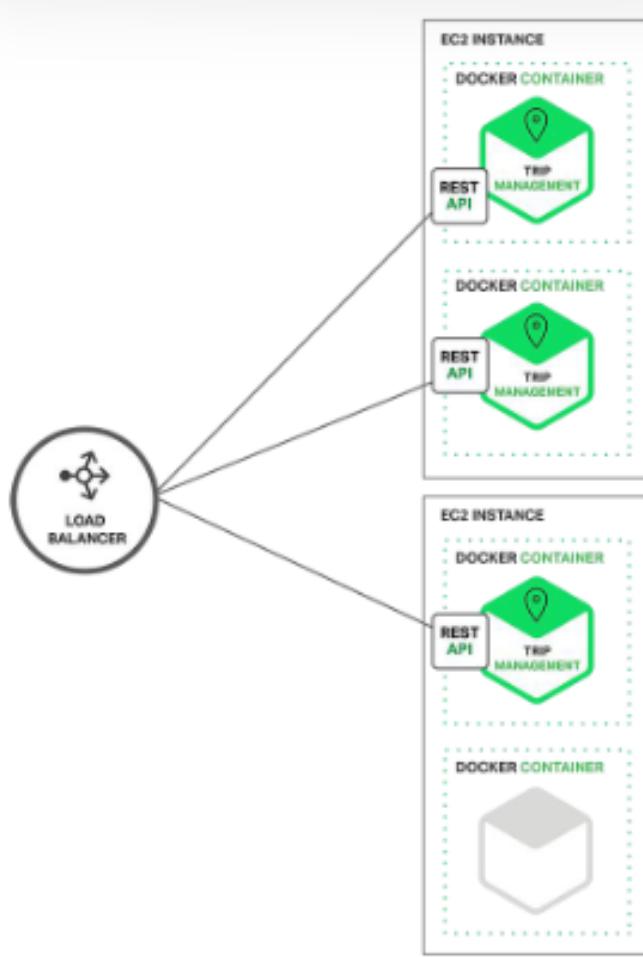


Figura 7.15: Dal monolitico ai microservizi

### Svantaggi:

Un sistema distribuito è più complesso di un sistema monolitico:

- il deployment è più articolato
- vi è la gestione della comunicazione e dei fallimenti
- vi è la gestione della consistenza dei dati in presenza di replicazione e frammentazione
- ci sono test di servizi remoti più difficili da realizzare

Quando un sistema è composto da decine o centinaia di microservizi, la sua distribuzione è più complessa di quella dei sistemi monolitici. Quando si segue il processo DevOps, il team di sviluppo si occupa anche del deployment e del monitoraggio del

microservizio sviluppato. Nell'ambito DevOps, è buona pratica adottare una politica di deployment continuo. Si fa solitamente uso di container per automatizzare il rilascio del servizio. Un container diventa una unità di deployment portabile e facilmente replicabile per questioni di scalabilità. Grandi insiemi di container sono gestiti da sistemi di orchestrazione, quali Kubernetes, in grado di svolgere varie attività (gestione del ciclo di vita, bilanciamento del carico,...).

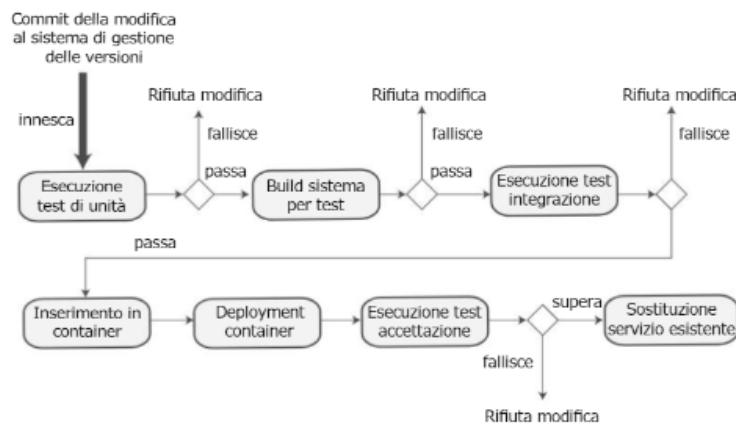


Figura 7.16: Deployment dei microservizi

### Architetture serverless:

Sebbene l'utilizzo dei container per il deployment di microservizi è ormai ampiamente diffuso, la tecnologia serverless è in rapida crescita. Un'architettura **serverless** è un modo per creare ed eseguire applicazioni e servizi senza dover gestire l'infrastruttura. Ossia:

- Provisioning delle risorse, scalabilità, manutenzione, ecc. gestiti in maniera trasparente dal cloud provider
- Non serve mantenere le immagini di container né le relative impostazioni e dipendenze di sistema

Una delle architetture serverless più popolari è **Function as a Service** (Faas), ossia dove gli sviluppatori scrivono il codice dell'applicazione come un'insieme di funzioni discrete, ogni funzione eseguirà un'attività specifica quando attivata da un evento (ad esempio una richiesta HTTP).

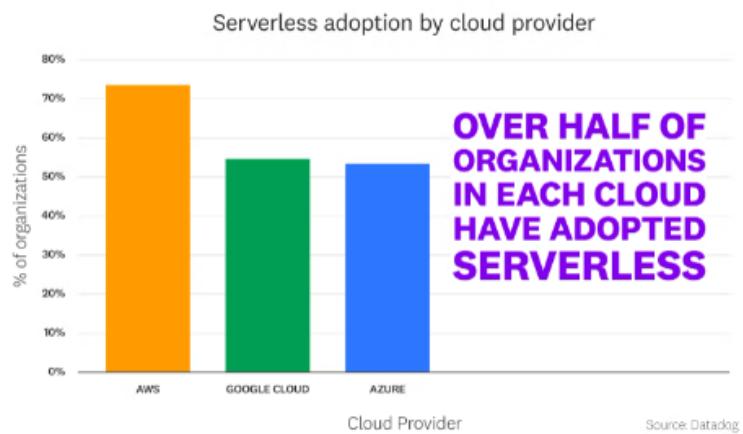


Figura 7.17: Architettura serverless

Quando viene richiamata una funzione, il provider cloud esegue la funzione su un server in esecuzione oppure, se non è attualmente in esecuzione alcun server, attiva un nuovo server per eseguire la funzione. Ci potrebbero essere possibili problemi di latenza legati a cold start, e il deployment di funzioni che richiedono dipendenze (ad esempio applicazioni di ML/DL) può avvenire mediante immagini di container.

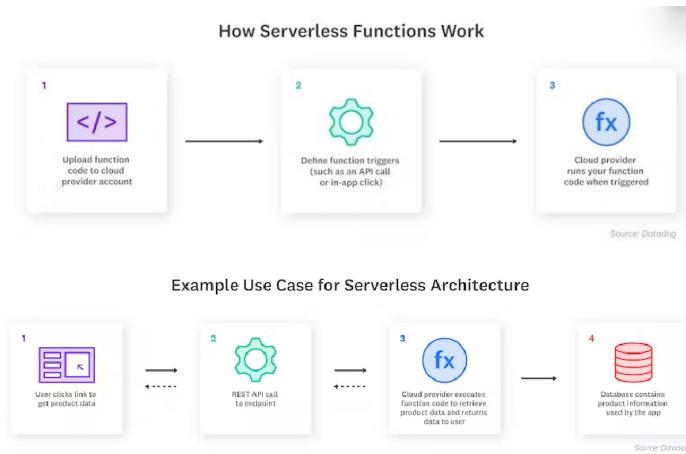


Figura 7.18: Architettura serverless

### Vantaggi:

- Costi per chiamata

- Scalabilità trasparente e gestita dal provider
- Maggiore produttività

### Svantaggi:

- Perdita di controllo rispetto allo stack software utilizzato e alla potenziale condivisione di dati con altre funzioni che utilizzano lo stesso server
- Tempi di latenza potenzialmente più alti a causa del cold start
- Test di integrazione più difficili in ambienti serverless

Spesso le applicazioni distribuite devono poter autenticare le richieste e trasmettere l'autenticazione tra vari servizi. Esistono vari metodi:

- **HTTP basic access authentication:**

- L'autenticazione base del protocollo HTTP è il metodo più semplice per autenticare delle chiamate ad una API
- Nell'header del pacchetto HTTP si specifica: *Authorization: Basic <credentials>*
- Le credenziali non sono criptate e pertanto va sempre usato il protocollo HTTPS per garantire la confidenzialità

- **API keys:**

- Una **API KEYS** è un token fornito da un client durante le chiamate API
- La chiave può essere inviata nella stringa di query: *GET /something?apikey = abcdef12345Oppure nell'header del pacchetto HTTP : GET /something HTTP/1.1 X-API*
- Anche in questo caso la chiave non è criptata e pertanto va sempre usato il protocollo HTTPS per garantire la confidenzialità.

- **Bearer authentication:**

- La **bearer authentication** (o autenticazione con token) è uno schema di autenticazione HTTP che coinvolge token di sicurezza chiamati **bearer token**
- Il bearer token è una stringa generalmente generata dal server in risposta a una richiesta di accesso.

- Il client deve inviare questo token nell'intestazione dell'autorizzazione quando effettua richieste a risorse protette: *Authorization: Bearer <token>*
- **JWT:**
  - **JSON Web Token (JWT)** è uno standard aperto che definisce un modo compatto e autonomo per la trasmissione sicura di token.
  - Queste informazioni possono essere verificate e sono attendibili perché firmate digitalmente: i JWT possono essere firmati usando una chiave privata o una coppia di chiavi pubblica/privata

Encoded PASTE A TOKEN HERE

Decoded EDIT THE PAYLOAD AND SECRET

**HEADER: ALGORITHM & TOKEN TYPE**

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

**PAYOUT: DATA**

```
{
  "sub": "1234567898",
  "name": "John Doe",
  "iat": 1516239022
}
```

**VERIFY SIGNATURE**

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) == secret base64 encoded
```

SHARE JWT

Figura 7.19: <https://jwt.io/>

```
function generateJWT($email){
    $header = json_encode(['typ' => 'JWT', 'alg' => 'HS256']);
    $payload = json_encode(['user_id' => $email]);
    $base64UrlHeader = base64_encode($header);
    $base64UrlPayload = base64_encode($payload);
    $tosign = $base64UrlHeader . "." . $base64UrlPayload;
    $signature = hash_hmac('sha256', $tosign, JWT_PWD, true);
    $base64UrlSignature = base64_encode($signature);
    $jwt = $base64UrlHeader . "." . $base64UrlPayload . "." .
                                     $base64UrlSignature;
    return $jwt;
}
```

Figura 7.20: JWT in PHP

- Quando l'utente accede utilizzando le proprie credenziali, viene restituito un token JWT
  - Il token può poi essere utilizzato mediante autenticazione di tipo bearer per successive richieste: i token hanno tipicamente una scadenza.
  - Il server verifica la presenza di un JWT valido nell'intestazione e, se presente, gestisce la richiesta.
- **auth0.com:** è possibile avvalersi di meccanismi di authentication-as-a-service

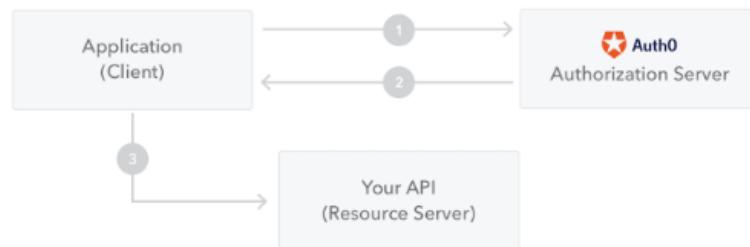


Figura 7.21: <https://auth0.com/>

## 8. Esercitazione 2: Progetto di API REST

**OpenAPI** è un formato di **descrizione** (non di sviluppo) per API REST. Un file OpenAPI consente di descrivere l'intera API tra cui:

- Endpoint disponibili (es: /utenti) e operazioni su ciascun endpoint (es: GET/utenti, POST/utenti)
- Parametri di input/output per ogni operazione
- Metodi di autenticazione

Le specifiche API possono essere scritte in YAML o JSON (useremo YAML). Le applicazioni implementate basandosi su OpenAPI possono automaticamente generare la documentazione di metodi, parametri e modelli (Semplifica la sincronizzazione della documentazione, delle librerie client e del codice sorgente). I clients possono comprendere e consumare servizi senza conoscere l'implementazione del server o accedere al codice del server (Approccio simile a SOAP).

**Swagger** è un insieme di strumenti open source basati sulle specifiche OpenAPI per progettare, costruire, documentare e utilizzare API REST. Principali strumenti di Swagger:

- **Swagger Editor:** editor web per OpenAPI
- **Swagger UI:** esegue il rendering delle specifiche OpenAPI come documentazione API interattiva
- **Swagger Codegen:** genera stub server e librerie client da una specifica OpenAPI

```
1. openapi: 3.0.0
2. info:
3.   title: Sample API
4.   description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/)
5.   version: 0.1.9
6.
7. servers:
8.   - url: http://api.example.com/v1
9.     description: Optional server description, e.g. Main (production) server
10.    - url: http://staging-api.example.com
11.      description: Optional server description, e.g. Internal staging server for development
12.
13. paths:
14.   /users:
15.     get:
16.       summary: Returns a list of users.
17.       description: Optional extended description in CommonMark or HTML.
18.       responses:
19.         '200': # status code
20.           description: A JSON array of user names
21.           content:
22.             application/json:
23.               schema:
24.                 type: array
25.                 items:
26.                   type: string
```

Figura 8.1: Swagger Editor

Ogni definizione di API deve includere la versione della specifica OpenAPI su cui si basa.

```
1. openapi: 3.0.0
```

Figura 8.2: OpenAPI (1)

La sezione **info** contiene informazioni quali titolo, descrizione (opzionale), versione (esempio: major.minor.patch).

```
1. info:
2.   title: Sample API
3.   description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/)
4.   version: 0.1.9
```

Figura 8.3: OpenAPI (2)

La sezione **server** specifica il server e l'URL di base. È possibile definire uno o più server, ad esempio produzione e sandbox.

```
1. servers:
2.   - url: http://api.example.com/v1
3.     description: Optional server description, e.g. Main (production) server
4.   - url: http://staging-api.example.com
5.     description: Optional server description, e.g. Internal staging server for
```

Figura 8.4: OpenAPI (3)

Nell'esempio sopra, `/users` significa quindi:

- `http://api.example.com/v1/users`, oppure
- `http://staging-api.example.com/users`

La sezione **paths** definisce i singoli endpoint (percorsi) nell'API e i metodi HTTP (operazioni) supportati da questi endpoint.

```
1. paths:
2.   /users:
3.     get:
4.       summary: Returns a list of users.
5.       description: Optional extended description in CommonMark or HTML
6.       responses:
7.         '200':
8.           description: A JSON array of user names
9.           content:
10.             application/json:
11.               schema:
12.                 type: array
13.                 items:
14.                   type: string
```

Figura 8.5: OpenAPI (4)

Le operazioni possono possedere parametri passati tramite:

- percorso URL (`/users/userId`)
- stringa di query (`/users?role=admin`)
- intestazioni (`X-CustomHeader: Value`)
- cookie (`Cookie: debug = 0`)

È possibile definire i tipi di dati dei parametri, il formato, siano essi obbligatori o facoltativi, e altri dettagli.

```

1. paths:
2.   /user/{userId}:
3.     get:
4.       summary: Returns a user by ID.
5.       parameters:
6.         - name: userId
7.           in: path
8.           required: true
9.           description: Parameter description in CommonMark or HTML.
10.          schema:
11.            type: integer
12.            format: int64
13.            minimum: 1
14.        responses:
15.          '200':
16.            description: OK

```

Figura 8.6: OpenAPI (5)

Se un'operazione invia un corpo (body) nella richiesta HTTP (POST), si può utilizzare la parola chiave **requestBody**.

```

1. paths:
2.   /users:
3.     post:
4.       summary: Creates a user.
5.       requestBody:
6.         required: true
7.         content:
8.           application/json:
9.             schema:
10.               type: object
11.               properties:
12.                 username:
13.                   type: string
14.       responses:
15.         '201':
16.           description: Created

```

Figura 8.7: OpenAPI (6)

Per ogni operazione, è possibile definire possibili codici di stato (200 OK o 404 Not Found), e lo schema del corpo della risposta:

- Gli schemi possono essere definiti **in linea** o **referenziati** tramite **\$ref**
- Si possono fornire risposte di esempio per diversi tipi di contenuto.

```

1. paths:
2.   /user/{userId}:
3.     get:
4.       summary: Returns a user by ID.
5.       parameters:
6.         - name: userId
7.           in: path
8.           required: true
9.           description: The ID of the user to return.
10.          schema:
11.            type: integer
12.            format: int64
13.            minimum: 1
14.        responses:
15.          '200':
16.            description: A user object.
17.            content:
18.              application/json:
19.                schema:
20.                  type: object
21.                  properties:
22.                    id:
23.                      type: integer
24.                      format: int64
25.                      example: 4
26.                    name:
27.                      type: string
28.                      example: Jessica Smith
29.          '400':
30.            description: The specified user ID is invalid (not a number).
31.          '404':
32.            description: A user with the specified ID was not found.
33.          default:
34.            description: Unexpected error

```

Figura 8.8: OpenAPI (7)

La sezione **components -> schemes** consente di definire strutture dati comuni utilizzare nell'API. (Tali modelli possono essere referenziati tramite \$ref ogni volta che è richiesto (parametri, corpi di richiesta e corpi di risposta)).

I seguenti modelli e JSON sono equivalenti:

```

1. components:
2.   schemas:
3.     User:
4.       properties:
5.         id:
6.           type: integer
7.         name:
8.           type: string
9.           # Both properties are required
10.          required:
11.            - id
12.            - name

```

```

1. [
2.   "id": 4,
3.   "name": "Arthur Dent"
4. ]

```

Figura 8.9: OpenAPI (8)

```

1. paths:
2.   /users/{userId}:
3.     get:
4.       summary: Returns a user by ID.
5.       parameters:
6.         - in: path
7.           name: userId
8.           required: true
9.           type: integer
10.      responses:
11.        '200':
12.          description: OK
13.          content:
14.            application/json:
15.              schema:
16.                $ref: '#/components/schemas/User'
17.   /users:
18.     post:
19.       summary: Creates a new user.
20.       requestBody:
21.         required: true
22.         content:
23.           application/json:
24.             schema:
25.               $ref: '#/components/schemas/User'
26.       responses:
27.         '201':
28.           description: Created

```

Figura 8.10: OpenAPI (9)

È possibile assegnare un elenco di **tag** a ciascuna operazione API. Swagger UI utilizza i tag per raggruppare le operazioni visualizzate:

```

1. tags:
2.   - name: pets
3.     description: Everything about your Pets
4.     externalDocs:
5.       url: http://docs.my-api.com/pet-operations.htm
6.   - name: store
7.     description: Access to Petstore orders
8.     externalDocs:
9.       url: http://docs.my-api.com/store-orders.htm

1. paths:
2.   /pet/findByStatus:
3.     get:
4.       summary: Finds pets by Status
5.       tags:
6.         - pets
7. ...

```

Figura 8.11: OpenAPI (10)

I metodi di autenticazione utilizzati nell'API sono descritti mediante i **security schemes**, metodi di autenticazione supportati sono:

- HTTP authentication: Basic, Bearer,...
- API Key
- OAuth 2
- OpenID Connect Discovery

SwaggerHub è un servizio cloud («software-as-a-service»)

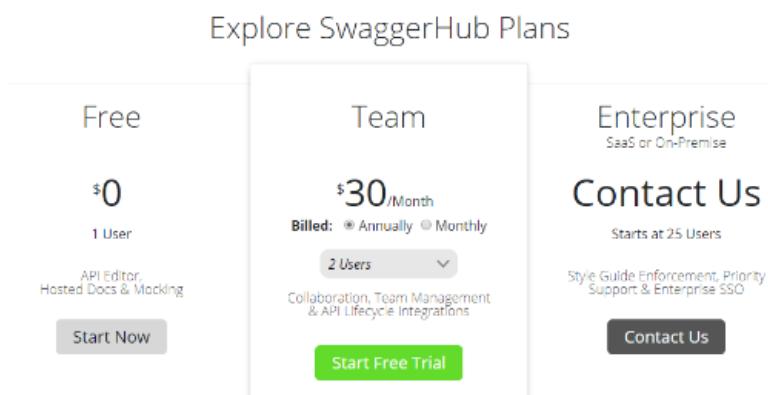


Figura 8.12: OpenAPI (11)

Bisogna prima registrarsi (gratuitamente).

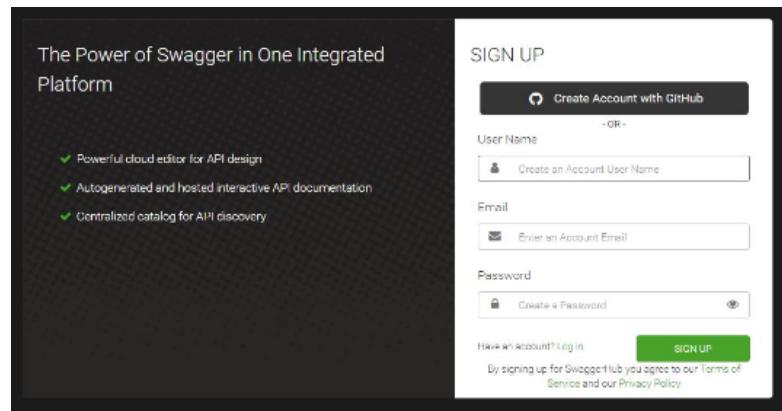


Figura 8.13: OpenAPI (12)

L'obiettivo è andare a scrivere una semplice API per la pubblicazione e la condivisione di annunci.

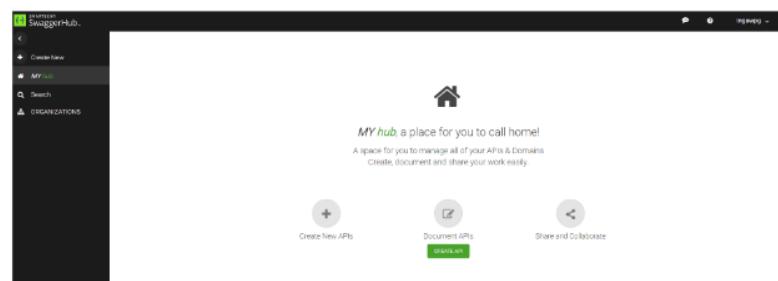


Figura 8.14: OpenAPI (13)

Le specifiche di progetto sono:

- Le API consentono di creare e cercare annunci
- Creazione di un annuncio:
  - Id (assegnato automaticamente)
  - Titolo
  - CATEGORIA
  - Descrizione
  - URL immagine
  - Email di Contatto

- Telefono di contatto
- Ricerca annunci per id, categoria, tutti
- Eliminazione annuncio per id

Infine occorre andare a testare le API. Per testare delle API esistono vari programmi. **Advanced REST client** consente di confezionare e memorizzare semplicemente delle richieste.

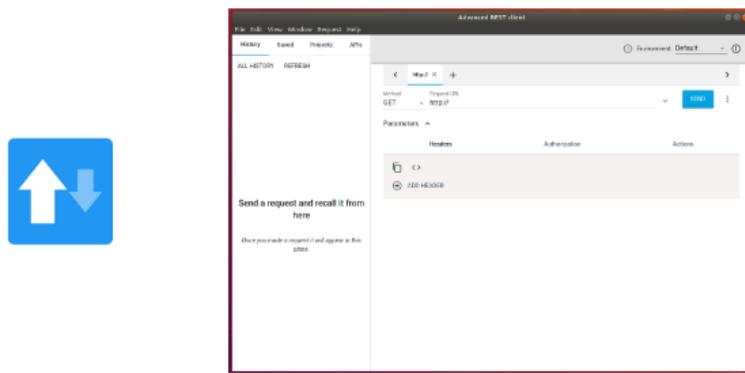


Figura 8.15: OpenAPI (14)

# 9. Programmare ad oggetti

## 9.1 Programmazione a oggetti

Terminata la progettazione architetturale, si passa a progettare e codificare i singoli moduli del sistema. Attività della **programmazione a oggetti**:

- individuare gli oggetti giusti
- fattorizzare gli oggetti in classi con la giusta granularità
- definire interfacce e gerarchie d'ereditarietà
- stabilire relazioni tra classi

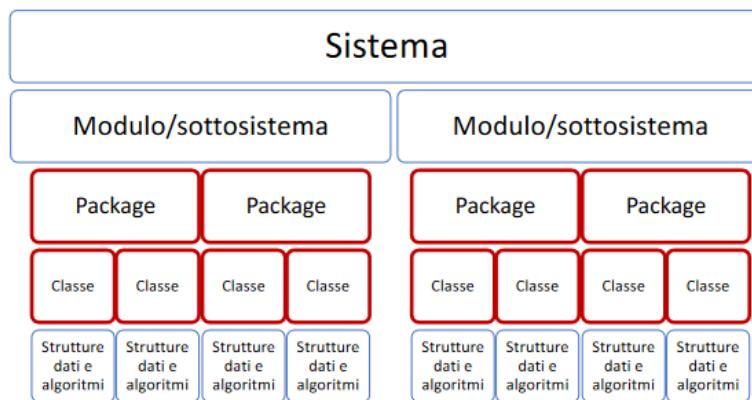


Figura 9.1: Progettazione e codifica a oggetti

Un **oggetto** racchiude sia i **dati(stato)** che le **procedure (metodi o operazioni)** che operano sui dati. Gli oggetti eseguono le proprie operazioni a seguito di

**richieste** (messaggi) da parte di altri oggetti. L'esecuzione di un'operazione può cambiare lo stato interno di un oggetto. Lo stato di un oggetto dovrebbe essere **incapsulato**, ovvero inaccessibile dall'esterno se non tramite le operazioni visibili esternamente.

L'implementazione di un oggetto (attributi e procedure) è definita nella sua **classe**. Gli oggetti sono creati **istanziando** una classe. Ogni oggetto è istanza di una classe e mantiene un proprio stato interno mediante **variabili d'istanza**

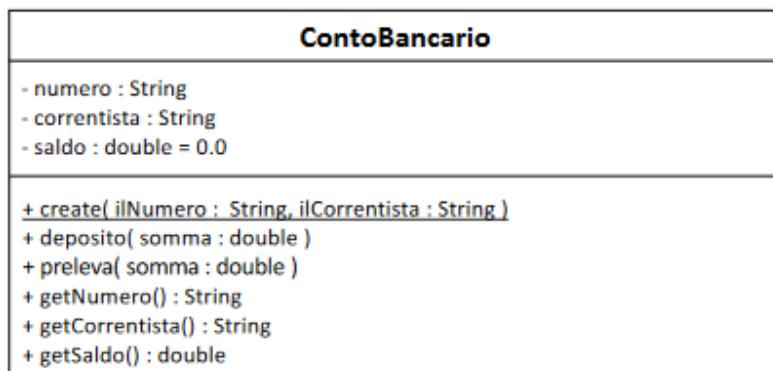


Figura 9.2: Classi

Compito fondamentale del progettista è scomporre il sistema in oggetti (o classi). Si deve tener conto di:

- Incapsulazione
- Granularità
- Dipendenze
- Flessibilità
- Performance
- Evoluzione
- Riusabilità

L'**interfaccia** di un oggetto definisce l'insieme completo di richieste che possono essere inviate all'oggetto. Per ogni operazione definisce:

- Nome
- Parametri in input

- Valore di ritorno

Oggetti diversi possono condividere la stessa interfaccia. Così si modellano implementazioni diverse delle stesse operazioni. L'associazione di una richiesta a un oggetto e una delle sue operazioni può essere nota:

- a tempo di compilazione, si parla di **legame statico**
- a tempo di esecuzione, si parla di **legame dinamico**

Grazie al legame dinamico, possiamo sostituire oggetti diversi purché condividano la stessa interfaccia. Questa caratteristica è nota come **polimorfismo**. Per sfruttare il polimorfismo, bisogna programmare riferendosi all'interfaccia e non all'implementazione. I vantaggi del polimorfismo sono:

- i client (utilizzatori) restano inconsapevoli dei tipi specifici degli oggetti che usano
- i client restano inconsapevoli della classe che effettivamente implementa questi oggetti
- la dipendenza implementativa tra sotto-sistemi diversi si riduce

L'**ereditarietà** ci consente di riusare definizioni di oggetti (ovvero le classi), abbiamo il modello di riuso **white-box**: dove la struttura interna della classe ereditata è (in parte) nota alla sotto-classe.

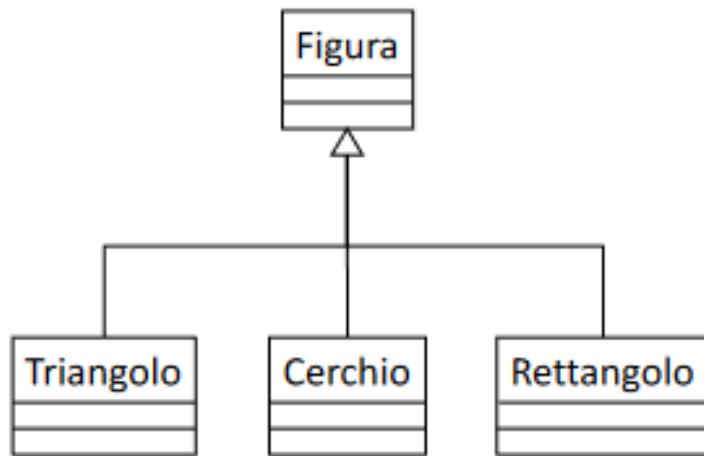


Figura 9.3: Ereditarietà

L'ereditarietà definisce l'implementazione di una **sotto-classe** in funzione dell'implementazione della **super-classe**: la sotto-classe eredita gli attributi e le

operazioni della super-classe (in base alla loro visibilità), la sotto-classe può ridefinire alcune operazioni. Legando l'implementazione di un oggetto a quella di un altro:

- il principio di encapsulamento potrebbe venir meno,
- modifiche implementative alla classe padrone potrebbero richiedere modifiche alla sotto-classe

La **composizione** di oggetti consente invece di creare nuovi oggetti assemblando oggetti più semplici. Il modello di riuso **black-box**: i dettagli implementativi degli oggetti assemblati sono nascosti all'oggetto che li usa.



Figura 9.4: Composizione

La composizione (o più in generale l'associazione tra oggetti) può sfruttare il meccanismo di **delega** per estendere le funzionalità di un oggetto.

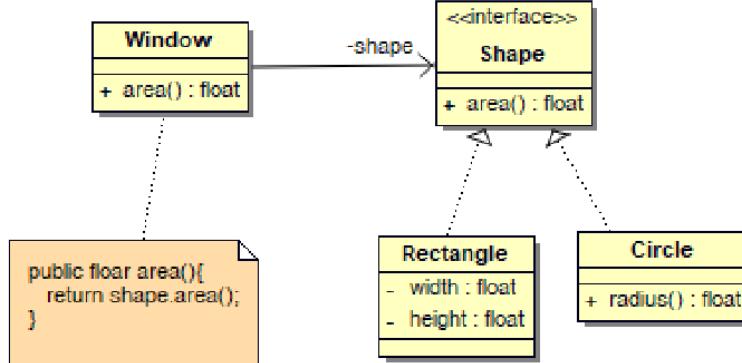


Figura 9.5: Composizione e delega

Mentre l'ereditarietà è definita in maniera statica, la composizione è definita dinamicamente, e si basa sull'utilizzo di interfacce piuttosto che di implementazioni:

- non viola il principio di encapsulamento

- consente di ridurre le dipendenze
- consente di mantenere piccole le gerarchie di classi

Svantaggi:

- può aumentare il numero di oggetti e quindi essere più inefficiente
- può essere difficile da comprendere

Un problema tipico nella programmazione a oggetti è quello di scegliere la classe giusta a cui assegnare la **responsabilità** per un determinato compito. Una buona prassi è di assegnare la responsabilità alla classe che ha (può ottenere) tutte le informazioni essenziali per svolgere tale compito (**classe esperta**). Se la classe ha bisogno di informazioni parziali può delegarne il calcolo ad altre classi che sono esperte rispetto a quelle informazioni. **PRINCIPI SOLID:**

- **Single responsibility:** ogni classe dovrebbe avere una sola responsabilità (alta coesione). Una classe con tante responsabilità è davvero esperta per tutti i compiti assegnati?
- **Open-closed:** una classe dovrebbe essere aperta all'estensione, ma chiusa alle modifiche
- **Liskov substitution:** Ogni superclasse dovrebbe essere rimpiazzabile da una sua sottoclasse, senza modificare il significato del programma
- **Interface segregation:** meglio tante interfacce, una per ciascun client, che un'unica interfaccia buona per tutti
- **Dependency inversion:** meglio dipendere dalle astrazioni che dalle concretizzazioni.

Un **toolkit** è una libreria di classi che possono essere incorporate da un'applicazione. Si compone di classi correlate e riusabili progettate per fornire una funzionalità di interesse generale, per esempio toolkit per la visualizzazione di grafi. Un **framework** è un insieme di classi cooperanti che forniscono lo scheletro di un'applicazione riusabile per uno specifico dominio applicativo, per esempio framework per lo sviluppo di API REST. L'applicazione che usa il framework crea sotto-classi specifiche e/o incorpora classi del framework. Chi progetta un framework effettua scelte (architetturali e di progettazione) che scommette vadano bene per tutte le applicazioni di uno specifico dominio applicativo.

# 10. Esercitazione 3: UML

## 10.1 Introduzione a UML

UML (Unified Modeling Language) è un **linguaggio di modellazione unificato**.

- È un *linguaggio* perché dotato di sintassi e semantica.
- Si dice di **modellazione** perché offre una rappresentazione semplificata di sistemi.
- Si dice *unificato* perché nasce come fusione di notazioni esistenti definite da Grady Booch, Jim Rumbaugh e Ivar Jacobson

UML si avvale di **diagrammi** per specificare, visualizzare e documentare modelli di sistemi software. UML non è un metodo di sviluppo poiché non definisce una metodologia di progetto ma serve per descrivere e visualizzare un progetto. UML è controllato dal Gruppo di Gestione Oggetti (OMG) ed è lo standard industriale per descrivere graficamente il software. UML viene prevalentemente impiegato nella progettazione di **software ad oggetti (OOP)**. Un modello UML prevede due tipi (complementari) di rappresentazione di un sistema.

- **Statica:** descrive la struttura di un sistema, le parti che lo compongono e relative relazioni
- **Dinamica:** descrive il comportamento del sistema, come lo stato del sistema si modifica durante il funzionamento.

Un **modello UML** è l'archivio di tutte le entità e le relazioni create per descrivere un sistema software. I **diagrammi UML** sono viste o finestre che consentono di

vedere il contenuto del modello. Il diagramma non è il modello! **Diagrammi strutturali (o statici):**

- modellano la struttura statica del sistema
- fissano le entità e le relazioni strutturali tra le entità

**Diagrammi comportamentali (o dinamici):**

- modellano la struttura dinamica del sistema
- fissano il modo in cui le entità interagiscono per ottenere il comportamento desiderato del sistema software.

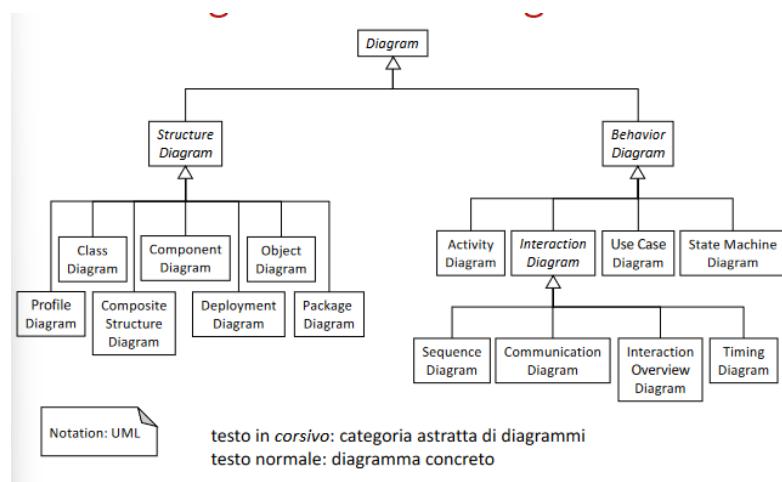


Figura 10.1: UML – gerarchia dei diagrammi

## 10.2 Diagramma delle classi

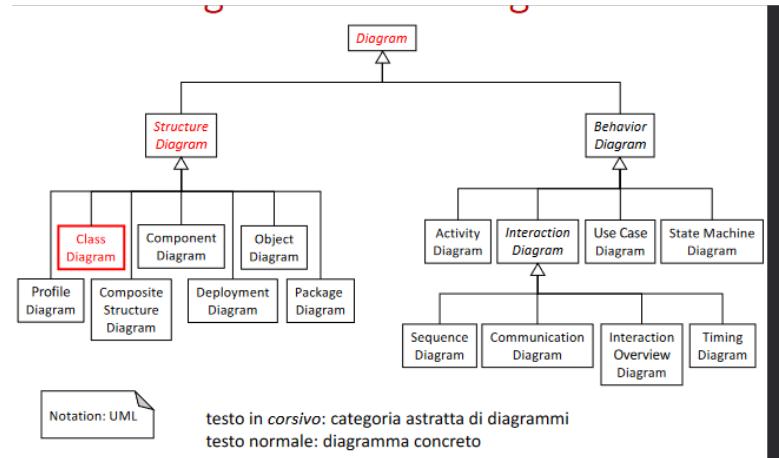


Figura 10.2: Gerarchia dei diagrammi

I **diagrammi di classe** mostrano le diverse classi che costituiscono un sistema e come si relazionano una all'altra. Sono diagrammi statici:

- mostrano le classi, insieme ai loro metodi e attributi oltre alle relazioni statiche tra loro,
- non mostrano le chiamate ai metodi tra di loro

Una **classe** è il descrittore di un insieme di oggetti che condividono gli stessi attributi, operazioni, metodi, relazioni e comportamento.

Una **classe** viene rappresentata da un rettangolo contenente il suo nome in una sottosezione rettangolare.

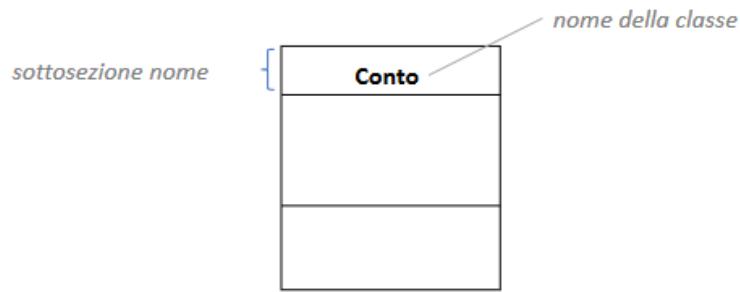


Figura 10.3: Notazione UML per classi

Il nome è l'unica sottosezione obbligatoria. Facoltativamente, si possono mostrare gli **attributi** e le **operazioni** (metodi) della classe in due altre sottosezioni del rettangolo.

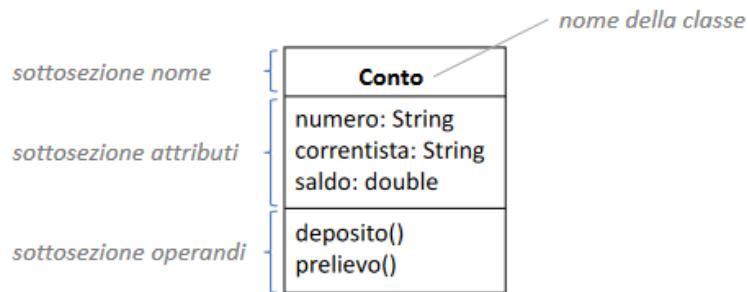


Figura 10.4: Notazione UML per classi

Gli **attributi** sono mostrati con almeno il loro nome. Si possono includere anche il loro **tipo**, il **valore iniziale** e altre proprietà. Ornamento di **visibilità** degli attributi:

- + sta per **pubblici** (public)
- # sta per **protetti** (protected)
- - sta per **privati** (private)
- ~ sta per **pacchetto** (package)

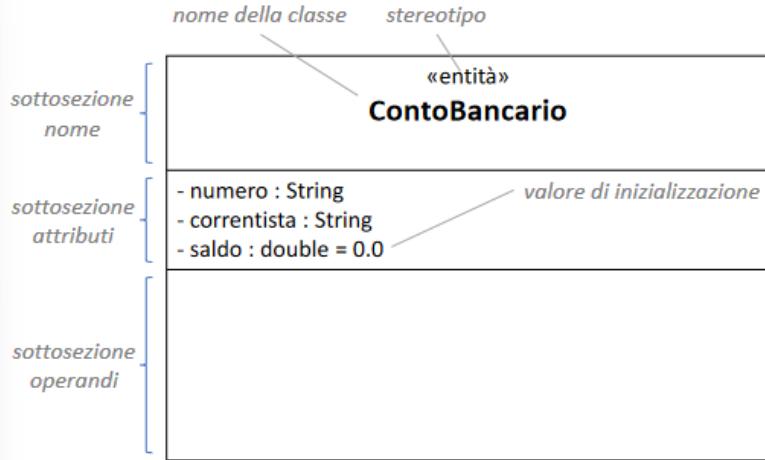


Figura 10.5: Notazione UML per classi

In UML le **operazioni** (metodi) sono mostrate con almeno il loro nome. Si possono includere anche i loro **parametri** e i **tipi restituiti**. Ornamenti di **visibilità** per le operazioni:

- + sta per **pubblici** (public)
- # sta per **protetti** (protected)
- - sta per **privati** (private)
- ~ sta per **pacchetto** (package)

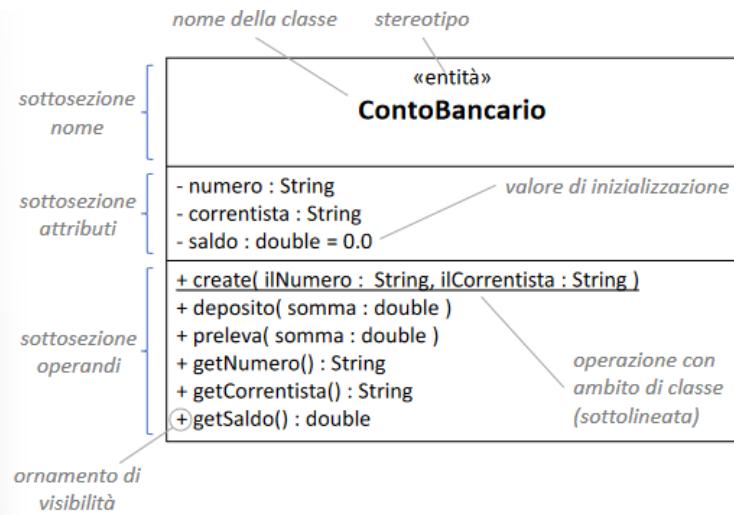


Figura 10.6: Notazione UML per classi

Le diverse classi possono relazionarsi una con l'altra in diversi modi:

- Associazione
- Aggregazione
- Composizione
- Generalizzazione (ereditarietà)
- Dipendenza
- Realizzazione

### **Associazione:**

Un **collegamento tra due oggetti** è una connessione semantica che consente loro di scambiarsi messaggi. Un'**associazione tra due classi** indica che si può avere un collegamento tra una coppia di oggetti appartenenti alle due classi. L'**aggregazione** e la **composizione** possono considerarsi delle forme di associazioni speciali.

Le associazioni possono essere indicate con:

- **Nome dell'associazione:** un verbo che specifica l'azione che l'oggetto origine esegue sull'oggetto destinazione
- **Nome dei ruoli:** un sostantivo che descrive il ruolo che gli oggetti possono ricoprire.
- **Navigabilità:** specifica se la comunicazione tra gli oggetti può essere uni- o bidirezionale
- **Molteplicità:** specifica per ciascun lato dell'associazione, quanti oggetti su questo lato possono relazionarsi a un oggetto sull'altro lato.

Un'**associazione** è rappresentata con una linea che connette le due classi che partecipano alla relazione.



Figura 10.7: Associazioni UML

In **nome dell'associazione**, **i nomi dei ruoli**, e **la molteplicità** possono essere inseriti nelle vicinanze della linea che descrive l'associazione e dal lato delle classi cui si riferiscono.

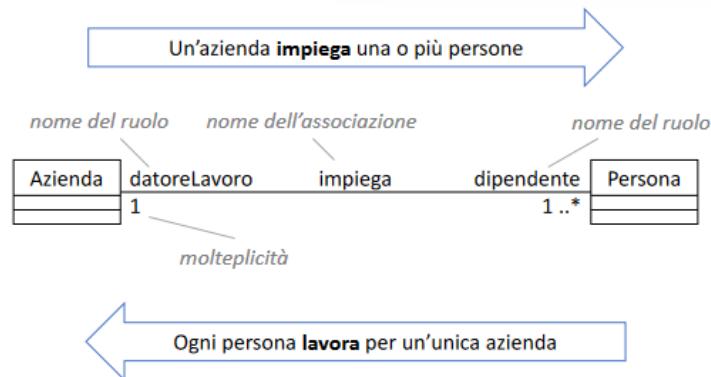


Figura 10.8: Associazioni UML

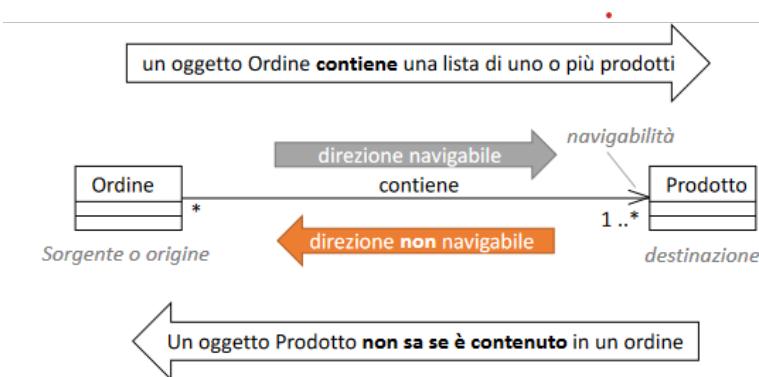


Figura 10.9: Associazioni UML

I valori di molteplicità possono essere:

- **n**: esattamente un intero n non negativo
- **\***: zero o più
- **a .. b**: un intervallo (a,b) di interi non negativi; b può essere \*
- **a .. b, n, m, p .. q**: un insieme di intervalli e/o numeri; il valore più a destra può essere \*

Ornamento	Significato
<b>1</b>	Esattamente uno
*	Zero o più
<b>0 .. 1</b>	Zero o uno
<b>0 .. *</b>	Zero o più (come un singolo *)
<b>1 .. *</b>	Uno o più
<b>1 .. 6</b>	Da uno a sei
<b>1 .. 3, 7 .. 10, 15, 19 .. *</b>	Da 1 a 3, oppure da 7 a 10, oppure esattamente 15, oppure 19 o più

Figura 10.10: Associazioni UML

In una relazione molti-a-molti tra due classi, non sempre è possibile assegnare un attributo ad una delle due classi. Si consideri il seguente esempio:

- Ogni oggetto **Persona** può essere dipendente di molti oggetti **Azienda**
- Ogni oggetto **Azienda** può impiegare molti oggetti **Persona**

Si assuma che ogni **Persona** percepisce uno stipendio da ogni **Azienda** in cui lavora. Dove collociamo l'attributo stipendio: nella classe **Persona** o nella classe **Azienda**?

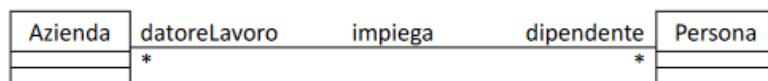


Figura 10.11: Associazioni UML

Ipotesi 1 - attributo stipendio in **Persona**: non riesco a modellare tutte le situazioni in cui una **Persona** lavora per diverse **Aziende** percependo uno stipendio diverso da ciascuna di esse.

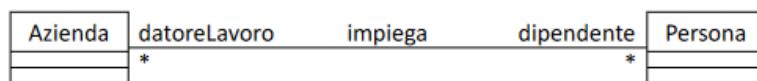


Figura 10.12: Associazioni UML

Ipotesi 2 - attributo stipendio in **Azienda**: non riesco a modellare tutte le situazioni in cui un' **Azienda** impiega molte **Persone** con stipendi potenzialmente diversi.

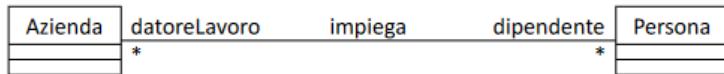


Figura 10.13: Associazioni UML

Le ipotesi 1 e 2 non consentono di ottenere un modello semanticamente corretto perché lo stipendio è una proprietà (attributo) dell'associazione stessa. Per ogni associazione "impiega" tra un oggetto **Persona** e un oggetto **Azienda**, c'è uno specifico stipendio per uno specifico rapporto di lavoro. L'UML consente di modellare questa situazione con una **classe associazione**

Una **classe associazione** è un'associazione che è anche una classe. Oltre a connettere due classi, definisce un insieme di caratteristiche proprie dell'associazione. Una classe associazione è rappresentata:

- dalla linea dell'associazione (compresi tutti i nomi di ruolo e molteplicità)
- dal rettangolo della classe e
- dalla linea tratteggiata verticale che li collega

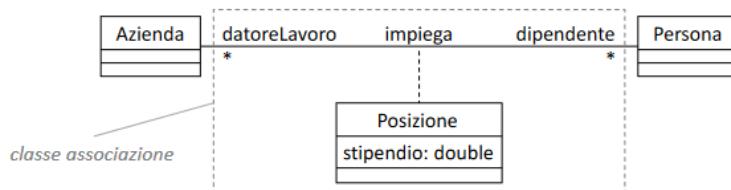


Figura 10.14: Associazioni UML

Si può usare una classe associazione quando vi sia al massimo un unico collegamento tra ogni coppia di oggetti in ogni istante: una **Persona** ha al più una sola **Posizione** con una stessa **Azienda**. In caso negativo, cioè se sono possibili molti collegamenti reciproci in un qualunque istante si "reifica" (rende reale) la relazione trasformandola in una classe: una **Persona** ha (simultaneamente) più di una **Posizione** con una stessa **Azienda**

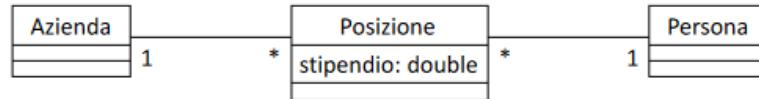


Figura 10.15: Associazioni UML

### Aggregazione:

Le **aggregazioni** sono un tipo speciale di associazione nel quale le due classi partecipanti non hanno un rango uguale, ma hanno una relazione di tipo **tutto-parte**. Un'aggregazione descrive come la classe che ha il ruolo del tutto è composta di (ha) altre classi, che hanno il ruolo di parti. L'aggregato può esistere indipendentemente dalle sue parti e le parti possono esistere indipendentemente dall'aggregato. L'aggregato può essere incompleto se mancano alcune delle sue classi. Aggregati distinti possono condividere una o più parti tra loro. Le aggregazioni sono rappresentate da un'associazione che mostra un rombo sul lato dell'aggregato



Figura 10.16: Aggregazioni UML

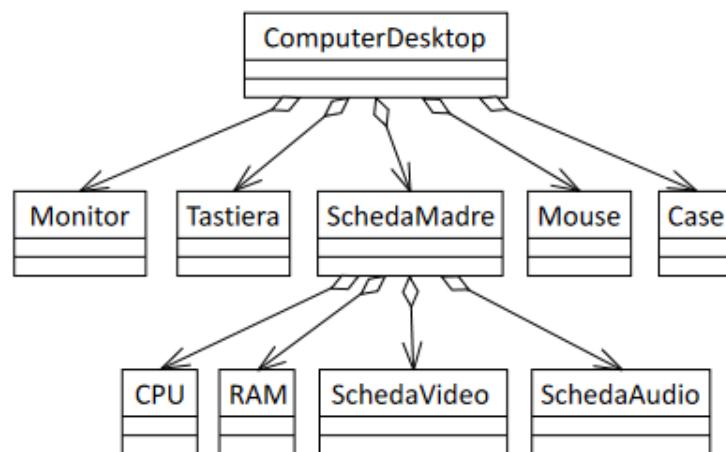


Figura 10.17: Esempio ComputerDesktop UML

### Composizione:

La **composizione** è una forma più forte di aggregazione. Come l'aggregazione, si tratta di una relazione di tipo **tutto-parte**. La differenza è che in una composizione la parte non può esistere al di fuori del tutto:

- le parti esistono solo all'interno del tutto
- se il tutto è distrutto anche le parti muoiono

Ogni parte appartiene ad un unico composito, dunque:

- possono esistere gerarchie di composizione
- non possono esistere reti di composizione

Il composito è l'unico responsabile del ciclo di vita delle parti. Quando si distrugge un composito, questo deve a sua volta,

- distruggere tutte le sue parti, oppure
- cederne la responsabilità a un altro oggetto

Le composizioni sono rappresentate da un'associazione che mostra un rombo solido sul lato del composito.

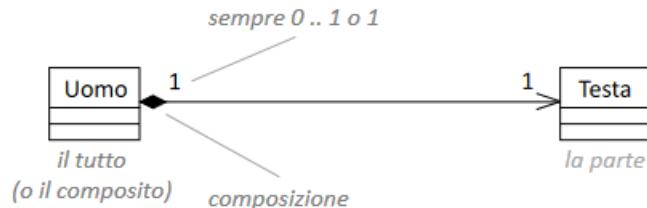


Figura 10.18: Composizioni UML

### Generalizzazione (ereditarietà)

Nella OOP l'**ereditarietà** serve a derivare una nuova classe (**sottoclasse**) da una classe esistente (**superclasse**) in modo tale che la sottoclasse:

- acquisisce (eredita) tutti gli attributi e le operazioni (metodi) della superclasse
- può aggiungere altri attributi e operazioni proprie
- può ridefinire alcune operazioni della superclasse

In UML si usa il termine **generalizzazione** per indicare che una classe è una superclasse di un'altra classe.

La **generalizzazione** è rappresentata da una linea che connette due classi con una freccia (triangolo) sul lato della classe base (superclasse). La superclasse viene posta sopra sopra la sottoclasse (classe derivata).

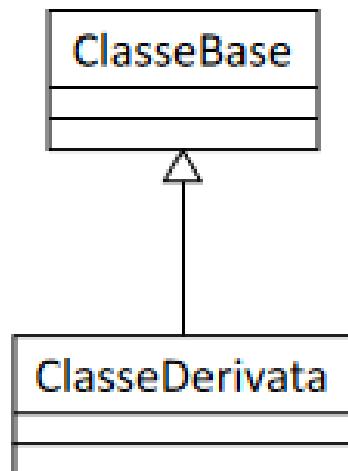


Figura 10.19: Generalizzazione UML

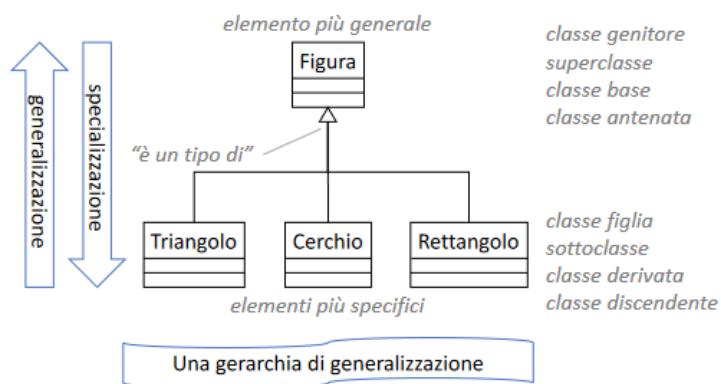


Figura 10.20: Generalizzazione/Specializzazione UML

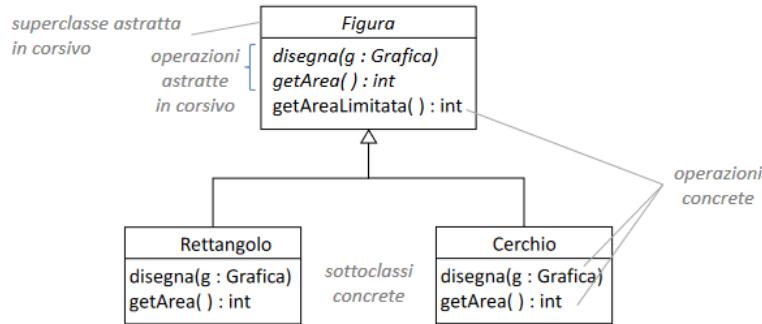


Figura 10.21: Classi astratte e polimorfismo UML

N.B.: *getAreaLimitata()* ritorna il prodotto della larghezza e dell'altezza di una figura, viene calcolata sempre allo stesso modo; non dipende dal tipo di figura concreta considerata.

### Dipendenze:

Le dipendenze più diffuse sono le **dipendenze di uso**: il cliente usa alcuni servizi del fornitore per implementare il proprio comportamento. In particolare, lo stereotipo «usa»: il cliente usa il fornitore come parametro, valore restituito o nella sua implementazione.

Le dipendenze sono indicate con **frecce tratteggiate** dal cliente verso il fornitore:

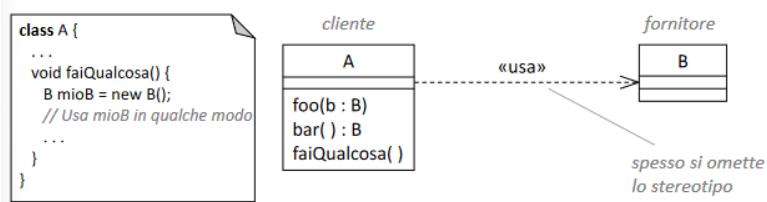


Figura 10.22: Dipendenze UML

### Realizzazione:

Un'**interfaccia** è un insieme di funzionalità pubbliche identificate da un nome. Un'interfaccia non ha attributi, ma soltanto operazioni (metodi). Una **realizzazione** è una relazione tra una classe e un'interfaccia; indica che la classe implementa le operazioni dell'interfaccia.

Un'interfaccia si rappresenta come una classe:

- si omette la sottosezione attributi
- nome e metodi si scrivono in corsivo
- si può includere lo stereotipo «interface»

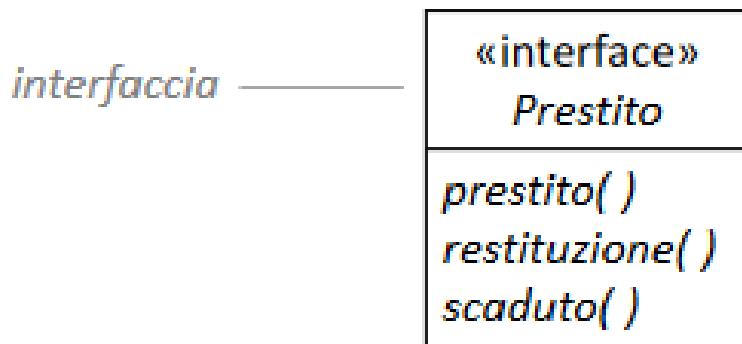


Figura 10.23: Dipendenze UML

La relazione di realizzazione è visualizzata da una linea trattegiata con una freccia sul lato dell’interfaccia:

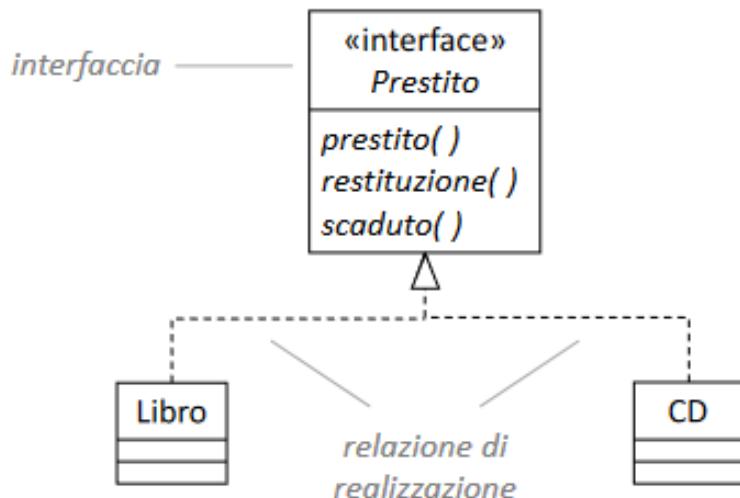


Figura 10.24: Dipendenze UML

### Tipi di relazioni (riassunto)

Sintassi UML		
Tipo di relazione	sorgente destinaz.	Descrizione
Dipendenza	----->	Sorgente dipende da destinazione e può essere influenzato dai suoi cambiamenti
Associazione	—————	Describe un insieme di collegamenti tra oggetti
Aggregazione	◇—————	Destinazione è parte integrante di sorgente
Composizione	◆—————	Aggregazione più forte (più vincolata)
Generalizzazione	————>	Sorgente è una specializzazione (estensione) di destinazione, che è più generale
Realizzazione	-----▷	Sorgente garantisce di eseguire il contratto specificato da destinazione

Figura 10.25: Dipendenze UML

### "Observer" - Diagramma di classe

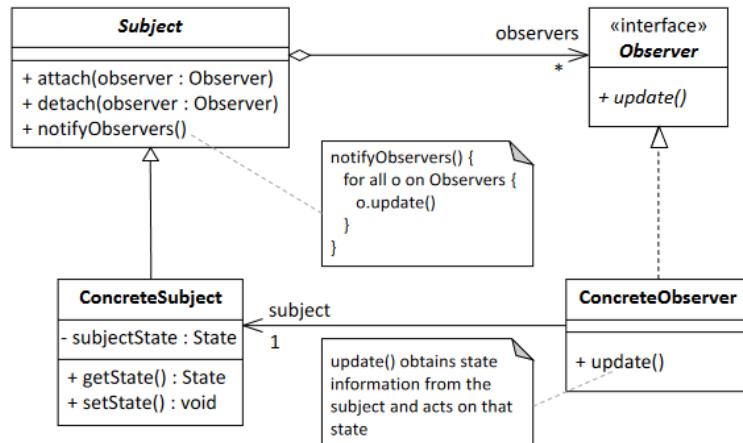


Figura 10.26: Dipendenze UML

## 10.3 Esercizio

# 11. Design patterns

## 11.1 Introduzione ai design patterns

I **design patterns** sono schemi di progettazione a oggetti utilizzati per risolvere problemi comuni. Sono soluzioni note e documentate, descritte ad alto livello, che mirano a favorire il **riuso** efficace di **progetti e architetture**. Semplificano la comunicazione tra esperti e la documentazione tecnica.

**Elementi essenziali:**

1. il **nome** del pattern, che descrive con uno o due parole lo schema proposto. Un **vocabolario** di pattern semplifica la comunicazione tra esperti.
2. Una descrizione del **problema** che il pattern affronta, solitamente in linguaggio naturale
3. Una descrizione della **soluzione** di tipo astratto, spesso supportata da schemi UML e pseudocodice.
4. Le **conseguenze** dell'applicazione del pattern.

I design pattern sono classificati seguendo due criteri:

1. Lo **scopo** del pattern, ovvero ciò che fa:
  - **Pattern creazionali:** riguardano il processo di creazione di oggetti.
  - **Pattern strutturali:** riguardano la composizione di classi e oggetti
  - **Pattern comportamentali:** riguardano il modo in cui classi e oggetti interagiscono tra loro e distribuiscono fra loro le responsabilità.

## 2. Il raggio d'azione del pattern:

- i **class pattern** riguardano le relazioni tra classi e sotto-classi (statiche, ad esempio ereditarietà)
- Gli **object pattern** riguardano le relazioni tra oggetti (dinamiche, ad esempio la composizione)

## 11.2 Pattern creazionali

I design pattern creazionali forniscono un'astrazione del processo di creazione degli oggetti. I principali obiettivi sono:

- Nascondere la modalità di creazione e composizione degli oggetti.
- Incapsulare la conoscenza delle classi concrete utilizzate dal sistema.

Le classi concrete da istanziare possono essere configurate a tempo di compilazione o dinamicamente.

Vogliamo gestire una coda di stampa: dobbiamo garantire un punto di accesso unico per tutti i client dell'applicazione.

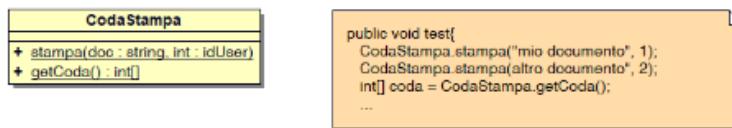


Figura 11.1: Problema: coda di stampa

Possiamo estendere **CodaStampa** gestendo la coda con politiche diverse?  
Soluzione:



Figura 11.2: Soluzione: coda di stampa

### Singleton: scopo

Assicurare che una classe abbia **una sola istanza** e fornire un **unico punto d'accesso globale** a tale istanza.

Esempi di utilizzo:

- Coda di stampa unica
- File di configurazione condiviso
- Per evitare di istanziare oggetti identici di cui si sfruttano solo le operazioni e non lo stato.

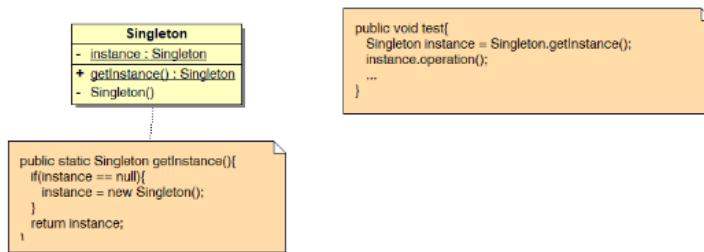


Figura 11.3: Schema Singleton

Il costruttore della classe **Singleton** è privato (o protetto) per evitare che si possano creare più istanze. La variabile **instance** è di classe e privata (o protetta).

L'accesso all'unica istanza **Singleton** è controllato:

- è possibile controllare i tempi e i modi di accesso all'istanza.
- La variabile è inizializzata solo quando e se è usata (**lazy initialization**)

È possibile definire sotto-classi della classe Singleton, configurando l'applicazione per scegliere quale sotto-classe utilizzare.

È possibile gestire un numero maggiore ma controllato di istanze con lo stesso approccio. Ad esempio, possiamo gestire una coda di thread di dimensione pre-definita. Più flessibile rispetto all'uso di operazioni di classe. Alcuni vantaggi del pattern Singleton rispetto all'uso di metodi statici:

- Un oggetto **Singleton** può essere passato come parametro
- La classe **Singleton** può implementare interfacce ed estendere altre classi.
- La classe **Singleton** può essere estesa e si può sfruttare il polimorfismo

### Problema: invio di messaggi

Vogliamo gestire l'invio di messaggi in diverse modalità (email, sms,...)

- Dobbiamo garantire che gli oggetti usati siano consistenti tra loro
- Vogliamo definire nuove modalità di invio in maniera parametrica

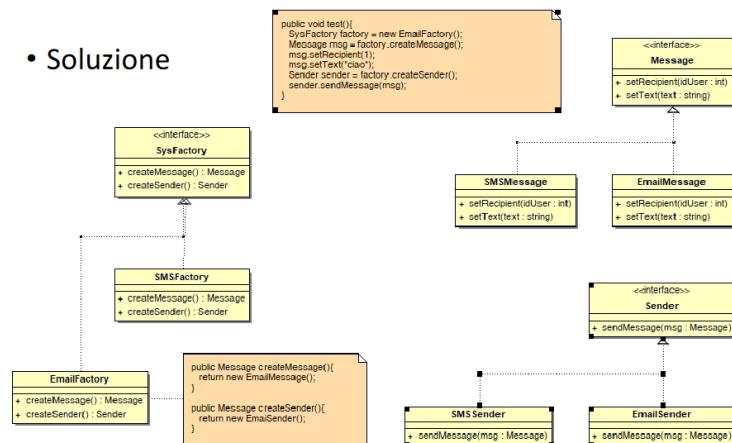


Figura 11.4: Soluzione: problema: invio di messaggi

### AbstractFactory: scopo

Fornire un'interfaccia per la creazione di **famiglie di oggetti correlati o dipendenti** senza specificare quali siano le loro classi concrete.

Esempio di utilizzo:

- Rendere un'applicazione indipendente dallo specifico tipo di look-and-feel utilizzato per i componenti grafici. Lo specifico look-and-feel può essere configurato dinamicamente.

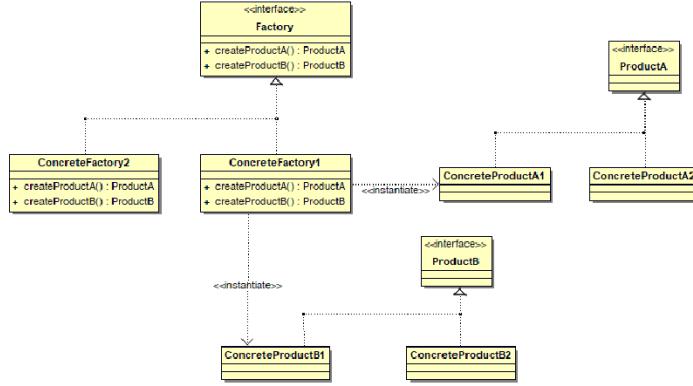


Figura 11.5: AbstractFactory: schema

- **Factory:** dichiara un’interfaccia per le operazioni di creazione di prodotti astratti
- **ConcreteFactory:** implementa le operazioni di creazione definite nella factory astratta
- **Product:** (prodotti astratti) definisce un’interfaccia per una tipologia di prodotti
- **ConcreteProduct:** implementa le operazioni della corrispondente interfaccia astratta

Le factory concrete possono essere del *Singleton*. La factory astratta e i prodotti astratti possono essere sviluppati in moduli diversi rispetto alle loro implementazioni concrete per favorire il disaccoppiamento. Il client deve far riferimento alla factory astratta e ai prodotti astratti, non deve legarsi a nessuna implementazione concreta. Il tipo di factory concreta da usare può essere configurato dinamicamente. I client manipolano i prodotti concreti soltanto tramite le loro interfacce. I nomi delle classi concrete sono dunque isolati nell’implementazione della factory concreta.

- Si evitano dipendenze di compilazione (principio di **inversione del controllo**)
- Favorisce la portabilità
- Factory concrete modificabili dinamicamente (principio dell’**iniezione delle dipendenze**)

Consente di gestire famiglie di prodotti coerenti che devono operare insieme, una famiglia per volta. Se si vuole aggiungere un nuovo prodotto si deve modificare l'interfaccia della factory. In alternativa, si può usare un metodo unico che accetta come parametro il tipo di prodotto (ad esempio, un identificatore di classe). Approccio meno robusto poiché richiede cast esplicativi.

## 11.3 Pattern strutturali

I design pattern strutturali si occupano della struttura del sistema e in particolare delle modalità di composizione di classi e oggetti per formare strutture complesse.

### Problema: file system

Vogliamo implementare un semplice file system che consenta, ad esempio, di spostare una directory e tutto il suo contenuto. Come implementare il metodo di trascinamento di un elemento del file system (directory o file)?

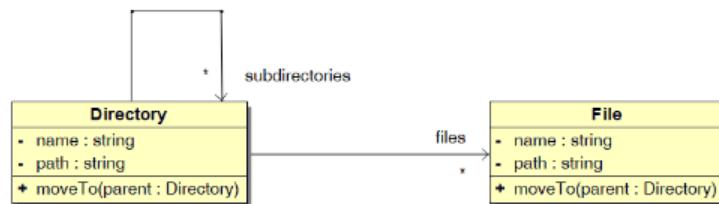


Figura 11.6: Problema file system

### • Soluzione

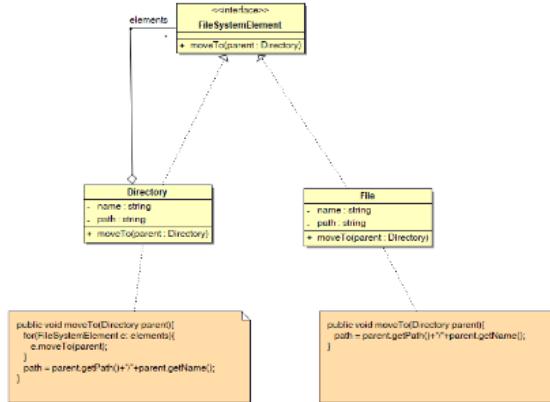


Figura 11.7: Problema file system

## Composite: scopo

Comporre oggetti in strutture ad albero per:

- rappresentare gerarchie parte-tutto e
  - consentire ai client di trattare oggetti singoli e composizioni in modo uniforme

## Esempi di utilizzo:

- Editor grafico in cui ogni oggetto può essere composto da varie primitive grafiche e può andare a comporre altri oggetti
  - Operazioni di trascinamento o ridimensionamento dovrebbero essere applicabili in maniera analoga a oggetti semplici o composti

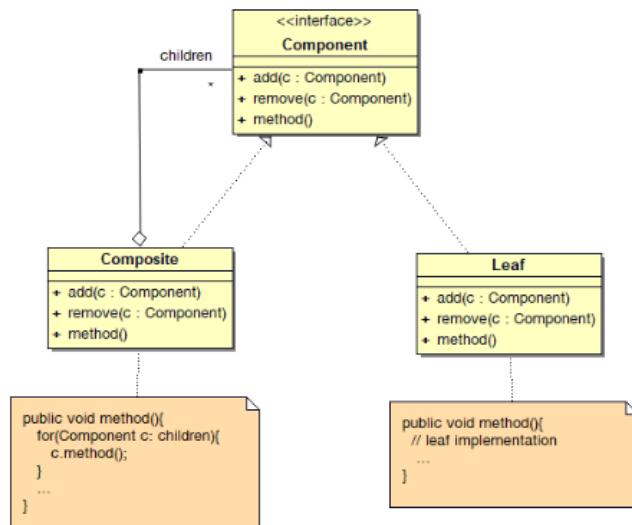


Figura 11.8: Composite: schema

- **Component:** dichiara l'interfaccia per gli oggetti che fanno parte della composizione.
  - **Leaf:** definisce il comportamento degli oggetti foglia della composizione
  - **Composite** Definisce il comportamento degli oggetti che possono avere figli, memorizza i figli.

Definire le operazioni per la gestione dei figli (*add* e *remove*) nell’interfaccia **Component** consente al client di trattare composti e foglie allo stesso modo. Tali operazioni però non hanno senso per le foglie, che devono comunque implementarle (o ereditarle). Queste operazioni si possono spostare su **Composite**. Tale scelta

richiede una conversione esplicita al client che deve poter distinguere tra foglie e composti.

**Component** può prevedere un riferimento al padre (per facilitare l'attraversamento della gerarchia). Non possibile se il riferimento a un componente può essere condiviso da più composti. I composti possono mantenere una lista ordinata dei figli, o utilizzare la struttura dati più consona all'applicazione. Semplifica il codice del client, che non distingue tra composti e foglie. Nuove tipologie di componenti sono facili da aggiungere, poiché i client usano l'interfaccia **Component** per le operazioni comuni. Risulta difficile porre eventuali vincoli sul tipo di oggetti che possono essere composti, poiché richiede l'introduzione di controlli specifici in fase di esecuzione.

#### Problema: controllare connessioni al database

Vogliamo limitare il numero di connessioni al database della nostra applicazione: non possiamo modificare la classe **RealDatabaseConnection** poiché fornita in una libreria esterna.

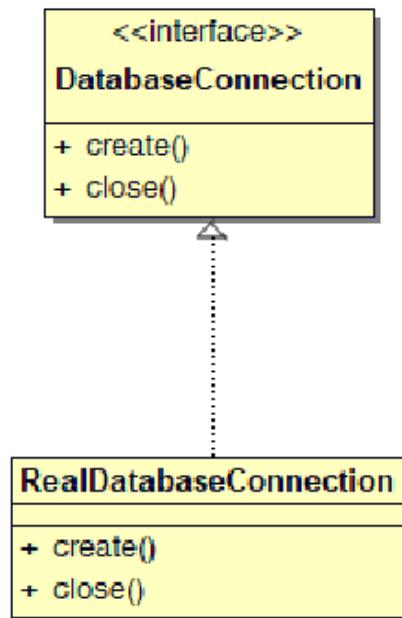


Figura 11.9: Problema: controllare connessioni al database

- Soluzione:

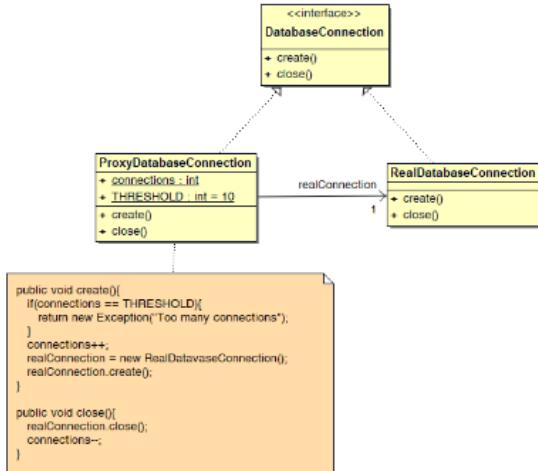


Figura 11.10: Soluzione Problema: controllare connessioni al database

### Proxy: scopo

Fornire un surrogato o un segnaposto di un oggetto per controllare l'accesso a tale oggetto. Esempi di proxy:

- Un **proxy remoto** fornisce una rappresentazione locale per un oggetto in un diverso spazio d'indirizzamento
- Un **proxy virtuale** gestisce la creazione su richiesta di oggetti costosi, quali ad esempio immagini
- Un **proxy di protezione** controlla l'accesso a un oggetto verificando i diritti di chi vuole accedere
- Un **riferimento intelligente** sostituisce un semplice puntatore a un oggetto, consentendo l'esecuzione di attività aggiuntive quando si accede all'oggetto referenziato (ad esempio, può contare il numero di riferimenti all'oggetto reale).

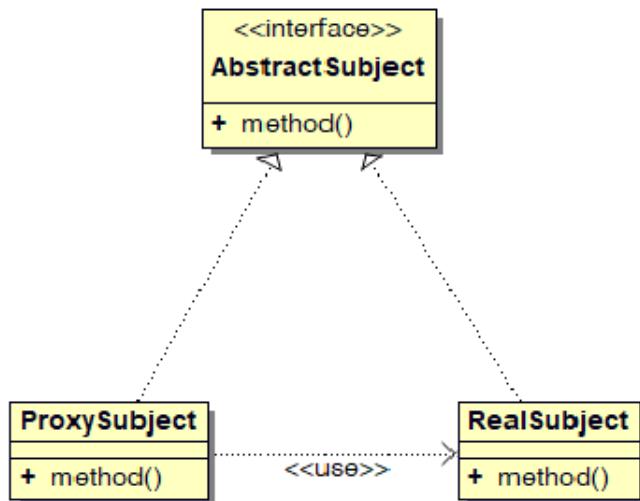


Figura 11.11: Proxy: schema

- **ProxySubject:** mantiene un riferimento al soggetto reale e fornisce una interfaccia identica ad esso
- **AbstractSubject (soggetto):** definisce l’interfaccia comune tra soggetti reali e proxy.
- **RealSubject (soggetto reale):** il soggetto reale che si vuole rappresentare tramite proxy.

Il proxy può controllare l’accesso all’oggetto rappresentato implementando meccanismi di lazy initialization. Un unico proxy può rappresentare soggetti reali diversi, poichè essi hanno tutti la stessa interfaccia.

Il pattern Proxy consente un accesso non diretto all’utilizzo di un oggetto, con finalità che possono essere diverse. Modifiche all’interfaccia **Subject** richiedono modifiche all’implementazione sia dell’oggetto reale che del proxy.

## 11.4 Pattern comportamentali

I design pattern comportamentali si occupano di algoritmi e dell’assegnamento di responsabilità tra oggetti collaboranti. La famiglia di algoritmi force-directed per il disegno di grafi segue un metodo basato su due step: calcolo delle forze tra coppie di vertici, calcolo di un punto di equilibrio del sistema di forze (disegno finale). Vorremmo poter modificare facilmente il modello delle forze che agiscono

tra i vertici. Vorremmo poter utilizzare diversi algoritmi risolutivi per il calcolo di un punto di equilibrio del sistema di forze.

Quali sono i problemi di questo schema?

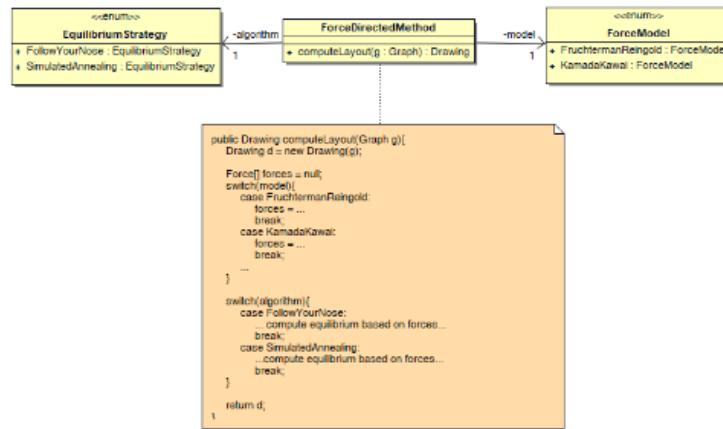


Figura 11.12: Problema: framework algoritmico

- Soluzione:

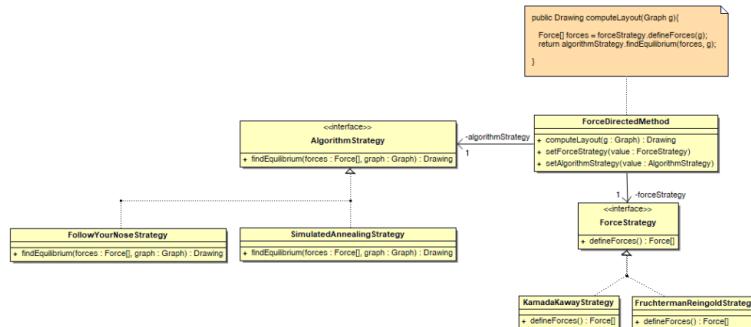


Figura 11.13: Soluzione problema: framework algoritmico

### Strategy: scopo

Definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Gli algoritmi possono variare indipendentemente dai client che ne fanno uso. Gli esempi di utilizzo sono:

- Quando si vuole rendere il client leggero non includendo l'algoritmo nel suo codice

- Quando si vuole poter cambiare dinamicamente l'algoritmo usato dal client

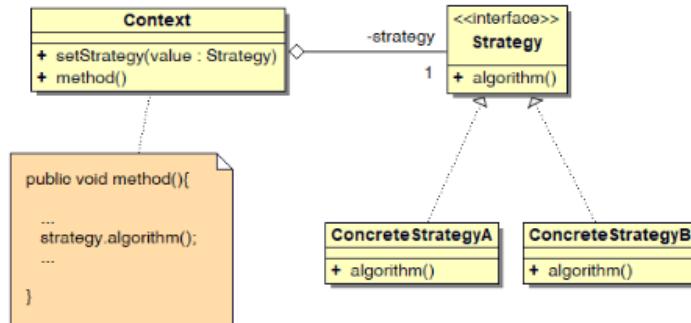


Figura 11.14: Strategy: schema

- **Strategy:** Interfaccia comune per gli algoritmi supportati
- **ConcreteStrategy:** Implementazione di uno specifico algoritmo
- **Context (contesto o client):** va configurato con una strategia concreta e può definire un'interfaccia che consente alle strategie concrete di accedere alle sue strutture dati.

Le strategie concrete potrebbero aver bisogno di vari dati per poter svolgere i loro compiti. Tali dati vanno passati come parametri al metodo ***algorithm*** la cui firma deve quindi essere sufficientemente generica. Oppure l'oggetto **Context** deve contenere tutti i metodi utili alle strategie concrete per eseguire i loro compiti e può essere passato come parametro alle strategie. Potrebbe essere utile definire una strategia di default con cui inizializzare il contesto.

Il pattern Strategy consente di definire famiglie di algoritmi correlati e riusabili. Le relazioni di ereditarietà possono essere usate per fattorizzare comportamenti comuni tra algoritmi. Consente di modificare l'algoritmo dinamicamente, disaccoppiando l'implementazione del client da quella dell'algoritmo. Estendere la classe Context renderebbe il legame statico, rendendo il codice più complesso da manutenere.

Evita l'utilizzo di blocchi condizionali, difficili da testare e manutenere. I client devono conoscere le varie strategie a disposizione oppure devono essere configurati da un iniettore di dipendenze. Può aumentare il numero di oggetti presenti in un'applicazione. Le strategie possono essere dei Singleton.

Si vuole definire una serie di algoritmi che analizzano testi:

- L'analisi del testo è sempre la stessa
- Il formato del testo può variare
- Ulteriori formati potrebbero aggiungersi in futuro

Quali sono i problemi di questo schema?

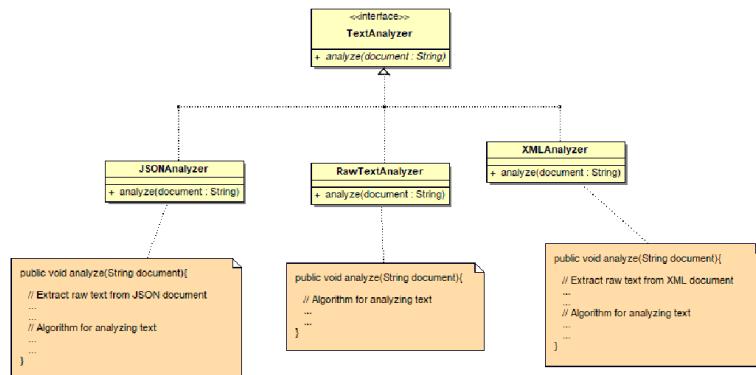


Figura 11.15: Problema: algoritmi con struttura simile

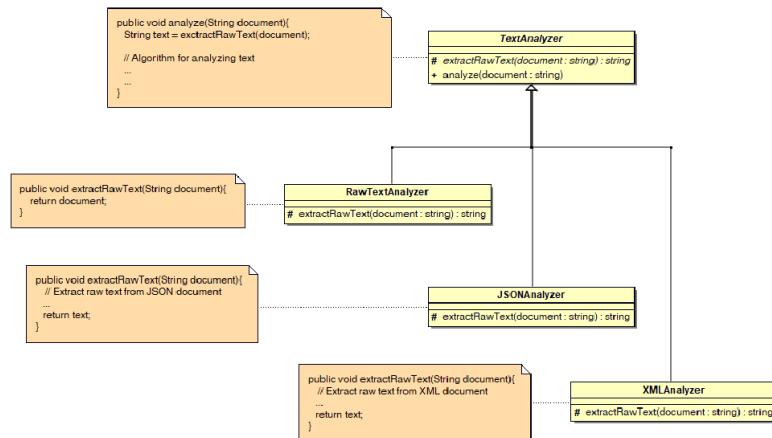


Figura 11.16: Soluzione problema: algoritmi con struttura simile

### Template Method: scopo

Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi dell'algoritmo alle sotto-classi.

- Le sotto-classi non devono implementare di nuovo la struttura dell'algoritmo

- La classe padre chiama il metodo delle sotto-classi «principio di Hollywood: non chiamarci, ti chiamiamo noi».

Esempi di utilizzo:

- Quando un algoritmo ha una parte invariante e alcune parti che invece possono cambiare
- Quando un comportamento comune tra sottoclassi vuole essere portato a fattore comune per evitare codice duplicato.

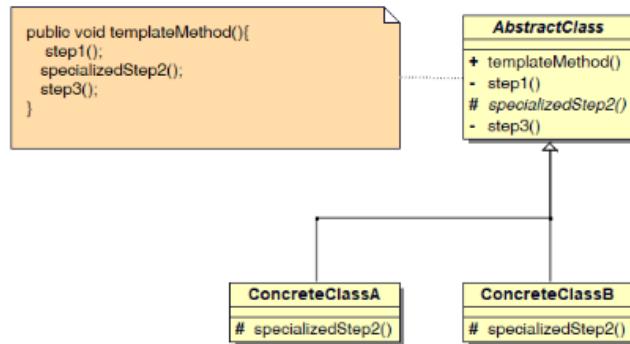


Figura 11.17: Template Method: schema

- **AbstractClass:**

- Definisce le operazioni specializzate ma non le implementa (o offre implementazioni di default)
- Implementa un **metodo template** che definisce la struttura di un algoritmo e chiama le operazioni specializzate

- **ConcreteClass:** implementa le operazioni specializzate

Spesso le operazioni specializzate iniziano con un prefisso standard, ad esempio «do». All'interno di un metodo template, è possibile invocare anche metodi **hook**, ovvero metodi opzionali che possono essere implementati dalle sotto-classi se desiderano «agganciare» un comportamento nel punto in cui il metodo hook è invocato.

Template consente di riusare il codice senza duplicarlo, consente di controllare cosa si può estendere e cosa no in una classe, e consente di integrare dei metodi con codice «custom» attraverso la definizione di operazioni di hook (che di default in genere non fanno nulla). Utile per progettare framework.

Vogliamo creare dei comandi complessi per il personaggio di un videogioco, quindi non conosciamo tutti i possibili comandi a priori.

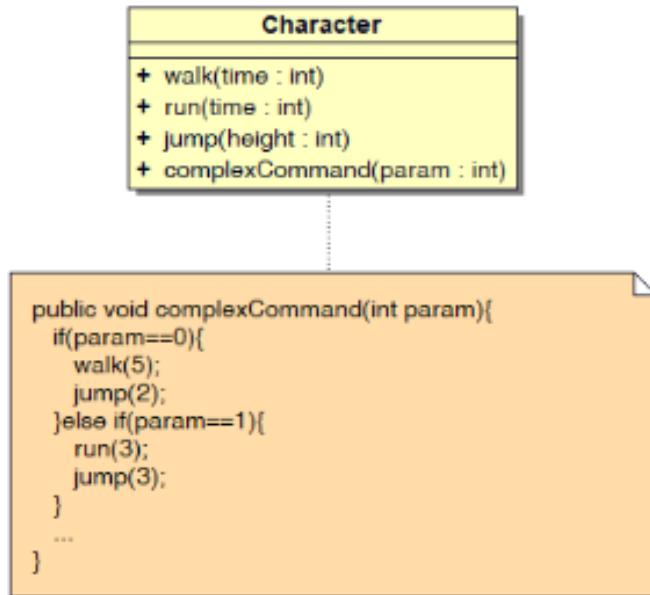


Figura 11.18: Problema: personaggio programmabile

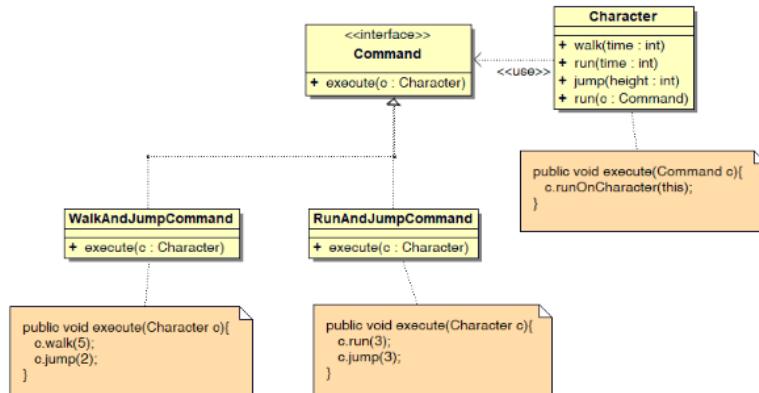


Figura 11.19: Soluzione problema: personaggio programmabile

### Command: scopo

Incapsulare una richiesta di azione (comando) in un oggetto, consentendo di parametrizzare i client con richieste diverse, accordare o mantenere uno storico delle

richieste e gestire richieste cancellabili. Nei linguaggi procedurali (e in molti linguaggi di script), i comandi sono rimpiazzati con funzioni di callback, ossia funzioni invocate solo quando si verifica un determinato evento.

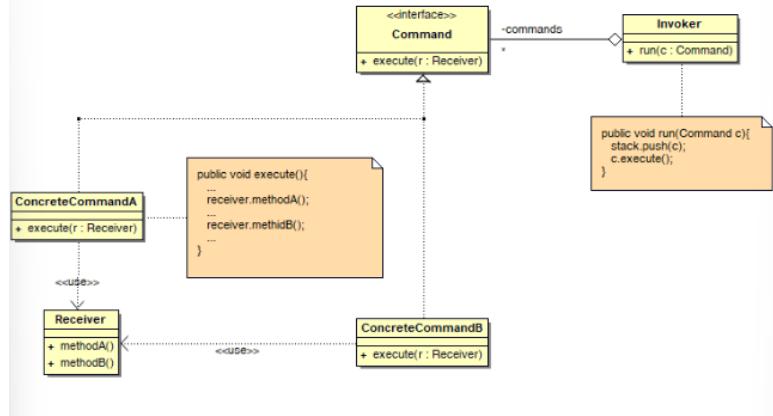


Figura 11.20: Command: schema

- **Command**: Specifica l’interfaccia per l’esecuzione di una generica operazione
- **ConcreteCommand**: Definisce un legame tra oggetto destinario e un’azione
- **Invoker**: Chiede a **Command** di portare a termine l’azione
- **Receiver**: Conosce il modo di svolgere le operazioni associate a una richiesta (può essere qualsiasi classe)

Il pattern può supportare l’annullamento e la riesecuzione dei comandi. Potrebbe dover memorizzare lo stato (o parte di esso) del receiver. Il receiver deve esporre le informazioni e i metodi utili ai comandi (senza però conoscerli).

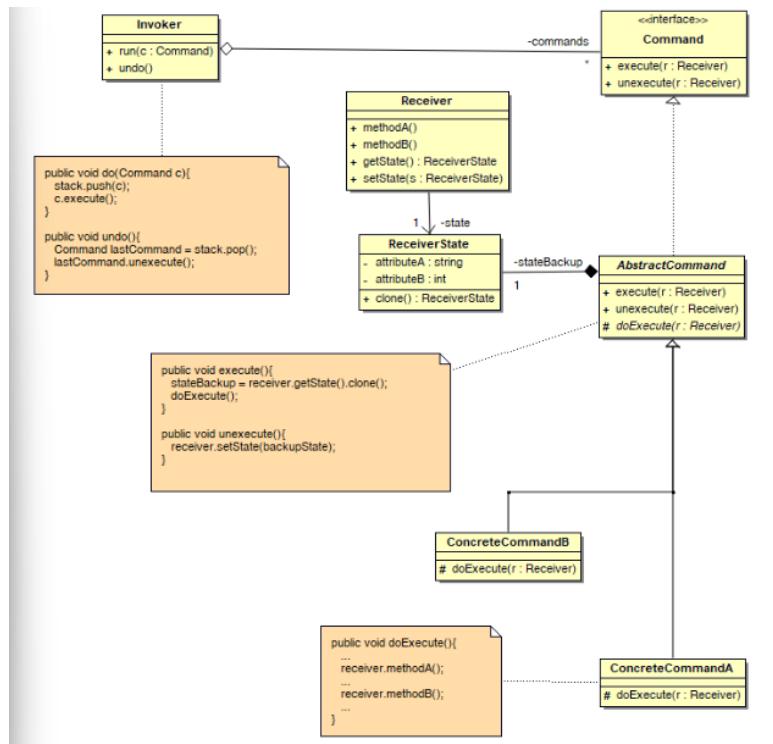


Figura 11.21: Command: schema con stato

Consente di disaccoppiare l'oggetto che invoca il comando dall'oggetto che lo riceve. Gli oggetti **Command** possono essere estesi o composti in oggetti più complessi (esempio: pattern Composite). Aggiungere nuovi comandi è semplice. Potrebbero crearsi accoppiamenti forti tra **Receiver** e oggetti **Command**.

In un sistema per la gestione di eventi, vogliamo che nel momento in cui si aggiunge un partecipante ad un evento, si aggiornino molteplici oggetti dipendenti. Il sistema dovrebbe essere flessibile rispetto agli oggetti da aggiornare. Vogliamo evitare accoppiamenti forti tra gli oggetti. Questa soluzione soddisfa i requisiti?

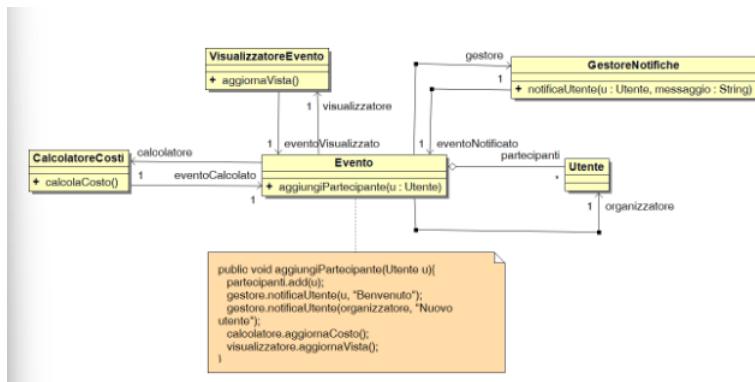


Figura 11.22: Problema: aggiornamento di tanti oggetti a seguito di un evento

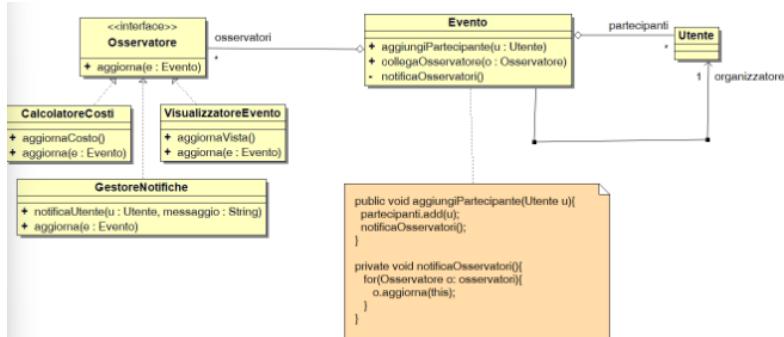


Figura 11.23: Soluzione problema: aggiornamento di tanti oggetti a seguito di un evento

### Observer: scopo

Definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo (il cui numero è imprecisato) siano notificati e aggiornati automaticamente. Esempi di utilizzo: nello schema architettonale MVC, i componenti View e Controller devono essere notificati quando il Model si aggiorna.

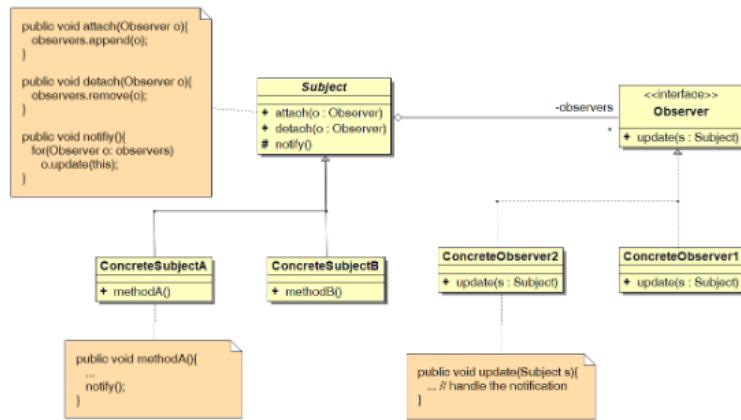


Figura 11.24: Observer: schema

- **Subject:**

- Conosce i propri **Observer**
- Fornisce un’interfaccia per associare e disassociare oggetti **Observer**

- **Observer:** Fornisce un’interfaccia di notifica per gli oggetti a cui devono essere notificati i cambiamenti.

- **ConcreteSubject:**

- Soggetti concreti a cui gli **Observer** sono interessati.
- Inoltrano le notifiche quando il proprio stato si modifica

- **ConcreteObserver:** Implementa l’interfaccia di notifica per mantenere il proprio stato sincronizzato con quello di uno o più **ConcreteSubject**

Gli observer possono osservare uno o più soggetti. Il riferimento ai soggetti osservati può essere memorizzato o meno negli observer. Se viene memorizzato va gestita anche la cancellazione. Il metodo **notify** potrebbe essere pubblico e invocata dai client (utilizzatori) dei soggetti, anziché dai soggetti. Questo evita chiamate multiple o inutili. Il metodo **notify** può contenere parametri utili a comprendere come è cambiato il soggetto. L’accoppiamento tra **Subject** e **Observer** è minima. Possono appartendere a livelli di astrazione diversi nel sistema (ad esempio, a strati diversi in un’architettura a strati). Gli observer possono essere aggiunti dinamicamente. Un aggiornamento ad un soggetto può causare un numero elevato di sincronizzazioni. Può risultare difficile per un observer capire cosa è cambiato nel soggetto.

Un'applicazione gestisce router di vari produttori in ambienti diversi (Linux, Mac, Windows, ecc.). Il design dovrebbe essere sufficientemente flessibile da consentire che in futuro i router possano essere facilmente configurati per ambienti aggiuntivi.

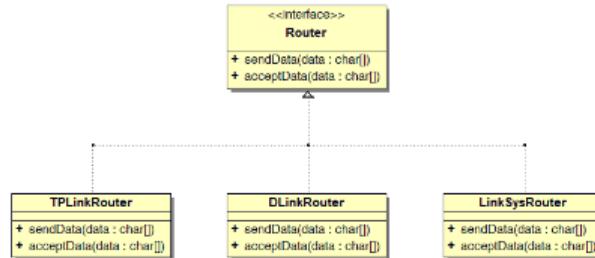


Figura 11.25: Problema: configurazione router

Questa soluzione soddisfa i requisiti?

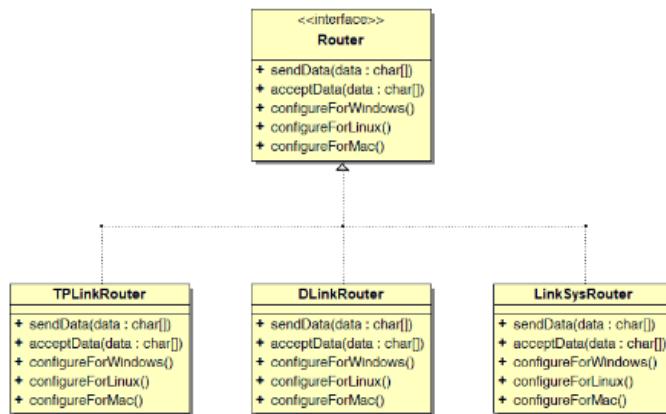


Figura 11.26: Soluzione problema: configurazione router

- Soluzione:

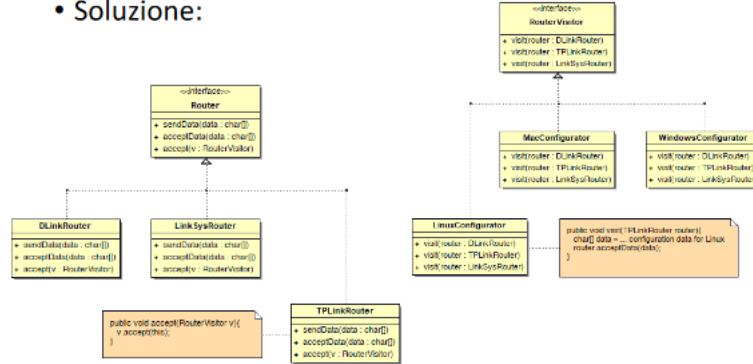


Figura 11.27: Soluzione problema: configurazione router

### Visitor: scopo

Data una gerarchia di classi, consentire la definizione di nuove operazioni su queste classi senza modificarne la definizione. L'implementazione delle operazioni dipende dal tipo specifico di classe e pertanto non può essere portata a fattor comune.

Esempi di utilizzo:

- Quando si vuole evitare di «inquinare» le interfacce delle classi con operazioni che possono cambiare di frequente
- Quando le classi che definiscono la gerarchia cambiano raramente, mentre le operazioni su questi oggetti cambiano spesso

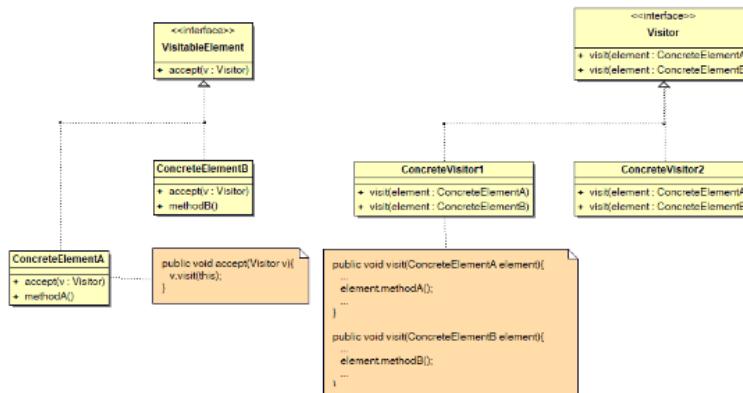


Figura 11.28: Visitor: schema

- **Visitor:** Definisce un metodo *visit* per ogni classe *ConcreteElement*
- **ConcreteVisitor:** implementa i metodi *visit*
- **VisitableElement:** Definisce il metodo *accept*
- **ConcreteElement:** implementa il metodo *accept*

Nuove operazioni sono facili da aggiungere, senza dover modificare le classi della gerarchia. Per contro è difficile aggiungere nuove classi della gerarchia. Può essere utilizzato per visitare le classi di un oggetto composto (es: pattern composite) e le operazioni possono accumulare uno stato mentre visitano gli elementi della composizione. Gli oggetti che accettano un visitor devono implementare dei metodi pubblici sufficientemente potenti e flessibili per essere manipolati dal visitor.

Vogliamo modellare il personaggio di un videogioco che può saltare, correre, camminare, e sparare. L'entità del salto, la velocità della corsa, e la capacità di sparare dipendono però dallo stato corrente del personaggio, che può essere piccolo (salta poco, è lento e non spara), grande (salta tanto ed è veloce ma non spara), o super (salta tanto, è veloce e può pure sparare). Il cambiamento di stato dipende da degli eventi e dallo stato corrente del personaggio (ad esempio, se colpito da un nemico passa allo stato inferiore).

Che problemi ha questa soluzione?

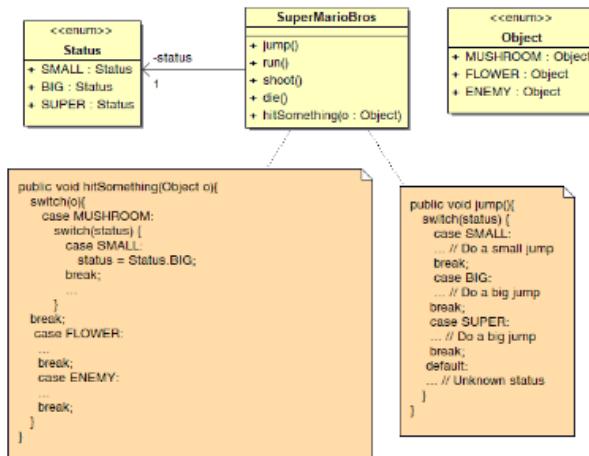


Figura 11.29: Problema. Super Mario Bros

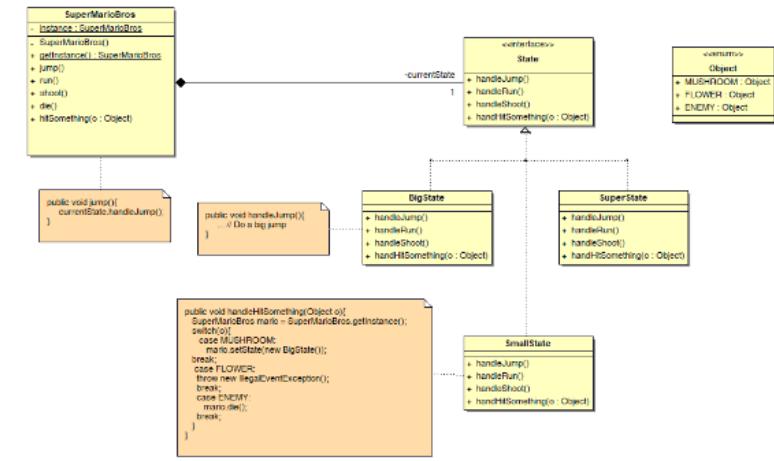


Figura 11.30: Problema. Super Mario Bros

### State: scopo

Permettere ad un oggetto di cambiare il suo comportamento al cambiare del suo stato interno. L'oggetto si comporterà come se avesse cambiato la sua classe. Esempi di utilizzo:

- Protocolli di rete che cambiano in base al proprio stato
- Refactoring di grandi metodi in cui i vari stati sono codificati in blocchi di codice monolitici gestiti da istruzioni condizionali.

## State: schema

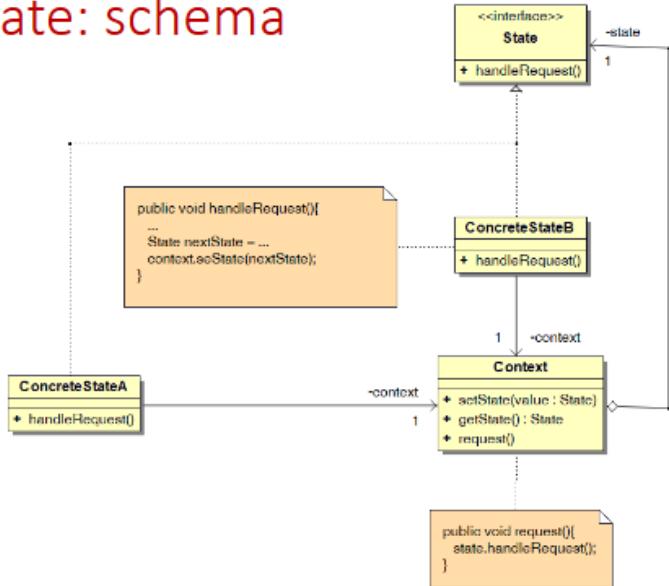


Figura 11.31: State: schema

- **Context:**

- Definisce un’interfaccia usabile dai client
- Contiene un’istanza di un **ConcreteState** che rappresenta il suo stato corrente

- **State:** Definisce un’interfaccia che incapsula il comportamento associato a uno stato particolare di **Context**

- **ConcreteState:** Implementa il comportamento associato a uno stato concreto

Le transizioni di stato sono solitamente gestite dagli stati concreti, tuttavia possono essere gestite anche da **Context** qualora siano chiare e predeterminate. Gli oggetti **State** possono avere visibilità **package** per evitare di essere utilizzati esternamente. Gli oggetti **State** rispetto al **Context** possono essere:

- strettamente legati (composizione), come nell’esempio di SuperMario
- oppure debolmente legati (aggregazione), come ad esempio nel caso di stati di un protocollo di rete condivisi da molteplici connessioni. In questo caso gli oggetti **State** possono essere dei **Singleton**

Il codice risulta così più semplice da comprendere e manutenere. Il comportamento specifico di uno stato è localizzato in un singolo oggetto, e aggiungere nuovi stati è semplice. Le transizioni di stato sono esplicite nel codice e più robuste.

D'altro canto, l'applicazione del pattern aumenta il numero di classi e riduce la compattezza dell'applicazione. Se non usano variabili d'istanza, gli oggetti *State* possono essere condivisi per ridurre il numero di istanze.

# 12. Test del software

## 12.1 Il processo di test

Il processo di test del software ha due obiettivi distinti:

1. Dimostrare allo sviluppatore e al cliente che il software soddisfa i suoi requisiti
2. Scoprire eventuali input (o sequenze di input) per i quali il comportamento del software è errato, indesiderato o non è conforme alle sue specifiche. Tutto questo è causato da difetti (bug) nel software.

**Nota:** I test possono dimostrare soltanto la presenza di errori, non la loro assenza. Il primo obiettivo si raggiunge tramite **test di convalida**. Serie di test che riflettono l'uso previsto del sistema. Il secondo obiettivo è perseguito mediante il **test dei difetti**. Serie di test progettati per scoprire dei difetti, non devono necessariamente riprodurre le condizioni di utilizzo normale.

Le **ispezioni** prevedono che persone esperte osservino il codice sorgente di un sistema e la documentazione a supporto (diagrammi UML, . . .). Si possono ispezionare anche altri documenti, come la specifica dei requisiti o lo schema architetturale. Gli ispettori studiano e apprendono il prodotto ispezionato e sono poi in grado di modificarlo (se necessario). **Vantaggi:**

Le ispezioni non richiedono l'esecuzione del software e molto spesso riescono a scoprire molti errori contemporaneamente (cosa più difficile da fare eseguendo il software). Le ispezioni possono essere applicate anche a software incompleto, senza dover sviluppare test appositi e costosi. Le ispezioni possono valutare anche altre qualità del software, come la portabilità e la manutenibilità. **Svantaggi:**

Le ispezioni non sono adatte a scoprire i difetti dovuti a interazioni impreviste tra varie parti di un programma, i problemi di tempistica tra componenti, o quelli

relativi alle prestazioni di un sistema. Il prodotto non è modificabile durante le ispezioni.

Tipicamente, un sistema software commerciale passa per tre stadi di test:

1. **Test di sviluppo:** effettuati da sviluppatori durante lo sviluppo per scoprire bug e difetti
2. **Test di release:** effettuati su una release del sistema prima del suo rilascio. Servono a verificare che il sistema soddisfi i requisiti prestabiliti.
3. **Test degli utenti:** gli utenti reali del sistema fanno delle prove nel loro ambiente di lavoro.

## 12.2 Test di sviluppo

I test di sviluppo sono svolti tipicamente da chi ha sviluppato il codice, o in alcuni casi da membri del team di sviluppo specializzati in test. Lo scopo principale è quello di scoprire difetti nel software e di evitare che i difetti scoperti possano ripresentarsi in futuro. I test di sviluppo passano per tre stadi.

1. **Test di unità:** vengono testate singole unità di programma (es: classi).
2. **Test dei componenti:** vengono integrate più unità di programma per creare componenti più complessi e si verificano le interfacce di tali componenti.
3. **Test di sistema:** si integrano tutti i componenti e si testa l'intero sistema.

Nella programmazione a oggetti, i test delle unità verificano singole classi testando tutti i loro metodi e ponendo l'oggetto in tutti i suoi possibili stati. Tipicamente questi test vengono automatizzati mediante framework come **JUnit** per Java. Un test automatico si compone di tre parti:

1. Parte di **impostazione**, dove inizializziamo il sistema con input e output previsti.
2. Parte della **chiamata**, dove chiamiamo i metodi da testare.
3. Parte dell'**asserzione**, dove viene confrontato il risultato della chiamata con il risultato previsto. Se l'asserzione è vera il test ha avuto successo, se è falsa il test è fallito.

Vanno progettati due tipi di test case:

- Test case che riflettono il normale comportamento del software.

- Test che usano input anomalo o utilizzi inattesi. Si basano molto anche sull'esperienza del tester.

Un modo per progettare i test case è identificare i gruppi di input che hanno caratteristiche comuni e che dovrebbero essere elaborati allo stesso modo. Questi gruppi sono spesso chiamati **classi di equivalenza dell'input**.

Per ogni classe di equivalenza, andrebbero scelti valori «centrali» e valori vicino ai «confini» della classe.

Esempio: un metodo accetta in input da 4 a 8 numeri. Valori «centrali»: 5,6. Valori di «confine»: 3,4, 8,9.

Partizione di equivalenza	Caratteristica
Nomi corretti 1	Gli input includono solo caratteri alfabetici e sono lunghi da 2 a 40 caratteri.
Nomi corretti 2	Gli input includono solo caratteri alfabetici, trattini o apostrofi e sono lunghi da 2 a 40 caratteri.
Nomi non corretti 1	Gli input sono lunghi da 2 a 40 caratteri ma contengono caratteri non consentiti.
Nomi non corretti 2	Gli input includono solo caratteri consentiti, ma sono lunghi solo 1 carattere o più di 40.
Nomi non corretti 3	Gli input sono lunghi da 2 a 40 caratteri, ma il primo carattere è un trattino o un apostrofo.
Nomi non corretti 4	Gli input contengono solo caratteri validi e sono lunghi da 2 a 40 caratteri, ma includono doppi trattini, testo tra apici o entrambi.

Figura 12.1: Scelta dei test case: esempio per verifica nomi

Metodo **black-box**: classi di equivalenza identificate mediante specifiche del programma.

Metodo **white-box**: classi di equivalenza identificate guardando il codice sorgente del programma. Analizzando il codice di un metodo, possiamo ad esempio identificare i parametri gestiti dalle eccezioni e metterli in una stessa classe di equivalenza.

Un altro modo per scegliere i test case è affidarsi a linee guide che riflettono le esperienze maturate sui tipi di errori più comuni commessi durante lo sviluppo. Ad esempio, se un metodo prende in input un array, possiamo verificare che succede se passiamo un array vuoto, un riferimento nullo, o un array estremamente grande.

Linea guida	Spiegazione
Testare casi limite (di frontiera)	Se la partizione ha dei limiti minimi e massimi (come la lunghezza di stringhe, numeri e così via), si scelgano input ai limiti dell'intervallo.
Forzare errori	Si scelgano input che forzino il sistema a generare tutti i messaggi d'errore. Si scelgano input che dovrebbero generare output non validi.
Riempire i buffer	Si scelgano input che causino l'overflow di tutti i buffer di input.
Ripetersi	Si ripeta più volte lo stesso input o la stessa serie di input.
Overflow e underflow	Se il programma esegue calcoli numerici, si scelgano input che causano il calcolo di numeri molto grandi o molto piccoli.
Non dimenticare null e zero	Se il programma usa puntatori o stringhe, si testi sempre con puntatori e stringhe null. Se si usano sequenze, si testi con una sequenza vuota. Per gli input numerici, si testi sempre con zero.
Tenere il conto	In presenza di liste e trasformazioni di liste, si tenga conto del numero di elementi in ogni lista e si verifichi che siano coerenti dopo ogni trasformazione.
Uno è diverso	Se il programma opera con sequenze, si testi sempre con sequenze che hanno un solo valore.

Figura 12.2: Linee guida di Whittaker

A volte l'oggetto testato ha delle dipendenze con altri oggetti che potrebbero non essere stati ancora implementati, oppure difficili da istanziare o utilizzare in fase di test. Esempi: database, servizi remoti, ...

In questi casi è possibile sviluppare oggetti finti ma dotati di una certa intelligenza, detti **mock**. I mock hanno la stessa interfaccia degli oggetti che simulano. Esempio: simulare un sensore di temperatura con un semplice oggetto che restituisce valori generati automaticamente in un dato range. Alcuni casi in cui usare oggetti mock può essere utile:

- Rimpiazzare un comportamento non deterministico (l'ora o la temperatura ambiente).
- Se l'oggetto ha degli stati difficili da riprodurre (un errore di sistema).
- Se l'inizializzazione dell'oggetto è lunga (creazione di un database).
- Se l'oggetto non esiste ancora o se il suo comportamento può ancora cambiare

I componenti software possono essere formati da molti oggetti che interagiscono tra di loro attraverso le loro interfacce al fine di realizzare una feature del prodotto.

Il test dei componenti verifica che le interfacce dei componenti si comportino conformemente alle loro specifiche. Vi sono vari tipi di interfacce:

- **Interfacce di parametri:** servono a passare dati e riferimenti tra componenti.
- **Interfacce a memoria condivisa:** blocchi di memoria condivisi tra componenti (ad esempio, registri usati in sistemi integrati).
- **Interfacce procedurali:** serie di procedure richiamabili da altri componenti.
- **Interfacce a passaggio di messaggi:** comunicazione tra oggetti mediante messaggi (ad esempio, nei sistemi client-server)

Possibili categorie di errori:

- Uso errato dell'interfaccia. Frequente nelle interfacce di parametri, in cui i parametri possono essere passati in modo errato.
- Incomprensione delle interfacce. Dovuto a interfacce poco chiare (problemi ad esempio nel naming di metodi e parametri, scarsa documentazione).
- Errori di tempistica. Nelle interfacce a memoria condivisa o a passaggio di messaggi, due componenti diversi potrebbero lavorare a velocità diverse e non compatibili

Linee guida per test delle interfacce:

- Come per i test di unità, per le interfacce di parametri coprire tutte le possibili chiamate e tentare diversi possibili classi di parametri in input. In caso di passaggio di riferimenti, testare il passaggio di valori nulli.
- Per interfacce procedurali, cercare di far fallire le procedure.
- Fare stress test per interfacce basate su passaggio di messaggi al fine di individuare errori di tempistica.
- Variare l'ordine in cui i componenti interagiscono con i dati in interfacce a memoria condivisa.

I test di sistema riguardano il sistema integrato con tutti i suoi componenti. Servono a testare il **comportamento emergente** di un sistema complesso, ovvero quell'insieme di proprietà macroscopiche, difficilmente predibili esaminando i singoli componenti. Si focalizzano sulle **interazioni** tra i componenti e gli oggetti che formano il sistema. Identificano difetti che emergono solo in corrispondenza di queste interazioni. I test di sistema spesso implementano i casi d'uso del sistema (**end-to-end**), in modo da scatenare tutte le interazioni necessarie tra i vari componenti. Andrebbero testate:

- Tutte le funzioni a cui l'utente ha accesso (tipicamente mediante un'interfaccia grafica).
- Tutte le combinazioni delle funzioni sopra indicate.
- Tutti gli input (corretti ed errati) fornibili dall'utente.

Automatizzare i test di sistema è piuttosto complesso. La scelta dei casi d'uso da testare è molto importante.

	Unit	End-toEnd
Fast		
Reliable		
Isolates Failures		
Simulates a Real User		

Figura 12.3: Confronto

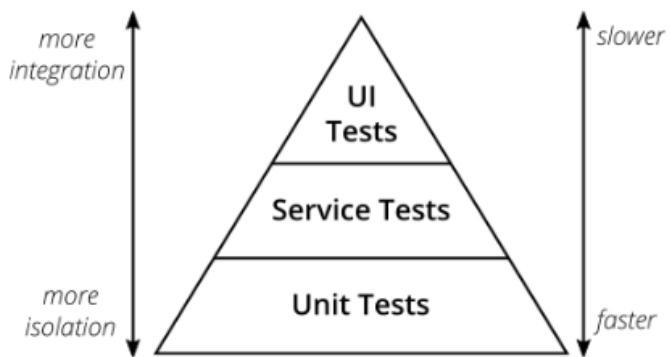


Figura 12.4: La piramide dei test

Lo sviluppo guidato dai test (**test-driven development**) è un approccio allo sviluppo di programmi in cui si interlacciano sviluppo e test del codice. Il test diventa una parte integrante dello sviluppo. Viene introdotto come parte del metodo di sviluppo agile XP. Sviluppo incrementale accompagnato da una serie di test per ogni incremento.

Fasi principali:

- Identificazione del nuovo incremento da sviluppare. Funzionalità implementabile con poche linee di codice
- Scrittura del nuovo test per la nuova funzionalità da introdurre.
- Esecuzione nuovo e vecchi test. La prima volta il nuovo test DEVE fallire per dimostrare che sta ampliando la suite di test. Implementazione del codice e ripetizione dei test fino al loro superamento. Possibile rifattorizzazione del codice esistente.

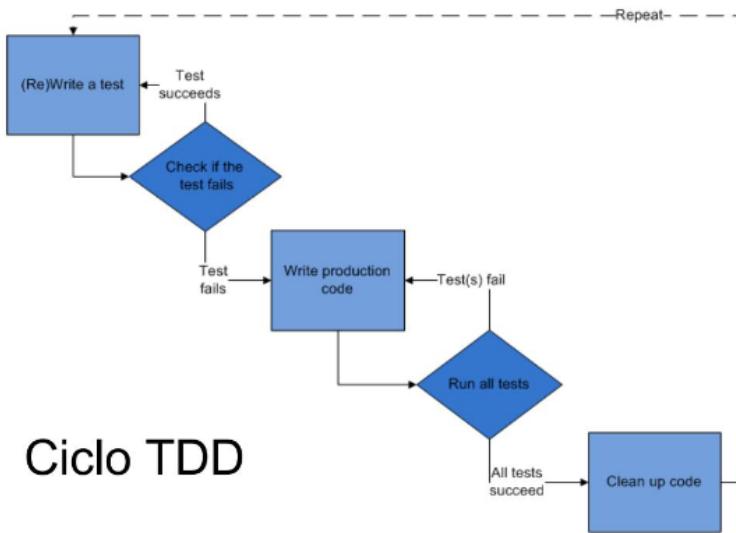


Figura 12.5: Ciclo TDD

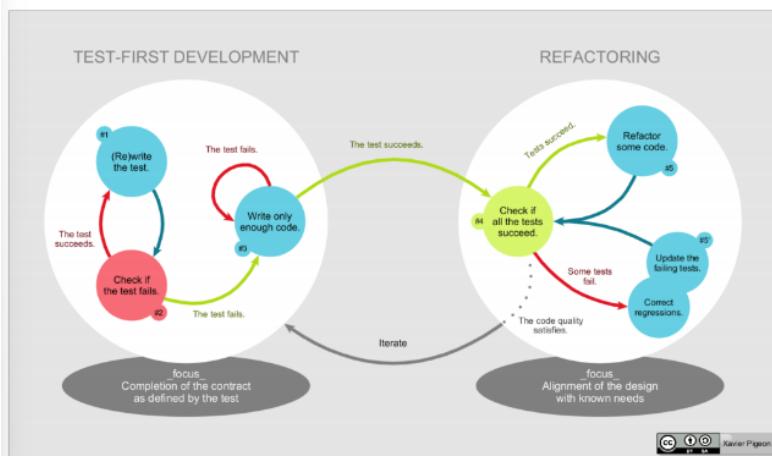


Figura 12.6: Test-first development

### Vantaggi:

- Gli sviluppatori capiscono meglio cosa deve fare il codice da sviluppare.
- Forte copertura del codice testato
- Test di regressione inglobati nel processo (tipicamente costosi).

- Debugging più semplice.
- I test sono una forma di documentazione del sistema che ne migliora la comprensibilità.

#### Svantaggi:

- Poco adatto per testare grossi blocchi di codice riusato o sistemi legacy.
- Può essere inefficiente per sistemi distribuiti o multithread.
- I test di convalida vanno comunque svolti.

### 12.3 Test di release

Il test delle release è il processo che testa una particolare release di un sistema prima che venga distribuita all'esterno. Processo di convalida che controlla se un sistema soddisfa i suoi requisiti (funzionalità, prestazioni,...). L'obiettivo non è trovare nuovi difetti ma convalidare il sistema. Test basato sui requisiti:

- I requisiti devono essere scritti in maniera da poter essere facilmente testabili.
- Per ogni requisito si sviluppa una serie di test case.
- Si può costruire una **matrice di tracciabilità** che mappa ogni requisito ai test case che lo riguardano.

Anziché testare i singoli requisiti, è possibile testare una release basandosi su degli scenari di utilizzo. Scenari credibili e abbastanza complessi ma facili da valutare. Le storie utente sono un esempio di scenario. Questo approccio testa più requisiti all'interno di un unico scenario. Adatto a verificare la combinazione di più funzionalità insieme.

Oltre alla correttezza, all'affidabilità e alla robustezza, è possibile testare anche altre qualità esterne del software:

- Prestazioni
- Usabilità
- ...

Per valutare le prestazioni di un sistema è possibile effettuare degli **stress test** che impegnano il sistema con richieste oltre i limiti previsti. Il sistema deve fallire «gradualmente», evitando un collasso improvviso. Possono verificarsi difetti che emergono solo quando il sistema lavora a pieno carico.

I test di usabilità valutano l’interazione tra utente e sistema. Forniscono un input diretto su come gli utenti reali utilizzano il sistema (spesso in modo molto diverso da come era stato previsto). L’obiettivo è quello di rivelare eventuali motivi di confusione e scoprire opportunità per migliorare l’esperienza complessiva dell’utente. Modalità per test di usabilità:

- Moderati/liberi
- Utenti selezionati/liberi/crowdsourced
- Comparativi (con altri software analoghi)
- Con l’ausilio di sensori (esempio: mouse-tracking, eye-tracking,...)
- ...

## 12.4 Test degli utenti

Il test degli utenti è una fase del processo di test di un sistema in cui gli utenti danno i loro feedback. I condizionamenti derivanti dall’ambiente di lavoro possono avere effetti rilevanti sulle prestazioni, affidabilità, robustezza e usabilità di un sistema. Esempio: cosa succede se devo rispondere al telefono e lascio una procedura incompleta per un’ora e poi la termine oppure la interrompo bruscamente? Gli **alpha** test sono test eseguiti internamente dall’azienda che produce software, col supporto di utenti selezionati. Gli utenti forniscono informazioni sul reale utilizzo del software che possono essere sfruttati per sviluppare scenari di test più plausibili. I **beta** test sono effettuati da alcuni utenti selezionati (o da tutti), avvertendo gli utenti che il prodotto distribuito potrebbe non essere definitivo e probabilmente richiede ulteriori correzioni. Gli utenti della versione beta possono utilizzare il software in casi realistici, e inviare al produttore resoconti dei malfunzionamenti riscontrati. Utile quando tutte le possibili configurazioni di utilizzo di un prodotto sono troppe o impossibili da prevedere (esempio: app mobile). Le versioni beta sono a volte usate anche come forma di commercializzazione del prodotto. I clienti imparano a conoscere e usare il sistema, aumentando la probabilità di acquisto della versione commercializzata.

I **test di accettazione** sono svolti dal committente che deve decidere se accettare o meno il prodotto. Tipicamente avvengono nel reale ambiente di lavoro, con dati reali. Richiedono dei criteri di accettazione stabiliti nelle fasi iniziali del processo. I test di accettazione vengono quindi derivati dai criteri di accettazione. I risultati dei test necessitano spesso di essere negoziati. A volte il cliente vuole usare il software anche in presenza di difetti. Il fornitore deve impegnarsi a risolvere i difetti velocemente. Nei metodi agili, i test di accettazione potrebbero mancare.

L'utente finale è spesso parte del team di sviluppo (esempio: product owner) e partecipa alla definizione delle storie utente. Limiti di questo approccio:

- Condizioni di utilizzo non realistiche.
- Utenti coinvolti non rappresentativi di tutti gli utenti.
- Test automatizzati poco flessibili.