

INGEGNERIA DEL SOFTWARE



**Università
degli Studi
di Palermo**

A.A. 2022-2023



Capitolo 1: INTRODUZIONE AL SOFTWARE	
- Introduzione al mondo dietro i Software	pag. 3
- Ciclo di vita di un Software	pag. 9
- Metodologia Agile	pag. 13
- Unified Model Language	pag. 19
<hr/>	
Capitolo 2: PROGETTAZIONE DEL SOFTWARE	
- Il ponte tra l'analisi e la progettazione	pag. 29
- Modelli architetturali	pag. 31
- Design Patterns	pag. 37
- Creational Design Pattern	pag. 41
- Structural Design Pattern	pag. 51
- Behavioral Design Pattern	pag. 61
<hr/>	
Capitolo 3: SVILUPPO DEL SOFTWARE	
- I requisiti	pag. 67
- Scrum	pag. 69
- I test	pag. 75
- Sistemi fidati	pag. 79

CAPITOLO 1:

INTRODUZIONE AL SOFTWARE

INTRODUZIONE AL MONDO DIETRO I SOFTWARE

Ad oggi, se ci chiedessimo quanto influirebbe un Software (SW) sulla vita di una persona, la risposta è “tanto”. L’uso di tempo digitale è aumentato notevolmente e solitamente riguarda campi medici e social, ma non solo, altri ambiti sono ad esempio lo sviluppo del livello di sicurezza dei voli e molto altro...



Il sistema MCAS degli aerei Boeing 737 è un software che corregge l’assetto dell’aereo e lo stabilizza.

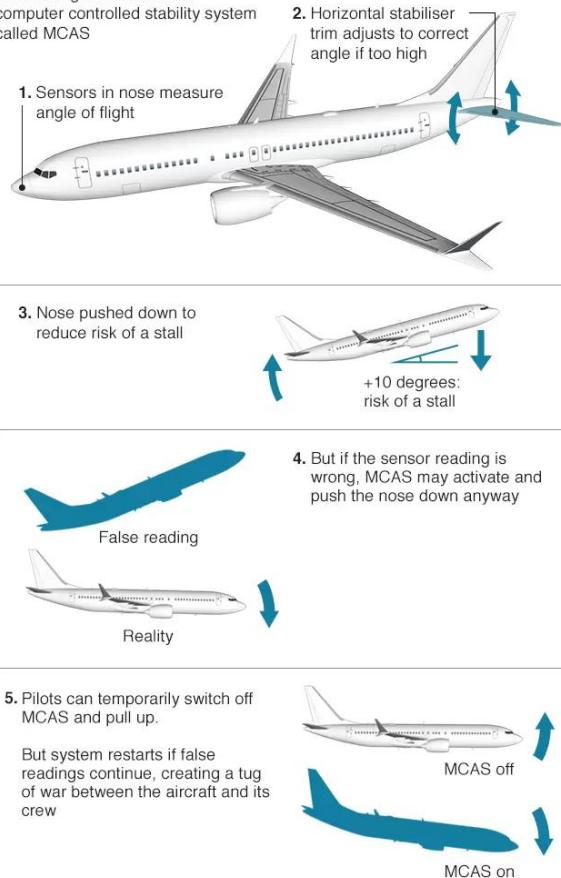
Il suo (mal)funzionamento potrebbe aver causato gli incidenti in Etiopia (2019) e Indonesia (2018) - oltre 300 morti. Sia le autorità federali statunitensi che il Congresso stanno indagando per capire se Boeing o la FAA avessero sottovalutato i rischi collegati all’uso di questo sistema.

In particolare, vogliono capire se la FAA abbia valutato il MCAS rigorosamente, basandosi su comprovati standard ingegneristici e di design, oppure se, per favorire Boeing, abbia seguito scorciatoie che non hanno garantito adeguati livelli di sicurezza di volo o una sufficiente formazione ai piloti.

All’aereo Boeing di Ethiopian Airlines, così come a quello di Lion Air precipitato in Indonesia, mancava un sistema di sicurezza in grado di avvisare i piloti di eventuali problemi con il sistema MCAS. Il software di sicurezza, ritenuto non essenziale, era venduto da Boeing come extra e non era stato comprato né da Ethiopian Airlines né da Lion Air.

How the MCAS system works

The Boeing 737 Max has a computer controlled stability system called MCAS



Quello che differisce tra gli ambiti e le nostre competenze è la conoscenza di come sviluppare il software affinché sia completo (fase di test, debug, strutturazione, ...).

Indipendentemente dal fatto che ci siano degli sviluppatori esperti o alle prime armi, ci sono delle leggi sullo sviluppo del software che possono essere utili a risolvere dei problemi.

Tra gli esempi tipici ci si ritrova spesso nella seguente

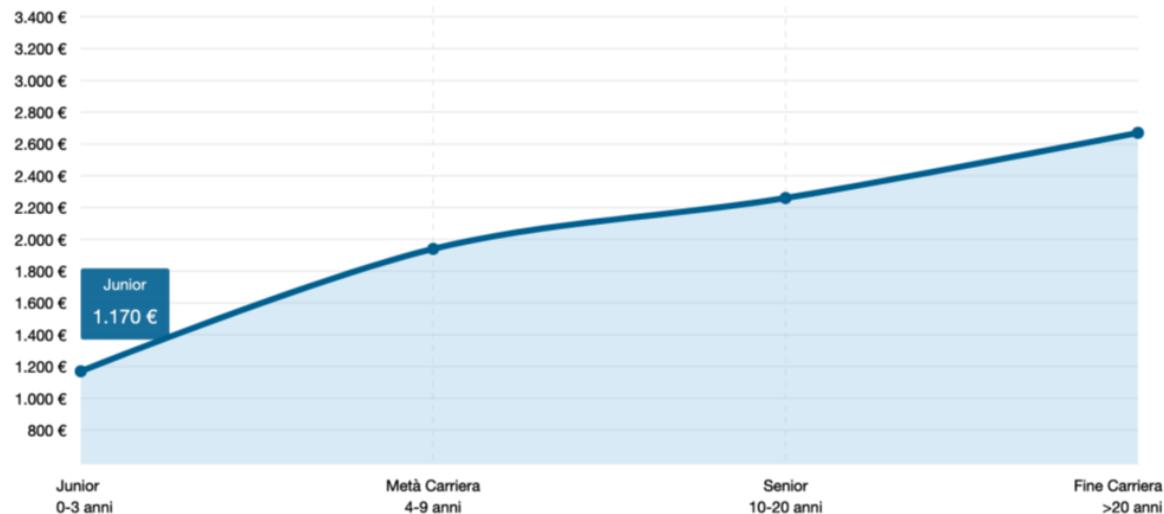
Legge di Eagleson: qualsiasi codice che non hai rivisto negli ultimi sei mesi o più ti sembrerà estraneo, come se l'avesse scritto qualcun altro.

Per ovviare a tale problema, col tempo, si sono sviluppati dei **design patter**, ovvero dei pattern di scrittura del codice, e della documentazione, che permettono di essere compresi (e ricompresi) facilmente da altri.

Ma perché c'è bisogno dei design pattern? Il software non rimane mai lo stesso, cambia costantemente con i tempi, le mode e le nuove esigenze.

Legge di Conway: le organizzazioni che progettano software sono indotte a generare design che sono copie delle strutture di comunicazione di tali organizzazioni.

Quanto guadagna chi sviluppa software? In media segue questo schema



Ma rispetto a quali criteri vengono valutati economicamente lo sviluppo dei software?

Gi aspetti economici sono:

- Ambito d'uso (dimensione in righe);
- Piano di sviluppo (durata);
- Sforzo di sviluppo (costo);
- Produttività del team;
- Qualità (difetti).

Es. sviluppo di un videogioco

Sforzo tipico: 100/500 anni/persona;

Team: di solito 50/100 persone

(Assassin Creeds 2009/450 persone)

Vendere un milione di copie è ok ma non eccellente.

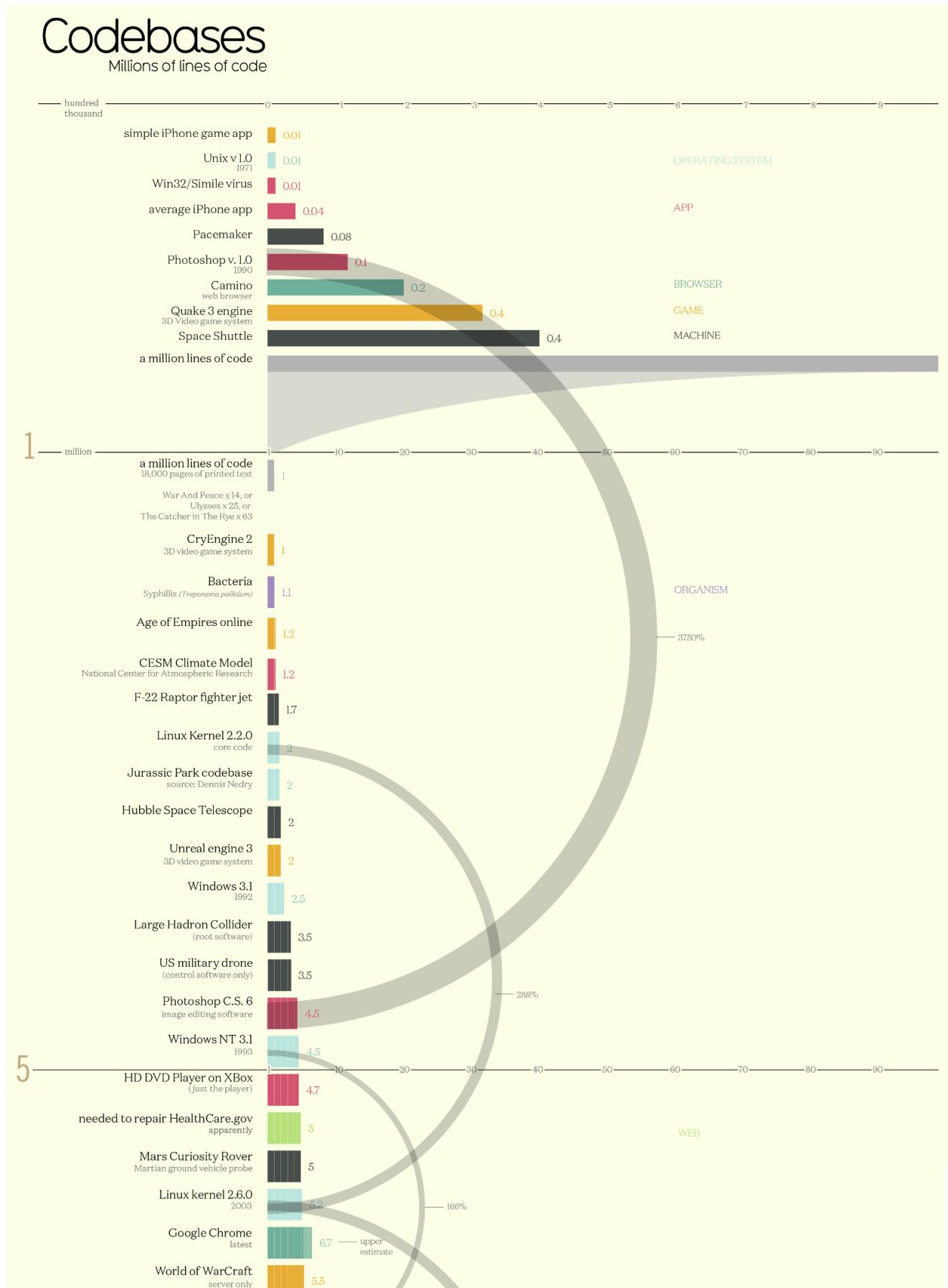
Which Country Has the Best Developers?

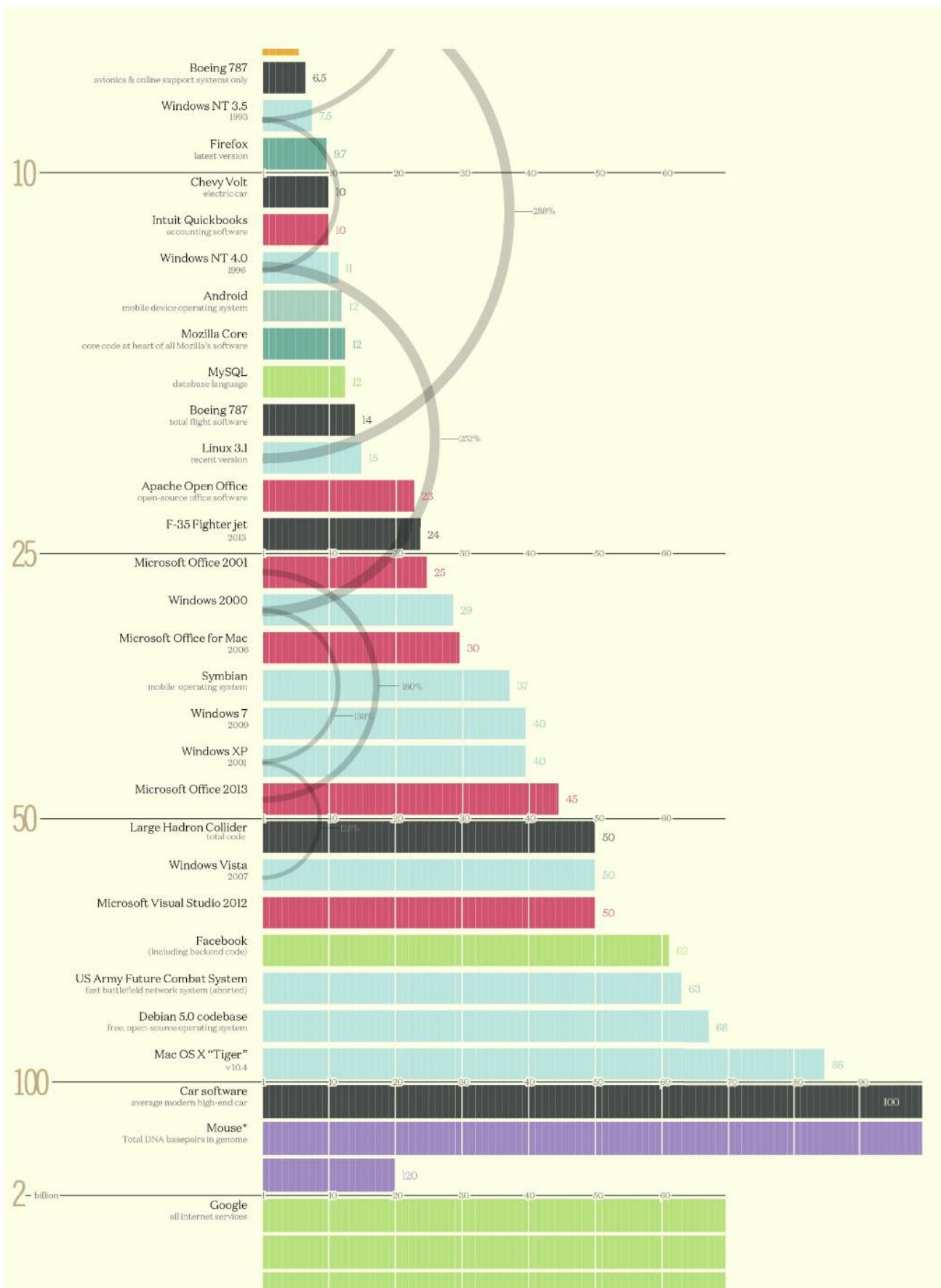
Ranked by Average Score Across All HackerRank Challenges

Rank	Country	Score Index	Rank	Country	Score Index
1	China	100.0	26	Netherlands	78.9
2	Russia	99.9	27	Chile	78.4
3	Poland	98.0	28	United States	78.0
4	Switzerland	97.9	29	United Kingdom	77.7
5	Hungary	93.9	30	Turkey	77.5
6	Japan	92.1	31	India	76.0
7	Taiwan	91.2	32	Ireland	75.9
8	France	91.2	33	Mexico	75.7
9	Czech Republic	90.7	34	Denmark	75.6
10	Italy	90.2	35	Israel	74.8
11	Ukraine	88.7	36	Norway	74.6
12	Bulgaria	87.2	37	Portugal	74.2
13	Singapore	87.1	38	Brazil	73.4
14	Germany	84.3	39	Argentina	72.1
15	Finland	84.3	40	Indonesia	71.8
16	Belgium	84.1	41	New Zealand	71.6
17	Hong Kong	83.6	42	Egypt	69.3
18	Spain	83.4	43	South Africa	68.3
19	Australia	83.2	44	Bangladesh	67.8
20	Romania	81.9	45	Colombia	66.0
21	Canada	81.7	46	Philippines	63.8
22	South Korea	81.7	47	Malaysia	61.8
23	Vietnam	81.1	48	Nigeria	61.3
24	Greece	80.8	49	Sri Lanka	60.4
25	Sweden	79.9	50	Pakistan	57.4

Se volessimo sapere quante righe sono lunghi i software, avremmo il seguente schema:

<https://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

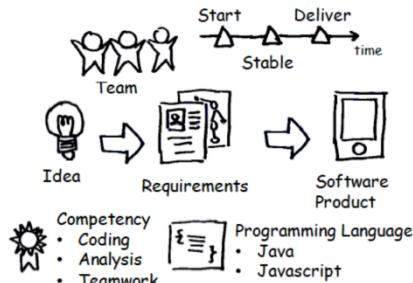




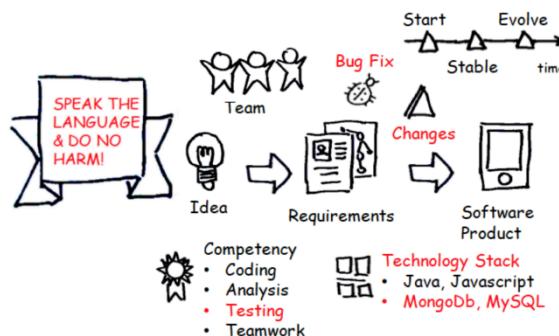
Supponiamo di aver intenzione di modificare un software, ci chiediamo: è più difficile leggerlo o scriverlo?

Sebbene la risposta più banale sarebbe "leggerlo" di fatto non è così, da un punto di vista studentesco leggere un software prodotto da qualcun altro, sì che è difficile ma dipende, perché lo è altrettanto scriverlo da zero.

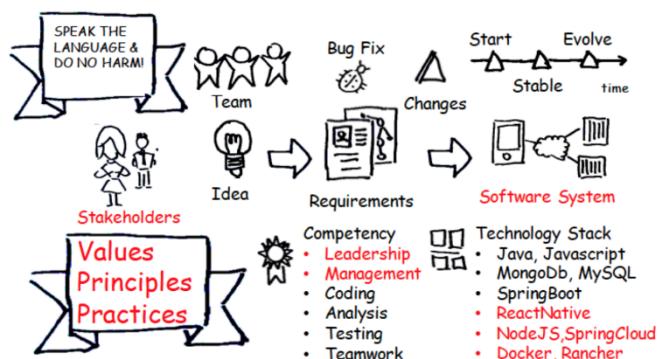
PUNTO DI VISTA: STUDENTE



PUNTO DI VISTA: TESISTA IN AZIENDA



PUNTO DI VISTA: PROFESSIONISTA



Più si cresce a livello pratico e professionale, più si possono notare come durante le fasi di sviluppo possano nascere nuovi problemi ed imprevisti (può capitare che bisogna cambiare l'intero progetto), di conseguenza anche la scrittura del codice diventa difficile, oneroso e sconfortante.

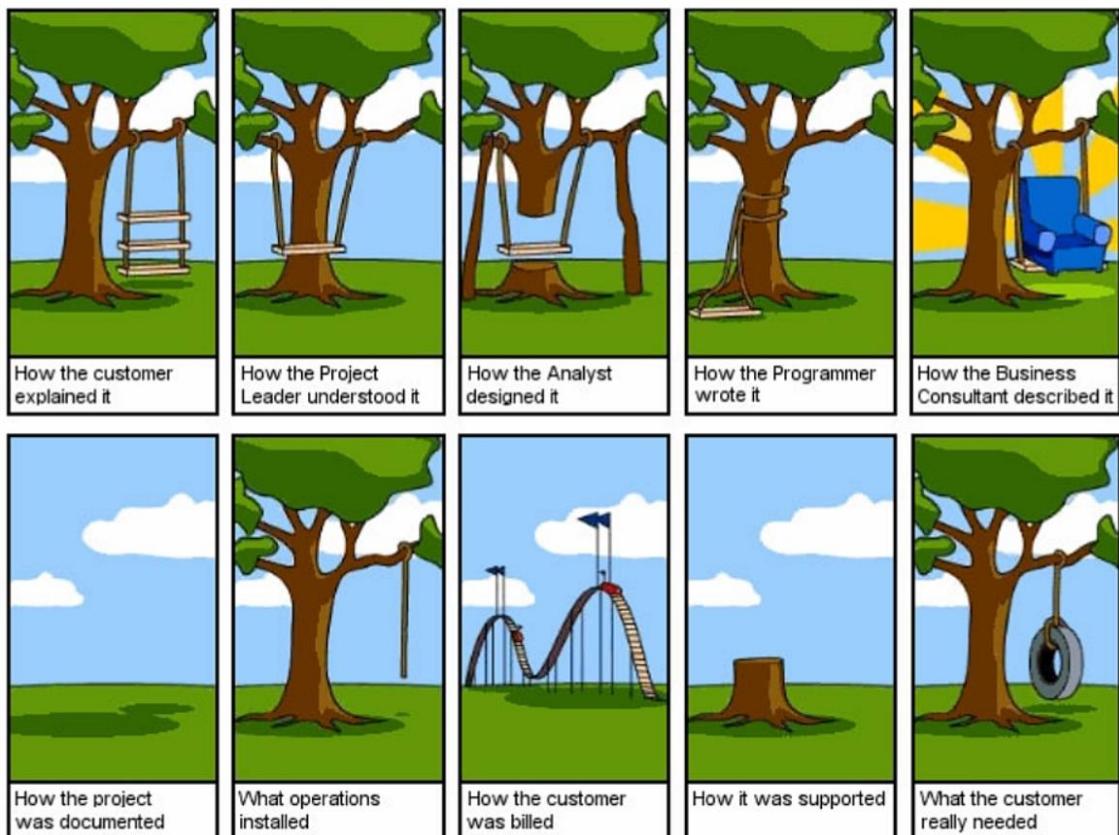
Nella "realtà dei fatti" si ha:

- *Entusiasmo;*
- *Disillusione;*
- *Panico;*
- *Ricerca del colpevole;*
- *Punizione dell'innocente;*
- *Lodi e onori a chi non si è fatto coinvolgere.*

Non tutti gli sviluppi si svolgono così: in quelli di successo la differenza la fanno sempre le **persone** e le **regole** (il **processo di sviluppo**) che seguono.

Lo sviluppo in team è molto diverso dallo sviluppo “personale”, nel team ci sono persone con esperienze diverse, che ricoprono diversi ruoli che hanno diverse abilità:

- Come progettare il prodotto software (**architetti**);
- Come costruire il prodotto SW (**programmatori**);
- A cosa serve il prodotto SW (**esperti di dominio**);
- Come va fatta l’interfaccia utente (**progettisti di interfaccia**);
- Come va controllata la qualità del prodotto SW (**testatori**);
- Come usare le risorse di progetto (**project manager**);
- Come riusare il software esistente (**gestori delle configurazioni**).



Vedremo che l'**Ingegneria del SW (Software Engineering)** è una disciplina **metodologica**, cioè studia i metodi di produzione, le teorie alla base dei metodi, e gli strumenti di sviluppo e misura della qualità dei sistemi software.

È anche una disciplina **empirica**, cioè basata sull’esperienza e sulla storia dei progetti passati.

CICLO DI VITA DI UN SOFTWARE

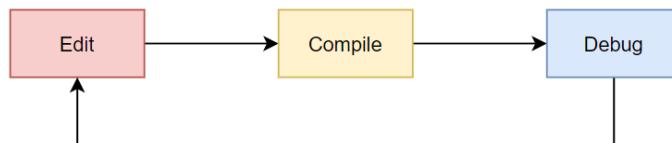
Un **processo di sviluppo del software** (o “**processo software**”) definisce Chi fa Cosa, Quando, e Come, allo scopo di conseguire un certo risultato, o più semplicemente, è un insieme di attività che costruiscono un prodotto software.

Il prodotto può essere costruito da zero o mediante riuso di software esistente (asset) che viene modificato o integrato.

Il **tempo** è una risorsa fondamentale, saperlo gestire durante la fase di sviluppo può risultare difficile. Più tempo si perde durante una specifica fase dello sviluppo, più ci saranno ritardi e di conseguenza anche i costi aumenteranno.

Possiamo dividere la programmazione in tre classi:

- **Programming in-the-small:** un programmatore, un *modulo* = *edit – compile – debug*;



Tale classe ha dei vantaggi, ma il lavoro di un singolo programmatore presenta anche degli svantaggi.

Vantaggi:

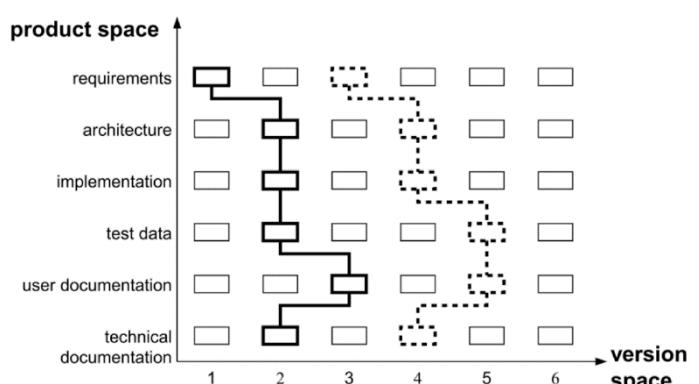
- Molto veloce, feedback rapido;
- Disponibilità di molti strumenti.

Svantaggi:

- Specializzato per la codifica;
- Non incoraggia la documentazione;
- Il processo non scala: in-the-large, in-the-many;
- Ingestibile durante la manutenzione;
- Il debugging ha bisogno di un processo specifico.

- **Programming in-the-large:** costruire software decomposto su più:

- **Configuration Item:** Un “elemento atomico” (**modulo**) gestito nel processo di gestione delle configurazioni non solo file sorgenti ma tutti i tipi di documenti in alcuni casi anche hardware;
- **Versioni:** stato di un configuration item in un determinato istante di tempo;
- **Configurazioni:**
 - Baseline: una specifica o un prodotto che è stato revisionato e approvato formalmente dal management, in modo tale che possa essere una base per ulteriori sviluppi, oppure può essere modificato solo mediante procedure formalmente controllate
 - Release: “promozione” di una baseline resa visibile anche al di fuori del team di sviluppo (per esempio al cliente).



- **Programming in-the-many:** costruire grandi sistemi SW richiede la cooperazione ed il coordinamento di più sviluppatori, nell’ambito di un ciclo di vita.

Il software non è mi statico, ci possono essere più specifiche o nuove scelte da parte del cliente, pertanto:

"Il software è flessibile e può cambiare"

I requisiti cambiano perché cambia l'ambiente della attività, quindi il software che sostiene tali attività deve evolvere; spesso capita che il software deve essere variato durante la fase di mantenimento.

Anche se si può distinguere lo sviluppo dall'evoluzione (*manutenzione*) la demarcazione tra le due fasi è sfumata, perché pochi sistemi sono del tutto nuovi.

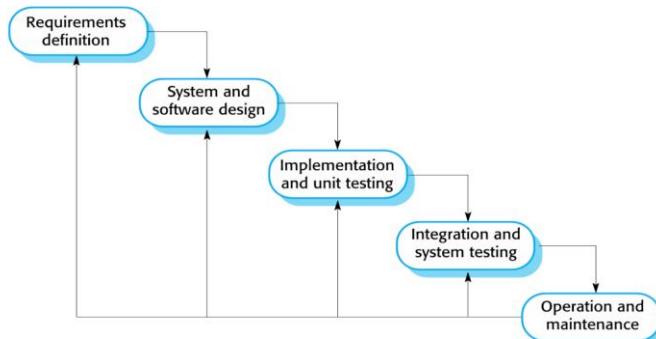
Distinguiamo principalmente due modalità di sviluppo del software:

- **Plan-driven processes**, dove tutte le attività dei processi sono pianificate in *avanzamento* e il loro *progresso* è misurato in base al piano;
- **Metodo agile**, dove la pianificazione è incrementale ed è più facile cambiare il processo in base ai cambiamenti dei requisiti da parte del cliente.

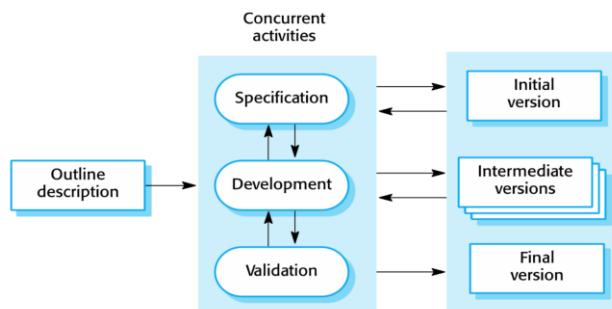
Nota: nel corso vedremo in particolare la metodologia agile.

Nella pratica, molti metodi pratici includono elementi di entrambi gli approcci, non esiste uno giusto o uno sbagliato. Tra i metodi di sviluppo del software più noti abbiamo:

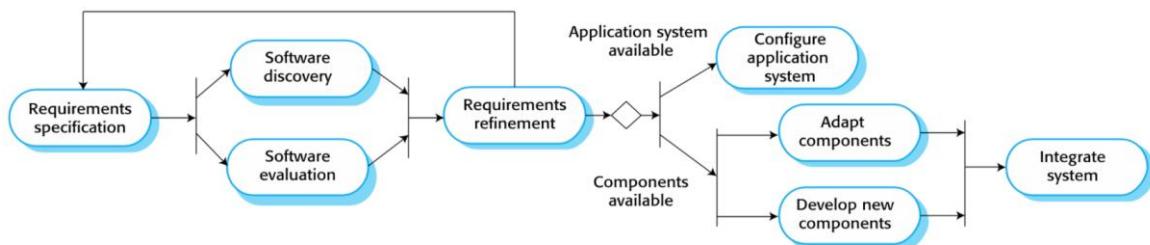
- Il **modello a cascata**, *plan-driven process*, separa e distingue le fasi di specifica e sviluppo.



- Lo **sviluppo incrementale**, *plan-driver process o agile*, le specifiche, sviluppo e validazione sono intercalati.



- **Integrazione e configurazione**, *plan-driver process o agile*, il sistema è assemblato da componenti configurabili già esistenti.



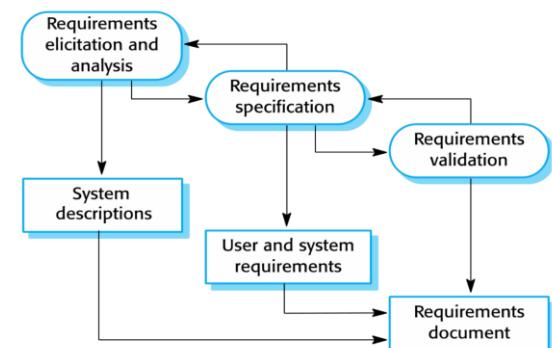
I modelli software reali sono sequenze intercalate di tecniche, attività collaborative e gestionali con l'obiettivo generale di specificare, progettare, implementare e testare un sistema software.

Le quattro attività di processo di base di *specificazione*, *sviluppo*, *validazione* ed *evoluzione* sono organizzate in modo diverso in diversi processi di sviluppo.

Ad esempio, nel modello a cascata, sono organizzati in sequenza, mentre nello sviluppo incrementale sono intercalati. Facendo un esempio

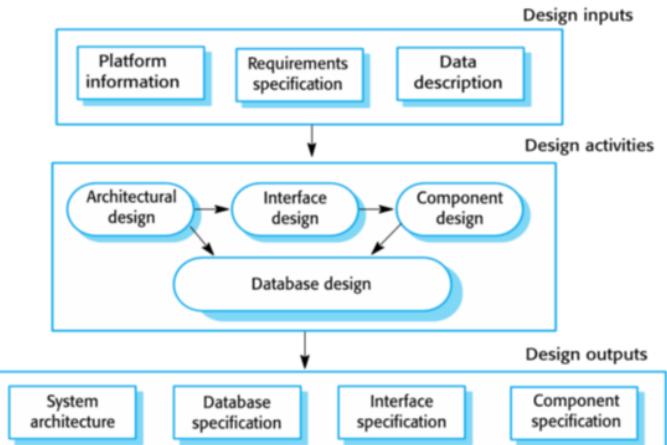
L'**ingegnerizzazione dei requisiti** è il processo per stabilire quali servizi sono richiesti e stabilire i vincoli al funzionamento e allo sviluppo del sistema. Dove:

- **Requirements elicitation and analysis**, ci dice cosa gli stakeholder si aspettano e richiedono dal sistema;
- **Requirements specification**, definisce i requisiti nel dettaglio;
- **Requirements validation**, conferma la validità dei requisiti.



Il processo di conversione delle specifiche del sistema in un sistema eseguibile si trascrive in due fasi:

- **Software design**, design della struttura del software che realizza le specifiche, dove:
 - **Architectural design**, identifica la struttura complessiva del sistema, i componenti principali (sottosistemi o moduli), i loro relazioni e come sono distribuite;
 - **Database design**, si progettano le strutture dati del sistema e come queste devono essere rappresentate in un database;
 - **Interface design**, in cui si definiscono le interfacce tra i componenti del sistema;
 - **Component selection and design**, in cui si ricerca il riutilizzabile, se non c'è si stabilisce come funzionerà.
- **Implementazione**, traduce la struttura in un programma eseguibile, il software viene implementato sviluppando un programma o programmi o configurando un sistema applicativo.

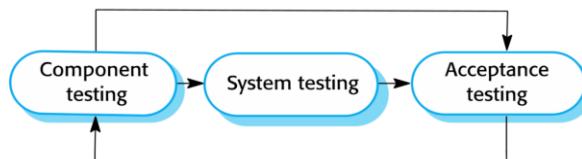


La progettazione e l'implementazione sono attività intercalate per la maggior parte dei tipi di sistemi software

La programmazione è un'attività individuale senza un processo standard, il debug è l'attività di ricerca e correzione degli errori del programma questi difetti.

La **Verifica e conValida (V&V)** ha lo scopo di dimostrare che un sistema è conforme alle sue specifiche e soddisfa i requisiti del cliente del sistema. Comporta il controllo e la revisione dei processi e dei test del sistema.

Il test del sistema comporta l'esecuzione del sistema con casi di test che sono derivati dalla specifica dei dati reali da elaborare dal sistema. Il test è l'attività V&V più comunemente utilizzata.

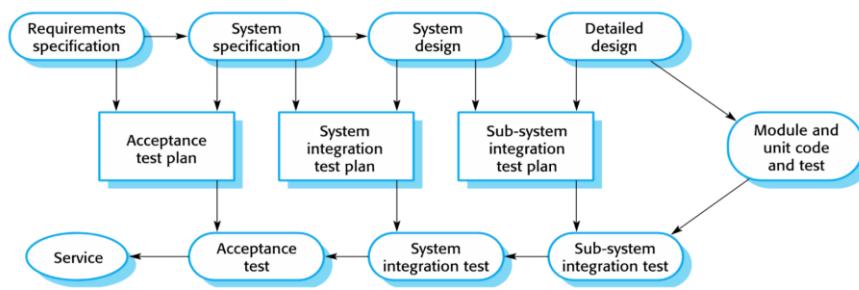


L'**MVP (Minimum Viable Product)** è una prima versione del prodotto finale che si sta andando a creare, questa serve avere una convalida del prodotto stesso e, se ha "successo", allora comincia pian piano a trasformarsi nel prodotto finale.

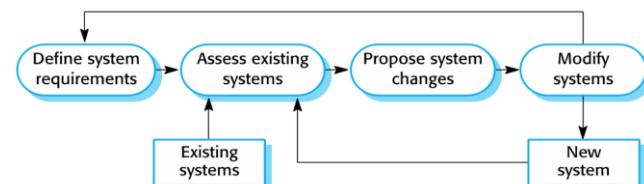
Dopo l'MVP si passa ad una versione **Alfa**, dove gli alfa-tester hanno il compito di fornire feedback per dichiarare se il prodotto può essere immesso nel caso reale.

Dopo la versione Alfa si passa alla **Beta**, dove, come gli alfa-tester, i beta-tester hanno il compito di fornire feedback per eventuali modifiche, bug, ... (la differenza tra i due tipi di tester è che nel primo caso il test è fatto in "laboratorio" nel secondo in ambiente esterno).

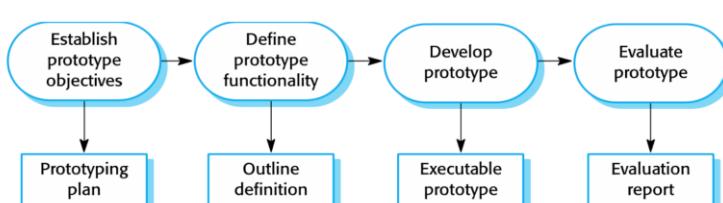
In conclusione la Verifica è la fase di testing durante lo sviluppo, mentre la conValida è la di testing sopradescritta.



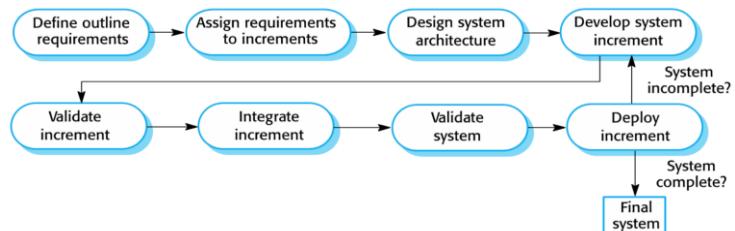
Fasi di testing in uno sviluppo plan-driven processes (V-Model)



Evoluzione del sistema



Processo dello sviluppo del prototipo



Consegna incrementale

METODOLOGIA AGILE

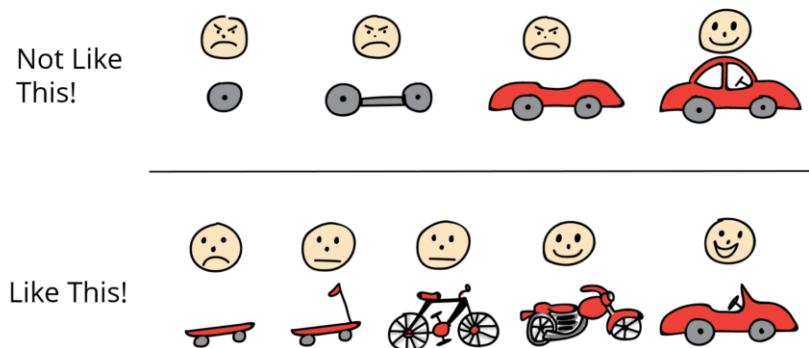
I metodi agili sono una famiglia di metodi di sviluppo che hanno in comune:

- Rilasci frequenti del prodotto sviluppato;
- Collaborazione continua del team di progetto col cliente;
- Documentazione di sviluppo ridotta;
- Valutazione sistematica e continua di valori e rischi dei cambiamenti.

Nello sviluppo di un nuovo prodotto, il prodotto minimo funzionante (MVP) è il prodotto con il più alto ritorno sugli investimenti rispetto al rischio: è una strategia mirata ad evitare di costruire prodotti che i clienti non vogliono, che cerca di massimizzare le informazioni apprese sul cliente per ogni euro speso.

Un MVP non è, quindi, un prodotto minimo, ma un processo iterativo di generazione di idee, prototipazione, presentazione, raccolta dati, analisi ed apprendimento.

È un fondamento del movimento “Lean”, alla base di Agile.



Stiamo scoprendo modi migliori di costruire il software facendolo e aiutando altri a farlo. Attribuiamo valore a:

- Individui e interazioni più che a processi e strumenti;
- Software che funziona più che a documentazione completa;
- Collaborazione col cliente più che a negoziazione contrattuale;
- Reagire al cambiamento più che a seguire un piano.

I valori a **destra** sono importanti, ma noi preferiamo quelli a **sinistra**.

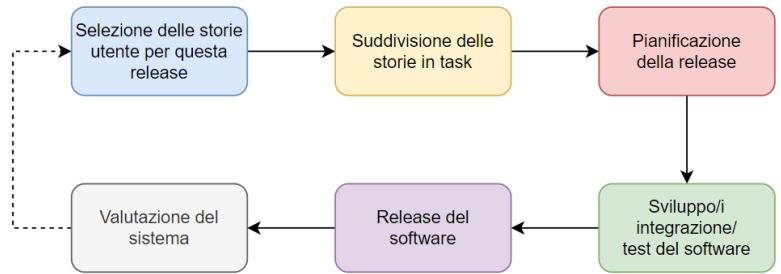
Tra le varie metodologie agili abbiamo:

- **Feature-Driven Development (FDD);**
- **Adaptive Software Process;**
- **Crystal Light Methodologies;**
- **Dynamic Systems Development Method (DSDM);**
- **Lean Development;**

Oltre alle precedenti metodologie a noi interessano in particolare:

- **Extreme Programming (XP)**, in cui:

1. Lo sviluppo incrementale è supportato attraverso piccole e frequenti release del sistema. I requisiti si basano su semplici scenari, che sono utilizzati come base per decidere quale funzionalità deve essere incluso in un incremento del sistema;
2. Il coinvolgimento dell'utente è supportato attraverso l'impegno costante del cliente nel team di sviluppo; anche i rappresentanti del cliente prendono parte allo sviluppo e hanno la responsabilità di definire i test di accettabilità del sistema;
3. Le persone, non il processo, sono supportate dalla programmazione in coppia, dal possesso collettivo del codice del sistema, e da un processo di sviluppo sostenibile che non richiede periodi di lavoro eccessivamente lunghi;
4. Le modifiche sono supportate da regolari release del sistema, dallo sviluppo preceduto da test, dal refactoring per evitare la degenerazione del codice, e dall'integrazione continua di nuove funzionalità;
5. Il mantenimento della semplicità è supportato dal costante refactoring che migliora la qualità del codice e dall'uso di semplici progetti che non necessariamente prevedono future modifiche del sistema.



Sia il seguente schema:

Principio/pratica	Descrizione
Proprietà collettiva	Le coppie di sviluppatori lavorano su tutte le aree del sistema, in modo che non ci siano esperti isolati che sviluppano; ogni sviluppatore è responsabile di tutto il codice; chiunque può cambiare qualsiasi cosa.
Integrazione continua	Appena un task è concluso, viene integrato nel sistema. Dopo ogni integrazione tutti i test sulle unità del sistema devono essere superati.
Pianificazione incrementale	I requisiti sono registrati su story card e le storie da includere in una release sono determinate dal tempo disponibile e dalla loro priorità relativa. Gli sviluppatori suddividono queste storie in "task" di sviluppo.
Cliente on-site	Un rappresentante dell'utente finale del sistema (il cliente) dovrebbe sempre essere disponibile per i membri del team XP. In un processo di programmazione estrema, il cliente è un membro del team di sviluppo e ha la responsabilità di consegnare i requisiti del sistema al team per l'implementazione.
Programmazione a coppie	Gli sviluppatori lavorano a coppie, verificando reciprocamente il loro lavoro e fornendo il supporto per fare sempre un buon lavoro.
Refactoring	Tutti gli sviluppatori effettuano in continuazione il refactoring del codice appena si trovano miglioramenti al codice stesso. Questo rende il codice semplice e mantenibile.
Progettazione semplice	Deve essere eseguita la progettazione sufficiente a soddisfare i requisiti correnti, niente di più.
Piccole release	Inizialmente deve essere sviluppato il minimo numero di funzionalità utili. Le release del sistema sono frequenti e aggiungono, in modo incrementale, nuove funzionalità alla release iniziale.
Ritmo sostenibile	Non sono considerati accettabili grandi ritardi, poiché spesso l'effetto finale è la riduzione della qualità del codice e della produttività a medio termine.
Sviluppo con test iniziali	Viene utilizzato un ambiente automatico di test delle unità per provare una nuova parte di funzionalità, prima che questa sia implementata.

Un problema fondamentale dello sviluppo incrementale è che le modifiche locali tendono a deteriorare la struttura del software. Di conseguenza, le modifiche future diventano sempre più difficili da implementare. In essenza, lo sviluppo procede in modo da trovare dei modi per aggirare i problemi, con il risultato che il codice spesso è duplicato, parti del software vengono riutilizzate in modi inappropriate, e la struttura complessiva del software si deteriora ogni volta che viene aggiunto nuovo codice.

Il **refactoring** migliora la struttura e la leggibilità del software, evitando così il deterioramento strutturale che si verifica naturalmente quando si modifica il software.

Esempi di *refactoring* sono la riorganizzazione della gerarchia delle classi per eliminare il codice duplicato, il riordino e la ridenominazione degli attributi e dei metodi, e la sostituzione di sezioni di codice simili, con chiamate ai metodi definite in una libreria di programma. Gli ambienti di sviluppo dei programmi di solito includono strumenti di *refactoring*. Essi semplificano la ricerca delle dipendenze tra sezioni di codice e la modifica del codice globale.

In teoria, quando il *refactoring* è parte del processo di sviluppo, il software dovrebbe essere sempre facile da capire e modificare quando vengono proposti nuovi requisiti. In pratica, questo non è sempre possibile. A volte, le pressioni che gravano sul processo di sviluppo fanno sì che il *refactoring* sia rinviato, perché il tempo viene dedicato all'implementazione di nuove funzionalità. Alcune caratteristiche e modifiche nuove non possono essere realizzate immediatamente dal *refactoring* a livello codice, in quanto richiedono che sia modificata l'architettura del sistema.

La programmazione estrema ha sviluppato un nuovo approccio al test dei programmi per superare alcune difficoltà tipiche dei test senza specifica. I test sono automatizzati e sono centrali nel processo di sviluppo, e lo sviluppo non può procedere finché tutti i test non sono stati superati con successo. Gli elementi caratteristici del test nella programmazione estrema sono:

1. Sviluppo con test iniziali;
2. Sviluppo di test incrementale dagli scenari;
3. Coinvolgimento dell'utente nello sviluppo e nella convalida dei test;
4. Uso di strutture automatiche per i test.

La scrittura dei test implica la definizione di un'interfaccia e di una specifica comportamentale per le funzionalità da sviluppare. I problemi con i requisiti e le incomprensioni dell'interfaccia si riducono. Lo sviluppo con test iniziali richiede che ci sia una relazione chiara tra i requisiti del sistema e il codice che li implementa. Nella programmazione estrema si può sempre vedere questa relazione perché le **story card** che rappresentano i requisiti sono suddivise in task, e i task sono le unità principali dell'implementazione.

Nello sviluppo con test iniziali, chi implementa i task deve capire bene le specifiche, in modo che possa scrivere test per il sistema. Ciò significa che le ambiguità e le omissioni nelle specifiche devono essere chiarite prima che inizi l'implementazione, evitando così il problema del **ritardo dei test (test-lag)**. Questo può accadere quando lo sviluppatore del sistema lavora più velocemente di chi esegue i test. L'implementazione è sempre più veloce del processo di test, e c'è la tendenza a tralasciare i test per poter rispettare la tempistica dello sviluppo.

Es.

STORY CARD

Prescrizione dei farmaci
Kate è un dottore che desidera prescrivere un farmaco a un paziente che frequenta una clinica. La cartella clinica del paziente è già visualizzata sul suo computer, quindi fa clic sul campo del farmaco e può selezionare "farmaco corrente", "nuovo farmaco" o "formulario".
Se sceglie "farmaco corrente", il sistema le chiede di controllare la dose. Se Kate vuole cambiare la dose, digita la nuova dose e conferma la prescrizione.
Se sceglie "nuovo farmaco", il sistema suppone che Kate conosca quale farmaco prescrivere. Kate digita le prime lettere del nome del farmaco. Il sistema visualizza una lista di possibili farmaci che iniziano con quelle lettere. Kate seleziona il farmaco richiesto e il sistema risponde chiedendole di verificare se il farmaco selezionato è corretto. Lei digita la dose e conferma la prescrizione.
Se sceglie "formulario", il sistema visualizza una finestra di ricerca per il formulario approvato. Kate può cercare il farmaco richiesto e poi lo seleziona. Il sistema le chiede di verificare se il farmaco scelto è corretto. Lei digita la dose e conferma la prescrizione.
Il sistema controlla sempre che la dose sia entro i limiti consentiti; in caso contrario, chiede a Kate di cambiare la dose.
Dopo che Kate ha confermato la prescrizione, dovrà controllarla e poi fare clic su "OK" o su "Cambia". Se fa clic su "OK", la prescrizione viene memorizzata nel database di controllo. Se fa clic su "Cambia", il sistema riavvia il processo di prescrizione dei farmaci.

TASKS DELLE CARTE

Task 1: cambia la dose del farmaco prescritto
Task 2: scelta dal formulario
Task 3: controllo della dose
Il controllo della dose è una precauzione per verificare che il dottore non abbia prescritto una dose pericolosamente piccola o grande.
Utilizzando il codice del formulario come nome del farmaco, ricerca nel formulario le dosi minima e massima consigliate.
Confronta la dose prescritta con la minima e la massima. Se la dose è fuori dal range ammesso, visualizza un messaggio di errore per segnalare che la dose prescritta è troppo piccola o troppo grande. Se è all'interno del range, abilita il pulsante "Conferma".

In conclusione, i rischi del processo XP sono:

- Il codice non viene testato completamente;
- Il team produce pochi test utili per il cliente;
- Il cliente non aiuta a testare il sistema;
- I test "non funzionano" prima dell'integrazione;
- I test sono troppo lenti – Le storie sono troppo complicate;
- Il sistema è troppo difettoso;
- Il cliente vuole il documento "Specifiche dei requisiti";
- Il manager vuole la documentazione di sistema;
- Il team è sovraccarico di compiti;
- Il team deve affrontare scelte tecniche rischiose;
- Alcuni membri (cowboy coders) ignorano il processo del team.

- **Scrum**, si divide in poche semplici fasi iterative:

1. **Product Backlog**, una lista di tutto ciò che si vorrebbe fare. Ogni Item di Backlog deve aggiungere un valore all'utente finale, ciò implica che il backlog deve essere ordinato e prioritizzato.

Il Backlog è un documento vivente, il processo di revisione e modifica giornaliera del Backlog da parte del Product Owner (PO) si chiama **Backlog Grooming**.

I dettagli di ogni Item dipendono tipicamente dalla loro posizione sul Backlog: in alto tanti dettagli, in basso pochi dettagli.

2. **Sprint – Sprint Planning**, lo Sprint inizia con un meeting detto **Sprint Planning**, diretto dallo scrum master a cui partecipano PO e Team di prodotto.

Team e PO Insieme selezionano item ad alta priorità dal Backlog che il team pensa di potersi impegnare a rilasciare all'interno dello sprint:

- Il PO lavora con il team, Insieme!
- È il team stesso a scegliere cosa portare in sviluppo e si impegna a rispettare la deadline: non c'è un blocco di feature che viene "calato dall'alto".

Ogni Item viene spacchettato alla sua unità più semplice e viene pesato in termini di "tempo di sviluppo necessario".

3. **Pull**, gli item vengono "tirati" dal backlog e non "pushati" da qualcun altro. Per l'intera durata dello sprint il team si focalizza soltanto sull'obiettivo di sprint e sugli item nello sprint backlog.

La bravura del *Product Owner* e dello *Scrum Master* sta proprio nel lasciar libero il team di fare focus sullo sprint e di tararsi sull'obiettivo di sprint.

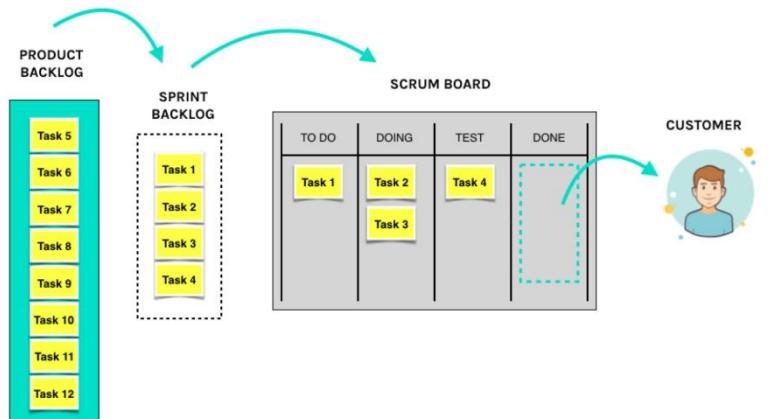
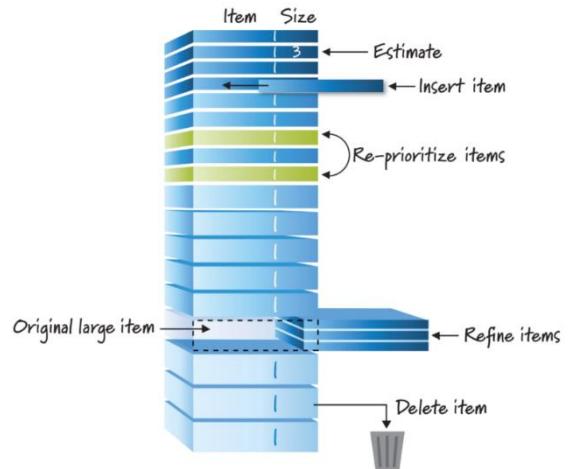
Per tracciare i progressi si usa una board chiamata **Scrum Board** (oppure Agile Board o Kanban Board), questa ha quattro colonne: *To Do*, *Doing* o *Progress*, *Test* o *QA* e *Done*.

Il movimento ideale degli Item sulla scrumboard da sinistra verso destra.

4. **Daily Scrum**, ogni giorno c'è un meeting chiamato **Scrum Meeting** (oppure Daily Scrum o Scrum) in cui l'obiettivo è quello di muoversi avanti verso la parte di *done*, della board.

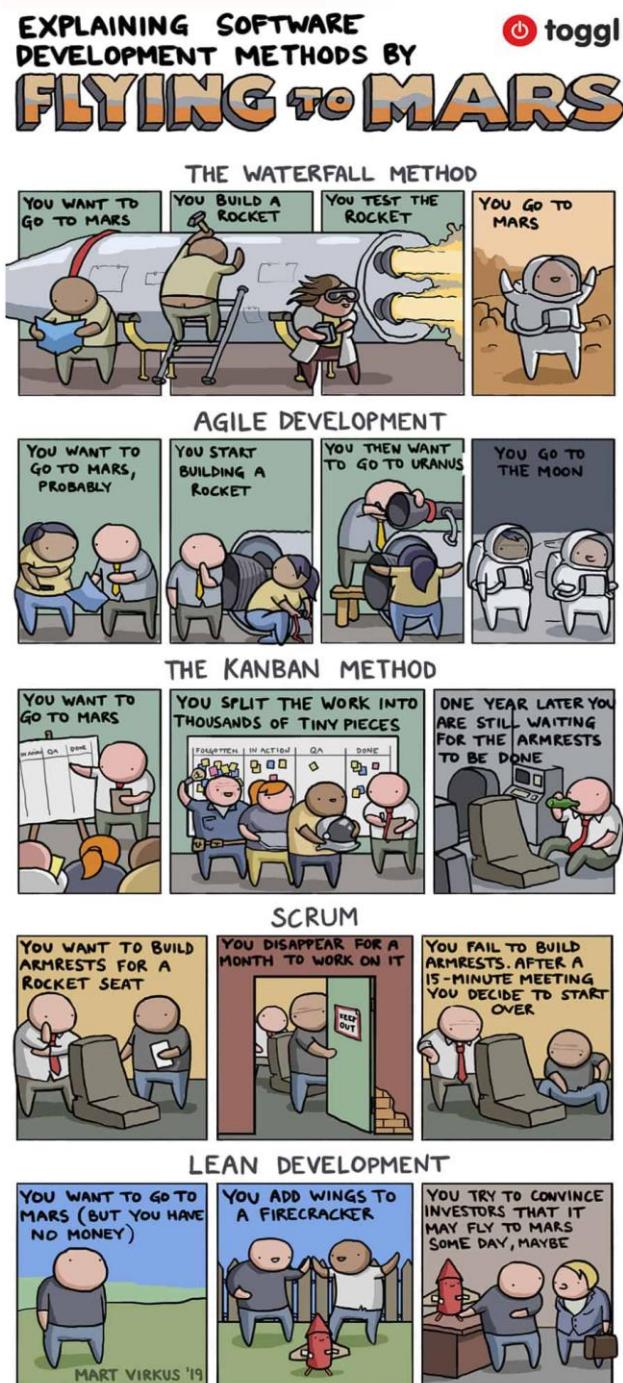
Durante lo Scrum Meeting viene discusso il progress, vengono spostati i Task e vengono identificati gli impedimenti. La durata massima è di 15 minuti (Di solito si sta in piedi) ogni persona nel meeting risponde a tre domande:

- Cosa ho fatto ieri?
- Quali sono i task di oggi?
- Che impedimenti non mi permettono di andare avanti?



In conclusione, ai vari metodi ci domandiamo quale ci conviene di più, agile o pianificato?

Fattore	Pro-Agile	Pro-Pianificato
Dimensione	Adatto a team che lavorano su prodotti SW "piccoli". L'uso di conoscenza tacita limita la scalabilità	Adatto a grandi sistemi e team. Costo da scalare verso i prodotti SW "piccoli".
Criticità	Utile per applicazioni con requisiti instabili (Es. Web).	Utile per gestire sistemi critici e con requisiti stabili.
Dinamismo	Refactoring.	Pianificazione dettagliata.
Personale	Servono esperti dei metodi agili; con Scrum devono essere "certificati".	Servono esperti durante la pianificazione.
Cultura	Piace a chi preferisce la libertà di fare.	Piace a chi preferisce ruoli e procedure ben definiti.



UNIFIED MODEL LANGUAGE

Unified Model Language (UML) è una famiglia di notazioni grafiche, serve a supportare la descrizione e il progetto di sistemi SW, in particolare è adatto a sistemi realizzati con il paradigma **orientato agli oggetti (OO)**.

UML è uno standard relativamente aperto, controllato dall'Object Management Group (OMG) che l'ha riconosciuto nel 1997 nella versione 1.0, nasce dalla fusione di alcuni linguaggi grafici di modellazione OO sviluppati tra gli anni '80 e '90.

Gli sviluppatori usano UML per documentare alcuni aspetti del sistema, gli sketch possono essere utilizzati sia in **fase di costruzione** di un sistema (**forward engineering**) che nel **reverse engineering** di un sistema esistente (nel reverse engineering, diagrammi abbozzati servono a comprendere il funzionamento di parti del sistema).

Lo scopo di questi diagrammi informali è supportare la comunicazione, la discussione delle idee e la valutazione delle alternative all'interno del team di sviluppo; non servono a descrivere dettagliatamente il codice da sviluppare, ma solo gli aspetti più importanti di cui si vuole discutere.

Ad oggi UML conta la versione 2 dove i diagrammi tipici sono:

Diagramma	Scopo	Origine
Casi d'uso	Interazioni tra utenti e sistema	UML 1
Classi	Classi, loro caratteristiche e relazioni	UML 1
Sequenza	Interazioni tra oggetti, con enfasi sulla sequenza	UML 1
Oggetti	Configurazione esemplificativa di istanze	UML 1*
Package	Struttura gerarchica a tempo di compilazione	UML 1*
Deployment	Distribuzione artefatti in diversi nodi	UML 1
Macchine a stati	Reazioni di un oggetto agli eventi	UML 1
Attività	Comportamento procedurale e parallelo	UML 1
Comunicazione	Interazione tra oggetti con enfasi sui collegamenti	UML 1+
Struttura composita	Scomposizione di una classe a runtime	UML 2
Componenti	Struttura dei componenti e loro connessioni	UML 1
Interazione generale	Fusione di un diagramma di sequenza ed uno di attività	UML 2
Temporizzazione	Interazione tra oggetti con enfasi sul tempo	UML 2

* presente in modo non ufficiale. + era chiamato "diagramma di collaborazione".

I diagrammi di cui noi faremo uso, per l'analisi dei requisiti, principalmente sono:

- I **diagrammi di casi d'uso**, descrivono le interazioni tra le entità esterne ed il sistema;
- I **diagrammi delle classi**, di livello concettuale, può essere utilizzato per definire un vocabolario per descrivere il dominio di interesse ed esprimere le relazioni tra i concetti di tale dominio;
- I **diagrammi di attività**, che possono essere impiegati per:
 - Illustrare il flusso di lavoro che interessa il sistema;
 - Chiarire il funzionamento dei casi d'uso più complessi;
- I **diagrammi di stato**, utili per descrivere entità che hanno comportamenti che possono cambiare in funzione del verificarsi di determinati eventi.

È importante che in questa fase i diagrammi siano semplici e comprensibili dagli utenti.

Nel processo di sviluppo invece utilizzeremo questi altri:

- I **diagrammi delle classi**, disegnati usando la prospettiva SW ed illustrano le classi e le loro relazioni (statiche);
- I **diagrammi di sequenza**, utilizzati per documentare gli scenari più comuni;
- I **diagrammi di package**, mostrano l'organizzazione del software su larga scala;
- I **diagrammi di stato**, utilizzati per descrivere le classi le cui istanze hanno un ciclo di vita complesso;
- I **diagrammi di deployment**, che illustrano la disposizione fisica del sistema.

Dopo aver implementato il sistema, UML può essere utilizzato per produrre la sua **documentazione**, i diagrammi sono utili per fornire una rappresentazione generale del sistema:

- Un *diagramma dei package* è un'ottima mappa logica del sistema, illustra come è suddiviso e le dipendenze tra le sue parti;
- Un *diagramma di deployment* mostra l'aspetto fisico del sistema ad un alto livello di astrazione;
- All'interno di ciascun *package* si può includere un *diagramma di classe* che evidenzi solo le caratteristiche più importanti;
- Un certo numero di *diagrammi di interazione* può accompagnare un *diagramma di classe* per documentare le interazioni più importanti;
- Ove utile, si possono includere *diagrammi di stato* per descrivere comportamenti complessi, frammenti di codice o diagrammi di attività per illustrare algoritmi complicati.

Uno **scenario** è una sequenza di passi che caratterizzano una particolare interazione tra utente e sistema.

Un **caso d'uso** è un insieme di scenari che hanno in comune lo scopo finale dell'utente.

(Uno scenario è una possibile “esecuzione” di un caso d'uso, si tratta di un'istanza del caso d'uso)

Gli utenti sono rappresentati per mezzo del concetto di attore, un **attore rappresenta un ruolo interpretato da un'entità esterna** (un utente umano, un sistema esterno) nei confronti del sistema: un singolo attore può partecipare a più casi d'uso; un singolo caso d'uso può coinvolgere più attori.

Diagramma di casi d'uso:

Non esiste un modo standard per esprimere un caso d'uso. Uno stile diffuso è il seguente:

- Si individua lo scenario principale di successo (main scenario);
- Il corpo del caso d'uso è costituito dalla sequenza numerata dei passi dello scenario principale;
- Si considerano scenari che vengono riportati come estensioni di quello principale: possono terminare con successo o fallire;
- Ogni caso d'uso ha un attore principale, ovvero quello che chiede il servizio al sistema;
- L'attore principale persegue lo scopo che il caso d'uso cerca di soddisfare;
- Gli altri attori, eventualmente coinvolti, sono detti attori secondari;
- Un passo di un caso d'uso corrisponde ad un'interazione tra un attore ed il sistema: è descritto da una frase semplice e non dovrebbe esprimere dettagli tecnici sulle azioni compiute.

Ci sono casi dove il semplice caso d'uso non basta ad esprimere lo scenario, in questo caso si fa uso delle:

- **Estensioni**, riportano la condizione che determina il verificarsi di interazioni diverse dallo scenario principale, la sua struttura è la seguente:
 - Si indica il passo in cui si può verificare la condizione;
 - Si aggiungono passi enumerati che dettagliano le interazioni dell'estensione;
 - Se necessario, si indica il punto di rientro nello scenario principale.
- **Inclusioni**, se un passo di un caso d'uso risulta complicato è possibile esprimere come un altro caso d'uso completo (il primo caso include il secondo). Per esprimere l'inclusione nella forma testuale non c'è una notazione standard, di solito si sottolinea il nome del caso d'uso incluso (come un collegamento ipertestuale).

Es.

Acquisto di un prodotto:

1. Il cliente naviga nel catalogo e seleziona gli articoli da acquistare.
2. Il cliente va alla cassa (effettua il check out).
3. Il cliente fornisce le informazioni relative alla spedizione.
4. Il sistema presenta un prospetto con il conto totale, comprese le spese di spedizione.
5. Il cliente riempie un modulo con i dati della sua carta di credito.
6. Il sistema autorizza l'acquisto.
7. Il sistema conferma immediatamente la vendita.
8. Il sistema invia al cliente una e-mail di conferma.

Estensioni:

3a. Il cliente è abituale:

1. Il sistema visualizza le preferenze memorizzate riguardanti la spedizione, il pagamento e la fattura ;
2. Il cliente può accettare il default o ridefinire le preferenze, in tal caso ritorna al passo 6.

6a. Il sistema non autorizza l'acquisto con la carta di credito:

1. Il cliente può inserire nuovamente le informazioni e riprovare oppure cancellare l'acquisto.

Un diagramma di caso d'uso è una sorta di sommario grafico, illustra i confini del sistema e le sue interazioni con il mondo esterno, raffigura gli attori, i casi d'uso e le loro relazioni:

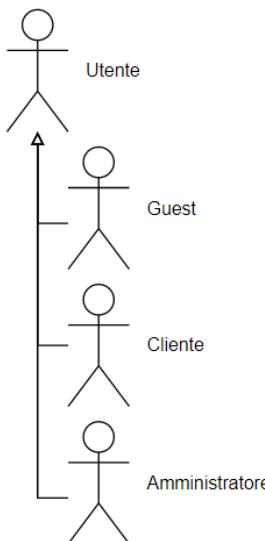
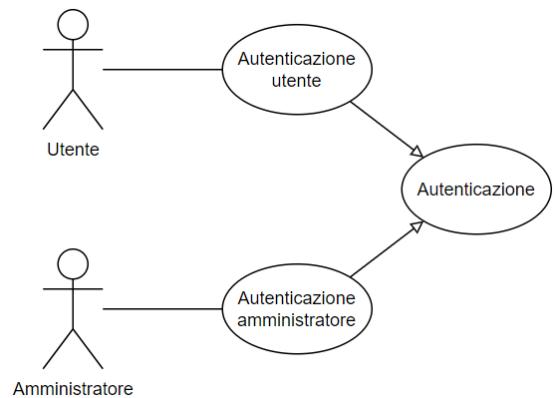


- Gli attori sono rappresentati come degli omini stilizzati;
- I casi d'uso sono disegnati come delle ellissi ciascuna riportante al proprio interno il nome del rispettivo caso d'uso;
- Gli attori sono collegati da una linea continua ai casi d'uso cui partecipano;
- La relazione di inclusione è rappresentata da una linea tratteggiata che termina con una freccia (relazione di dipendenza) etichettata con la parola chiave <include>.

La relazione di generalizzazione è indicata in UML come una linea continua terminata da un triangolo vuoto.

La relazione di generalizzazione tra casi d'uso indica che il caso d'uso figlio, pur essendo simile al padre, ne specializza alcuni aspetti.

Il figlio eredita il comportamento del padre e lo può estendere modificando e/o aggiungendo passi elementari



La generalizzazione tra attori indica che il ruolo corrispondente all'attore figlio è più specifico di quello dell'attore padre.

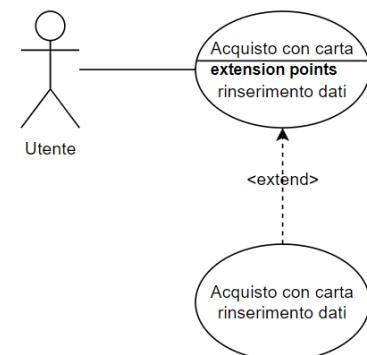
Il ruolo corrispondente all'attore figlio è compatibile con quello corrispondente al padre (chi ricopre il ruolo del figlio può ricoprire anche il ruolo del padre).

Il viceversa non vale.

La relazione di estensione tra casi d'uso si indica con una relazione di dipendenza etichettata dalla parola chiave <extend>.

La direzione della freccia va dal caso d'uso che rappresenta l'estensione verso quello principale.

I punti di estensione sono indicati all'interno del caso d'uso principale



I diagrammi dei casi d'uso servono per identificare le funzionalità svolte dal sistema software. Tali funzionalità dovranno poi essere assegnate agli oggetti/classi (**assegnazione di responsabilità**)

Tipicamente si parte da macro-funzionalità che poi vengono via via raffinate in funzionalità più semplici sino ad arrivare a operazioni che non ammettono decomposizione. Eventuali ambiguità sull'ordine di esecuzione di casi d'uso inclusi in un altro, possono essere chiarite da un *diagramma di attività*.

I casi d'uso di UML sono stati aspramente criticati da alcuni autori (es. B. Meyer) perché possono drenare l'attenzione del progettista più sulla scomposizione funzionale che non sul progetto ad oggetti e classi

L'analisi dei casi d'uso può: aiutare a identificare gli oggetti (aspetto fondamentale), definire i casi di test da effettuare in seguito sui moduli implementati e caratterizzare la dinamica di interazione col sistema, ulteriormente descrivibile mediante diagrammi di interazione e collaborazione

Diagrammi di sequenza:

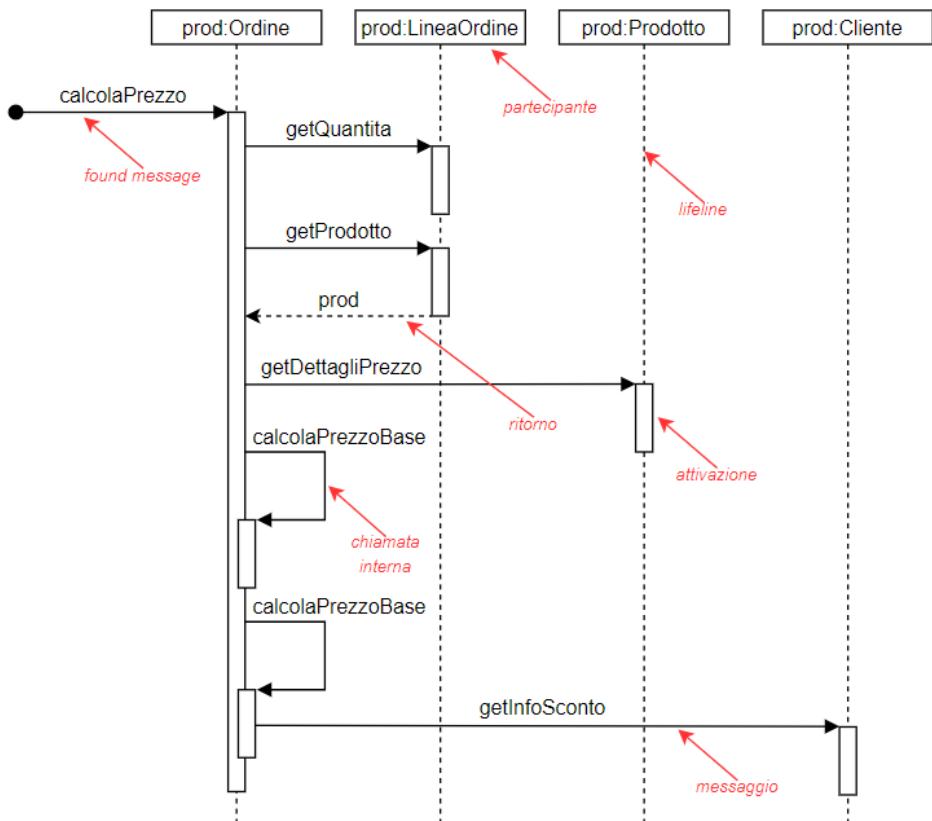
I **diagrammi di sequenza** appartengono alla categoria dei diagrammi di interazione i quali descrivono la collaborazione di un gruppo di oggetti che devono implementare un certo comportamento, documenta tipicamente il comportamento di un singolo scenario.

Il diagramma include un certo numero di oggetti e i messaggi scambiati tra di essi durante l'esecuzione del caso d'uso.

Es.

Si consideri il seguente scenario: calcolo del prezzo di un ordine.

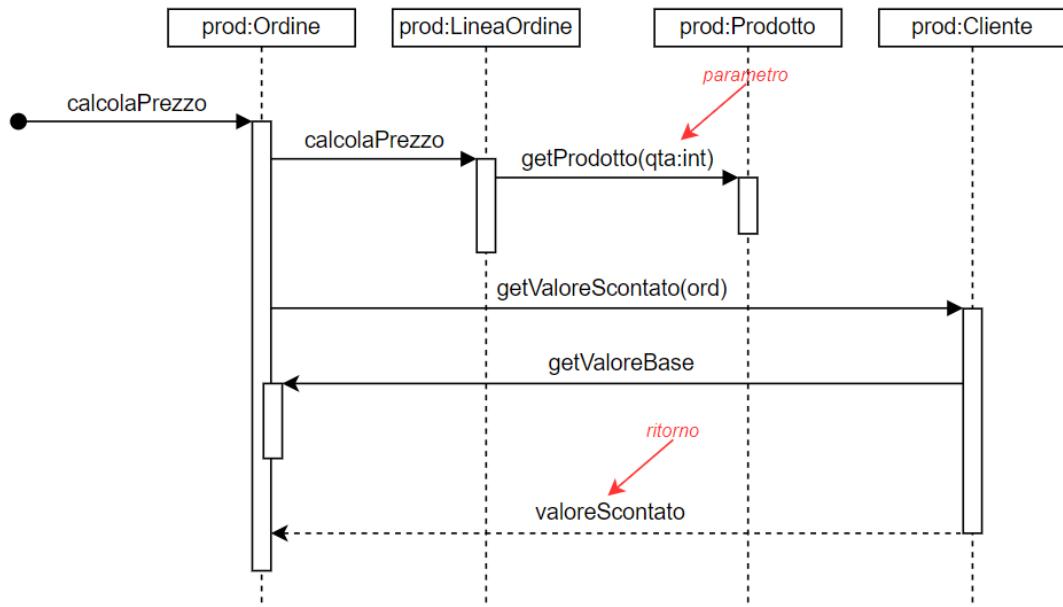
L'ordine deve calcolare il prezzo di ciascuna linea, per farlo occorre ottenere la quantità relativa al prodotto ed il suo prezzo unitario, una volta calcolato il prezzo totale, si calcola lo sconto che dipende dallo specifico cliente.



Elementi di un diagramma di sequenza:

- Ciascun partecipante è rappresentato da un box che contiene il nome (nella notazione degli Object Diagram ma non sottolineato) e da una linea tratteggiata verticale detta “linea di vita” (**lifeline**);
- L'ordinamento dei messaggi è determinato scorrendo il diagramma dall'alto verso il basso;
- UML 1 prescriveva che i partecipanti fossero oggetti;
- Ogni linea di vita ha una **barra di attivazione** che indica quando il partecipante è attivo nell'interazione;
- Le frecce indicano i **messaggi (invocazioni)** e sono etichettate con il nome del messaggio (**nome del metodo**);
- Le frecce di ritorno sono opzionali;
- Quando presenti, possono riportare il valore risultante dall'invocazione;
- Nell'esempio, il primo messaggio non scaturisce da un partecipante ed è detto **found message**;
- Quando un partecipante invia un messaggio a sé stesso, di solito si sovrappone una barra di attivazione a quella già presente;

Il seguente diagramma illustra una differente modalità di interazione tra i partecipanti.



Nel caso precedente, un partecipante svolge tutta l'interazione, gli altri forniscono solo i dati (controllo centralizzato).

In questo caso, i compiti sono ripartiti tra i partecipanti (controllo distribuito).

Controllo centralizzato e controllo distribuito:

- Ognuno dei due stili ha i suoi punti di forza e di debolezza;
- I progettisti che non hanno ben assimilato il paradigma Object-Oriented spesso adottano quello centralizzato;
- I puristi degli oggetti preferiscono quello distribuito;
- Per limitare gli effetti dei cambiamenti è opportuno che dati e logica che li gestisce siano concentrati nello stesso posto;
- Distribuendo il controllo si può utilmente sfruttare il polimorfismo: se gli algoritmi che determinano il prezzo sono diversi in funzione della tipologia del prodotto si possono utilizzare le sottoclassi per gestire le variazioni;
- Lo stile Object-Oriented prevede l'uso di molti piccoli oggetti, ognuno dotato di diversi metodi che possono essere ridefiniti e consentono di inserire cambiamenti;

Diagramma delle classi:

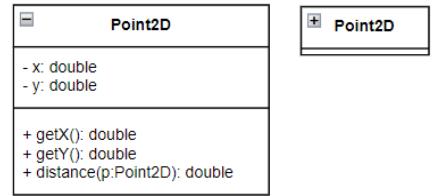
Il diagramma delle classi è il diagramma UML più ampiamente utilizzato, si presta a rappresentare il maggior numero di concetti.

Describe il **tipo degli oggetti** che fanno parte del sistema e le varie tipologie di **relazioni statiche** tra di essi.

Mostra le **proprietà** e le **operazioni (metodi)** delle classi ed i vincoli tra gli oggetti istanze delle classi.

UML usa il termine **caratteristica (feature)** per indicare sia le proprietà che le operazioni di una classe.

Le **classi** sono graficamente rappresentate come dei rettangoli (box) contenenti all'interno (almeno) il nome della classe, i box sono solitamente divisi in tre compartimenti: nome della classe, attributi, operazioni.



Le **proprietà** rappresentano le caratteristiche strutturali di una classe, in prima approssimazione si può affermare che le proprietà corrispondono ai campi di una classe.

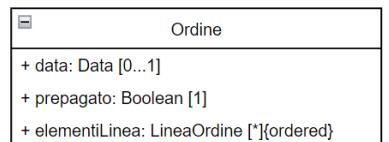
Sono rappresentate mediante due possibili notazioni:

- **Attributi**, un attributo descrive una proprietà come una stringa all'interno del relativo compartmento secondo

il seguente formato: visibilità nome: tipo molteplicità = default {stringa-di-proprietà}

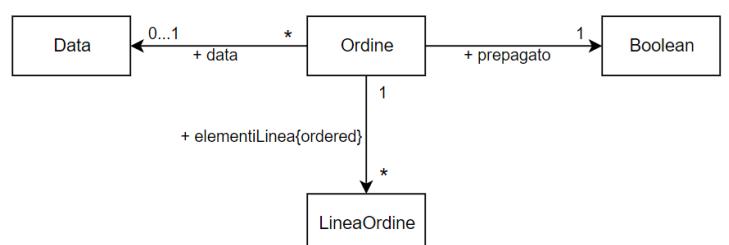
Ad esempio: titolo: String = "Hello world" {read-only}

- La **visibilità** specifica se l'attributo è pubblico (+), privato (-), protetto (#) o ha visibilità di package (~);
 - Il **nome** di un attributo corrisponde al nome di un campo;
 - Il **tipo** è un vincolo sugli oggetti che possono corrispondere all'attributo (il tipo del campo nel linguaggio di programmazione);
 - Il **default** è il valore assunto dall'attributo se non diversamente specificato;
 - La {stringa-di-proprietà} indica **caratteristiche aggiuntive** di un attributo (ad esempio vincoli).
- Nell'esempio {read-only} indica un attributo il cui valore non è modificabile dai client;
- La **molteplicità** esprime un vincolo sul numero di oggetti che possono essere collegati all'attributo.



- **Associazioni**, un'associazione che modella una proprietà è raffigurata come una linea continua che collega due classi, orientata dalla classe sorgente a quella destinazione.

Il *nome della proprietà* e la *molteplicità* sono indicati vicino all'estremità dell'associazione relativa alla destinazione.



La classe destinazione corrisponde al tipo della proprietà, di solito si utilizzano gli attributi per le cose "piccole" (oggetti valore).

Le associazioni sono usate per le proprietà le cui classi sono più "significative"

La **molteplicità** di una proprietà indica quanti oggetti possono entrare a farvi parte. Le più comuni sono:

- **1**, un ordine deve essere fatto da un solo cliente;
- **0...1**, un'azienda cliente può avere un rappresentante o meno;
- *****, da zero a molti: un cliente può anche non fare un ordine, ma non c'è limite superiore;
- Si possono specificare degli intervalli (ad es. **2...4**) fornendo esplicitamente i limiti inferiore e superiore;

Se i due estremi coincidono si usa un solo numero, ad esempio per abbreviare **1...1**, si usa solo **1**.

Si può in particolare abbreviare **0...***, con solo *****.

Terminologia attributi:

- **Opzionale**: implica 0 come limite inferiore;
- **Obbligatorio**: implica un limite inferiore di 1 (almeno uno);
- **Ad un sol valore**: implica un limite superiore di 1 (al più uno);
- **A più valori**: implica un limite superiore maggiore di 1 (di solito *****);

Di default gli elementi coinvolti in una molteplicità a più valori formano un set: non è definito un ordinamento tra di essi (è sottointesa la proprietà **{unordered}**), per specificare il vincolo che gli elementi devono essere ordinati si aggiunge l'indicazione **{ordered}**.

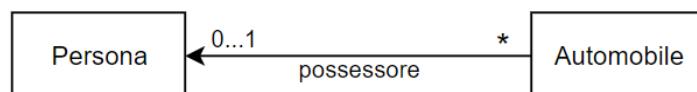
UML 1 consentiva l'uso di molteplicità discontinue come **2..4**, la molteplicità di default di un attributo è **[1]**.

È bene specificare che non esiste un unico metodo interpretativo delle proprietà nella programmazione, ad esempio:

- Nei linguaggi senza proprietà i campi sono resi visibili all'esterno per mezzo di metodi accessori (**get**) e/o mutatori (**set**);
- Se un attributo ha più valori i dati corrispondenti formano una collezione, ed hanno proprietà diverse da quelle a singolo valore;

Un'**associazione bidirezionale** è costituita da una coppia di proprietà collegate.

Nell'esempio la classe **Automobile** ha proprietà **proprietario:Persona**, mentre **Persona** ha proprietà **automobili:Automobile** **[*]**.



Il doppio collegamento indica che se si segue il valore di una proprietà e poi il valore di quella collegata si deve tornare ad un insieme che contiene l'elemento di partenza.

Implementare le associazioni bidirezionali è difficoltoso perché occorre assicurarsi che le proprietà coinvolte siano sincronizzate, la cosa più importante è fare in modo che l'associazione sia controllata da una sola delle due parti (possibilmente quella a valore singolo).

La classe subordinata deve violare temporaneamente l'incapsulamento dei suoi dati per farli arrivare a quella che controlla l'associazione.

Le **operazioni** sono le azioni che la classe esegue (di solito corrispondono ai metodi della classe), la sintassi UML completa è la seguente:

```
visibilità nome (lista-parametri): tipo-di-ritorno {stringa-di-proprietà}
```

- La **visibilità** segue le stesse regole sintattiche delle proprietà;
- Il **nome** è una stringa che indica il nome dell'operazione;
- **lista-parametri** è la lista ordinata dei parametri dell'operazione separati da virgolette;
- **tipo-di-ritorno** specifica il tipo del valore restituito dall'operazione, se esiste;
- **stringa-di-proprietà** indica caratteristiche aggiuntive dell'operazione (ad esempio vincoli).

Gli **elementi della lista dei parametri** seguono una sintassi simile a quella degli attributi:

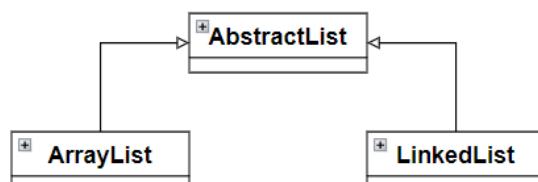
```
direzione nome: tipo = default
```

- **nome**, **tipo** e **default** sono gli stessi degli attributi;
- La **direzione** indica se il parametro è di input (**in**), output (**out**) o entrambi (**inout**).
Se omessa la direzione predefinita è **in**.

Spesso è utile distinguere tra le operazioni che cambiano lo stato di un sistema e quelle che non lo fanno
UML definisce **query** quelle che ottengono un valore senza modificare lo stato; tali operazioni sono etichettate con la stringa di proprietà **{query}**.

UML distingue tra operazione e metodo: la prima corrisponde alla dichiarazione di una procedura, il secondo al corpo della prima. Spesso i due termini sono utilizzati in modo intercambiabile.

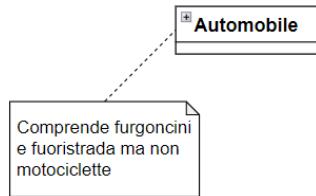
La relazione di **generalizzazione** può essere interpretata in vari modi



Dal punto di vista concettuale esprime la relazione che esiste tra concetti più generici e concetti più specifici, dal punto di vista software l'interpretazione più comune è quella che implica il meccanismo dell'ereditarietà.

il principio fondamentale da rispettare quando si usa l'ereditarietà è il principio di **sostituibilità** (Liskov): *ovunque ci si attende un'istanza dell'entità più generica deve essere possibile utilizzare, senza alterare il corretto funzionamento del sistema, un'istanza di quella più specifica.*

Le **note** sono commenti aggiuntivi che possono apparire in qualunque tipo di diagramma UML



Possono essere collegate con una linea tratteggiata agli elementi cui fanno riferimento o essere disegnate isolate

Per inserire un commento interno di un elemento del diagramma si scrive il commento dopo due trattini (--) .

Tra due elementi di un diagramma esiste una relazione di **dipendenza** se una modifica alla definizione di uno (fornitore o supplier) può comportare un cambiamento all'altro (il client).

Nel caso delle classi la dipendenza può essere dovuta a varie cause: una classe invoca i metodi di un'altra, la usa come tipo di un suo campo o come tipo di un parametro di un suo metodo.

UML consente di indicare dipendenze tra ogni tipo di elemento, la dipendenza è una relazione unidirezionale, si indica con una linea tratteggiata terminante con una freccia che ne specifica la direzione, UML prevede inoltre vari tipi di dipendenza, ognuna con una particolare semantica e varie parole chiave.

È importante sottolineare che la dipendenza non è una relazione transitiva.

Parola chiave	Significato
<<call>>	La sorgente invoca un'operazione della classe destinazione
<<create>>	La sorgente crea istanze della classe destinazione
<<derive>>	La sorgente è derivata dalla classe destinazione
<<instantiate>>	La sorgente è un'istanza della classe destinazione (in questo caso se la sorgente è una classe, ed è istanza della destinazione, quest'ultima deve essere una meta-classe)
<<permit>>	La classe destinazione permette alla sorgente di accedere ai suoi campi privati
<<realize>>	La sorgente è un'implementazione di una specifica o di un'interfaccia definita dalla destinazione
<<refine>>	Il raffinamento indica una relazione tra livelli semantici differenti; la classe sorgente ad esempio potrebbe essere una classe di progettazione, la destinazione una classe di analisi
<<substitute>>	La sorgente è sostituibile alla destinazione
<<trace>>	Usata per tenere traccia di cose come i requisiti o di come i cambiamenti a una parte di modello si collegano ad altre sue parti
<<use>>	a sorgente richiede la destinazione per la sua implementazione

CAPITOLO 2: PROGETTAZIONE DEL SOFTWARE

IL PONTE TRA L'ANALISI E LA PROGETTAZIONE

Visti i primi strumenti per dare una prima forma ai requisiti passiamo adesso alla progettazione di un software, principalmente si seguono tre regole:

- **Semplicità**
 - Nessun sistema nasce complesso;
 - Un design semplice spesso è elegante;
 - Il codice semplice è più facile da capire e correggere.
- **Affidabilità e robustezza**
 - *Primum non nocere*: non progettare sistemi che fanno danni;
 - Il sistema dovrebbe essere capace di gestire input inattesi;
 - Il sistema dovrebbe superare ogni errore senza crash.
- **Efficacia**
 - Bisogna risolvere il problema “giusto”;
 - Bisogna risolvere il problema in modo efficiente;
 - Prevede integrazione e compatibilità con altro software.

Dopo l'analisi, distinguiamo due fasi di progettazione:

- **Progettazione architettonica**, che si occupa principalmente di D
 - Decomporre il modello del sistema in moduli;
 - Determinare le relazioni tra i moduli.
- **Progettazione dettagliata**
 - Specifica delle interfacce dei moduli;
 - Descrizione funzionale dei moduli.

Il ponte tra l'analisi e la progettazione è l'**architettura logica**, la quale si compone delle seguenti astrazioni:

- **Strutture dati e algoritmo**;
- **Classe** (strutture dati e molti algoritmi);
- **Package/Moduli** (gruppi di classi correlate, magari interagenti via design pattern);
- **Moduli/Sottosistemi** (moduli interagenti contenenti ciascuno molte classi; solo le interfacce pubbliche interagiscono con altri moduli/sottosistemi);
- **Sistemi** (sistemi interagenti con altri sistemi - hardware, software ed esseri umani);

E si basa su:

- Suddivisione (partizione) della applicazione in una collezione di sottosistemi;
- Stratificazione (layering) delle componenti (moduli, classi e oggetti) di ciascun sottosistema su più livelli.

I vantaggi sono:

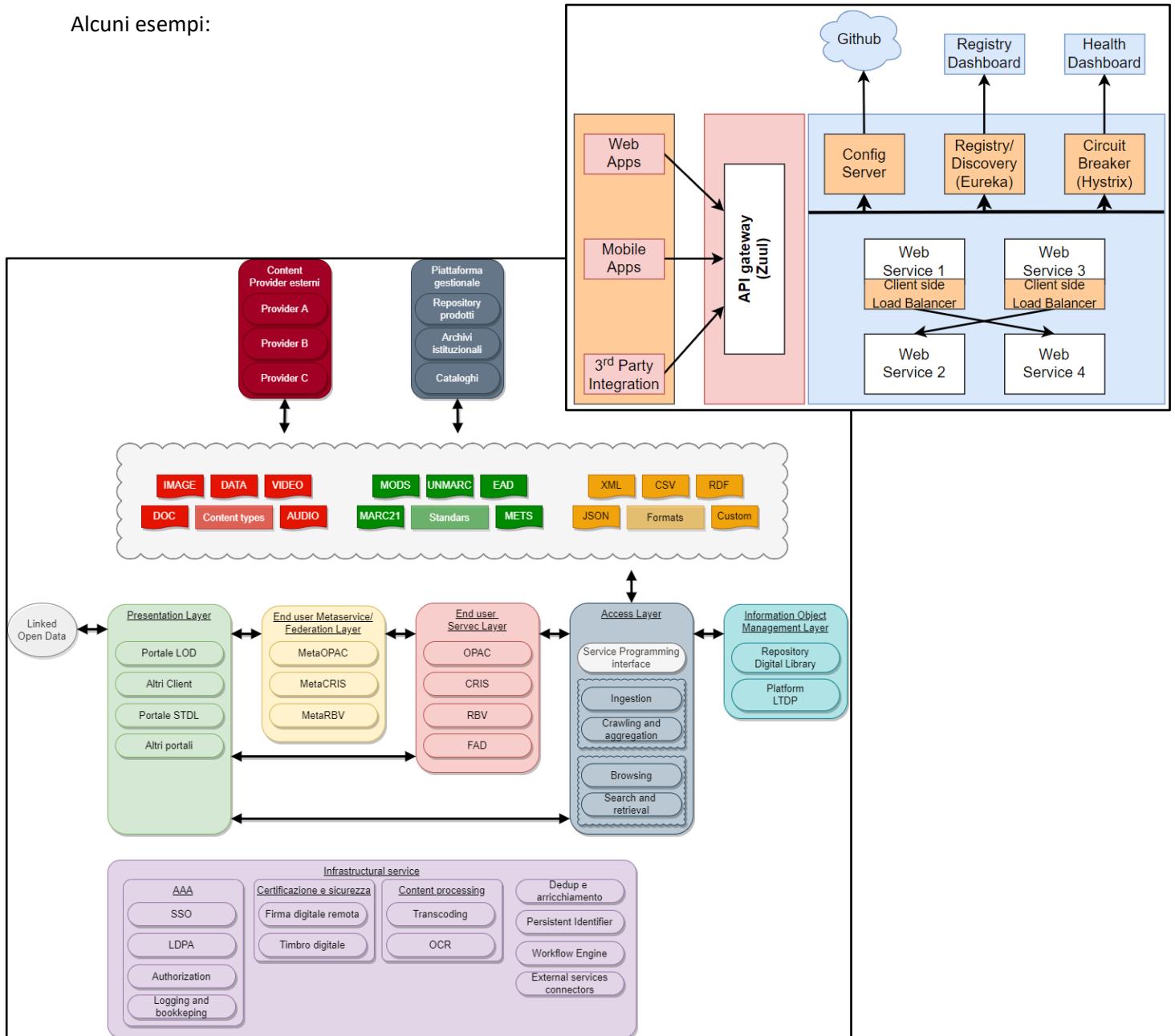
- Abbattere la complessità dei problemi lega: alla realizzazione (secondo il principio del *Divide et Impera*, risulta più semplice affrontare poche complessità per volta piuttosto che tutte insieme);
- Facilitare la suddivisione dei compiti tra le persone del gruppo di progetto o tra i gruppi di progetto;
- Predisporre la successione dei rilasci sulla base della realizzazione dei sottosistemi;
- Isolare scelte o decisioni non ancora definite;
- Differenziare i servizi applicativi con un forte grado di riusabilità.

Si hanno due modalità di sviluppo dell'architettura:

- **Bottom-up:** a partire dalle classi del sistema, precedentemente individuate, raggruppando in uno stesso sottosistema classi strettamente correlate dal punto di vista funzionale.
Vengono normalmente poste nello stesso sottosistema le classi che:
 - Appartengono allo stesso dominio;
 - Eseguono diverse operazioni l'una sull'altra;
 - Sono legate strutturalmente da relazioni di associazione, generalizzazione o specializzazione, aggregazione;
 - Dipendono da una stessa classe interfaccia o sono coinvolte in uno stesso caso d'uso.
- **Top-down:** prima vengono identificati i principali sottosistemi, di cui vengono delineano i servizi che dovranno fornire e solo successivamente, analizzando più in dettaglio questi sottosistemi, vengono scoperte e progettate le classi che li compongono.

La linea guida per identificare i sottosistemi sono i **casi d'uso**, secondo la relazione per cui un sottosistema implementa uno o, se sono semplici, più casi d'uso a livello utente.

Alcuni esempi:



MODELLI ARCHITETTURALI

Abbiamo visto cos'è e com'è composta un'architettura, tuttavia esistono dei modelli predefiniti, questi sono:

- **Architettura a Strati Multi-Tier**, organizza il sistema in vari strati, con funzionalità associate a ciascuno strato. Uno strato fornisce i servizi allo strato sopra di esso; quindi, gli strati di livello più basso rappresentano i servizi di base che probabilmente saranno utilizzati in tutto il sistema (ad esempio: un modello a strati di un sistema di apprendimento digitale che supporta l'apprendimento di tutti i soggetti nelle scuole).

Si usa per costruire nuove funzioni per un sistema esistente; quando lo sviluppo è distribuito fra più team, dove ogni team ha la responsabilità di sviluppare le funzioni di uno strato; quando c'è una richiesta di protezione su più livelli.

Vantaggi:

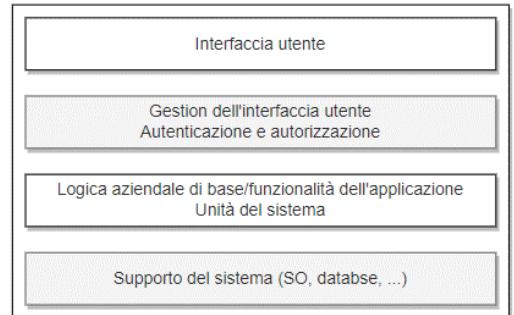
- Consente la sostituzione di interi strati, se l'interfaccia non viene modificata.
- Le funzioni ridondanti (es. l'autenticazione) possono essere fornite in ciascuno strato per aumentare la fidatezza del sistema.

Svantaggi:

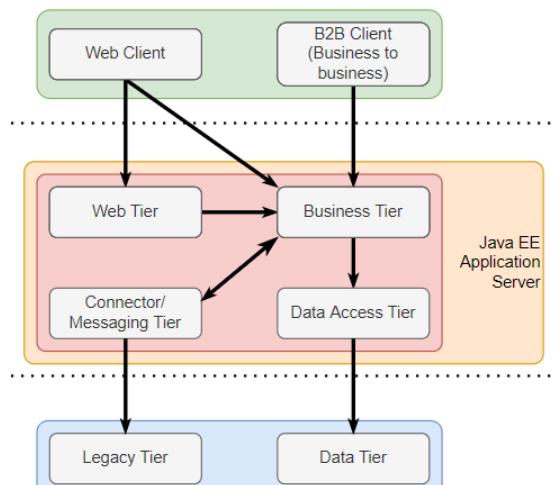
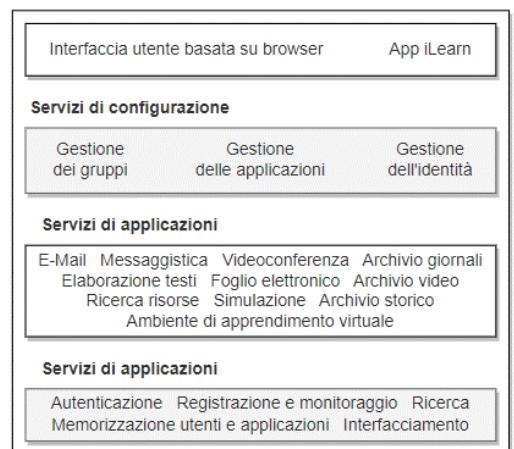
- Nella pratica spesso è difficile ottenere una netta separazione fra gli strati, e uno strato di alto livello potrebbe aver bisogno di interagire direttamente con strati di livelli inferiori, anziché con lo strato immediatamente sotto di esso.
- Le performance possono essere un problema, a causa dei vari livelli di interpretazione di una richiesta di servizio, essendo questa elaborata in ciascuno strato.

Gli strati che suddividiamo sono tre:

- **Livelli Logici**
 - Usati per separare le responsabilità e gestire le dipendenze;
 - A ogni livello logico è assegnata una responsabilità specifica;
 - Un livello logico superiore può usare i servizi di un livello logico inferiore, ma non viceversa.
- **Livelli Fisici**
 - I livelli fisici sono separati fisicamente, ovvero vengono eseguiti in computer diversi; Un livello fisico può chiamare direttamente un altro livello fisico o usare la messaggistica asincrona (coda di messaggi);
 - Ogni livello logico può essere ospitato nel proprio livello fisico, anche se questo non è obbligatorio.
Lo stesso livello fisico può ospitare più livelli logici;
 - La separazione fisica dei livelli fisici ne migliora la scalabilità e la resilienza, ma implica una maggiore latenza dovuta all'incremento delle comunicazioni di rete.
- **Multi-tier**, una tipica applicazione a tre livelli fisici include:
 - Un livello presentazione;
 - Un livello intermedio (facoltativo);
 - Un livello database.Le applicazioni più complesse possono contenere più di tre livelli fisici



Es.



- **Model View Controller (MVC)**, separa la presentazione e interazione dai dati del sistema.

Il sistema è strutturato in tre componenti logiche che interagiscono tra loro:

- Il **componente Model**, gestisce i dati del sistema e le operazioni associate;
- Il **componente View**, definisce e gestisce il modo in cui i dati sono presentati all'utente;
- Il **componente Controller**, gestisce l'interazione degli utenti (tasti premuti, clic del mouse ecc.) e passa queste interazioni ai componenti Model e View.

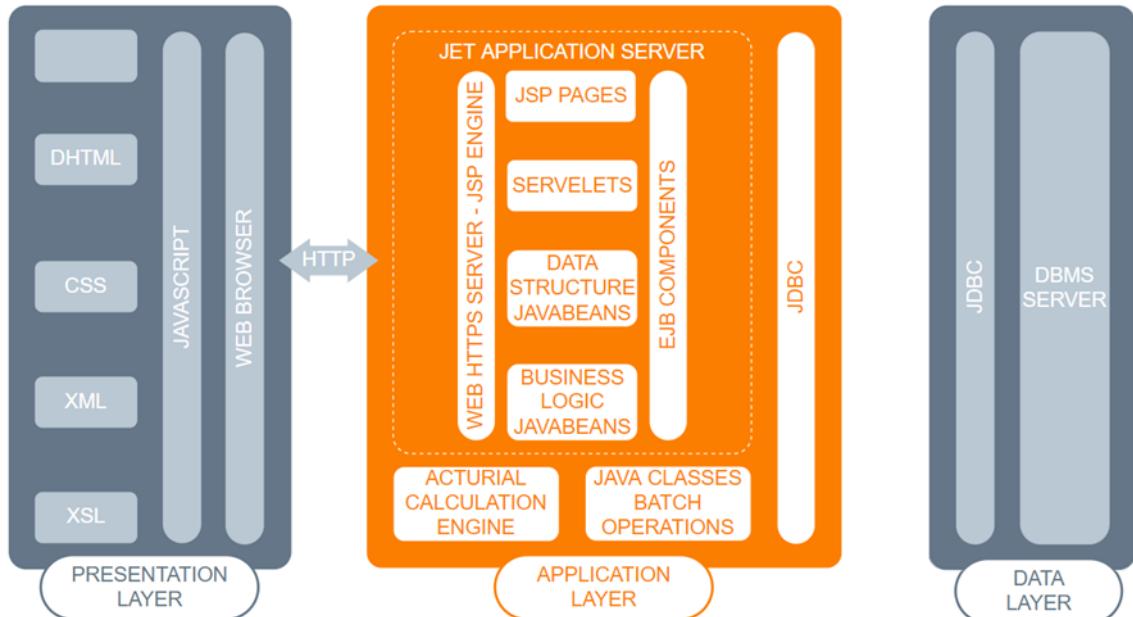
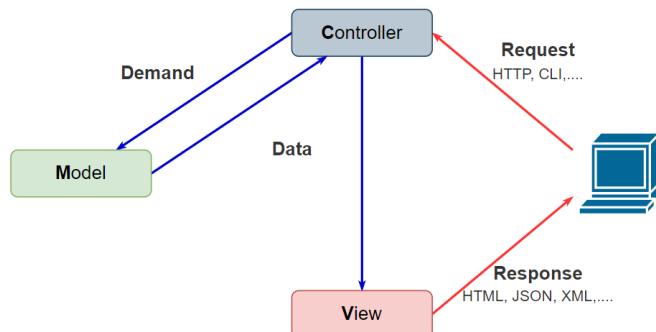
Il MVC si usa quando ci sono più modi di visualizzare e interagire con i dati; si usa anche quando non sono noti i requisiti futuri di interazione e presentazione dei dati.

Vantaggi:

- Consente ai dati di cambiare indipendentemente dalla loro rappresentazione e viceversa;
- Supporta la presentazione degli stessi dati in modi differenti, con le modifiche fatte in una rappresentazione mostrate in tutte le altre.

Svantaggi:

- Potrebbe richiedere altro codice o un codice complesso quando il modello dei dati e le interazioni sono semplici.



- **Client-Server**, il sistema è presentato come un insieme di servizi, ciascuno dei quali è fornito da un server separato. I client sono utenti di questi servizi e accedono ai server per utilizzare tali servizi.

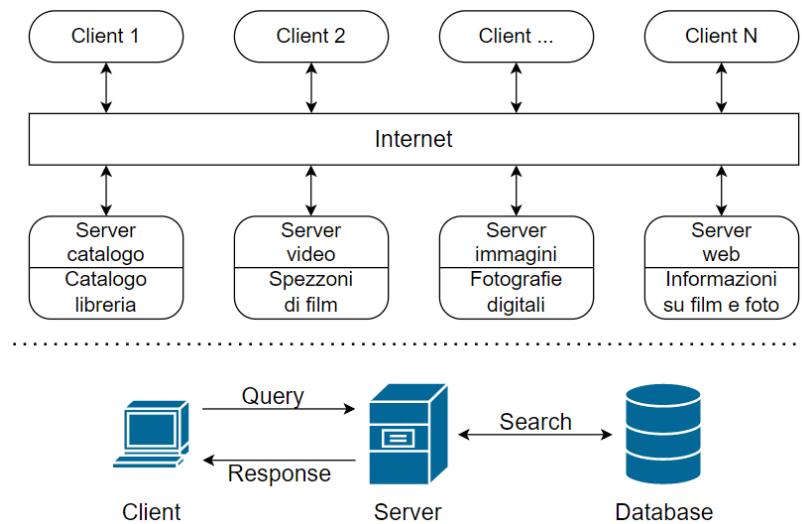
Si usa quando occorre accedere ai dati di un database da più postazioni. Poiché i server possono essere replicati, lo schema può essere utilizzato anche quando il carico su un sistema è variabile.

Vantaggi:

- Il principale vantaggio di questo modello è che i server possono essere distribuiti in una rete;
- Le funzionalità più comuni (per esempio, il servizio di stampa) possono essere a disposizione di tutti i client e non devono essere necessariamente implementate da tutti i servizi;

Svantaggi:

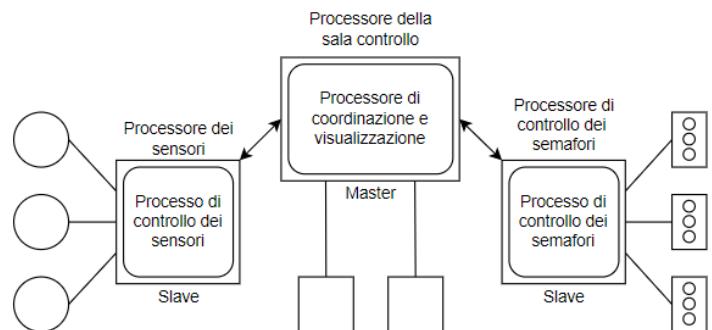
- Ciascun servizio è un punto comune di malfunzionamento, nel senso che è suscettibile di attacchi *denial-of-service* ed è soggetto a guasti del server;
- Le prestazioni possono essere imprevedibili in quanto dipendono dalla rete e anche dal sistema. Possono nascere problemi di gestione se i server sono di proprietà di più organizzazioni.



L'architettura Client-Server non è unica, essa ha cinque modelli e casi d'uso diversi:

- **Architettura Master-Slave**, utilizzata nei sistemi real-time in cui devono essere garantiti i tempi di risposta delle interazioni;

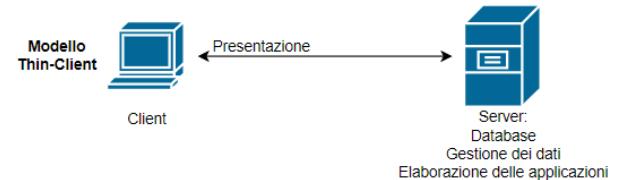
Es. sistemi di gestione del traffico con un'architettura Master-Slave



- **Architettura Client-Server a due livelli (Two-Tier)**, utilizzata per sistemi Client-Server semplici e nei casi in cui è importante centralizzare il sistema per ragioni di protezione; si hanno due modalità:

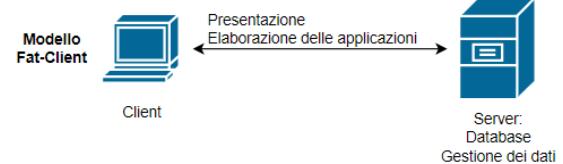
Thin-Client, in cui si hanno applicazioni:

- Di sistemi ereditati, dove non è possibile separare l'elaborazione delle applicazioni dalla gestione dei dati. I client possono accedere a queste come servizi.
- Di calcolo intensivo, come i compilatori con gestione di dati minima o inesistente.
- Con grandi volumi di dati (ricerca e interrogazione), ma con elaborazione minima. La semplice navigazione nel Web è il tipico esempio di utilizzazione di questa architettura.

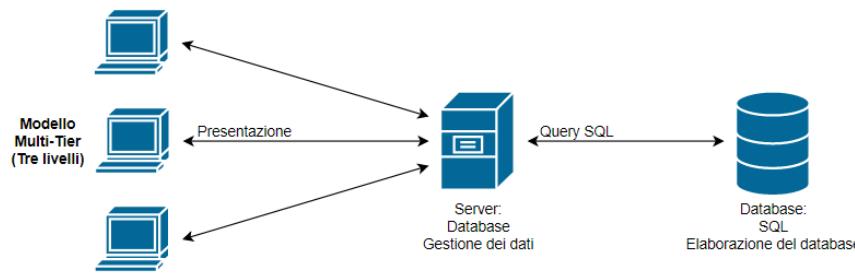


Fat-Client, in cui si hanno applicazioni:

- Dove l'elaborazione fornita da software off-the-shelf (per esempio, Microsoft Excel) avviene sul client.
- Dove è richiesta un'elaborazione con calcoli intensivi sui dati (per esempio, la visualizzazione di dati).
- Dove la connessione a Internet non può essere garantita. È quindi possibile un'elaborazione locale che usa le informazioni del database memorizzate nella cache.



- **Architettura Client-Server a più livelli (Multi-Tier)**, utilizzata quando i server deve elaborare un grande volume di transazioni;



Applicazioni su vasta scala con centinaia o migliaia di client.

Quando i dati e le applicazioni sono entrambi volatili.

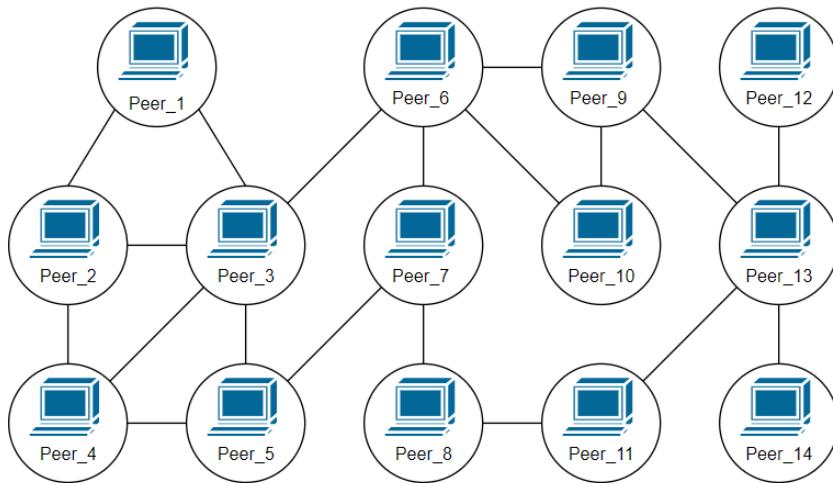
Applicazioni dove i dati vengono integrati da più sorgenti.

- **Architettura di componenti distribuiti**, utilizzata quando è necessario combinare insieme le risorse di vari sistemi e database, o come modello di implementazione per i sistemi Client-Server Multi-Tier;
- **Architettura peer-to-peer**, utilizzata quando i client si scambiano localmente le informazioni memorizzate e il ruolo del server consiste nel presentare un client a un altro client.
Può essere utilizzata anche quando devono essere eseguiti numerosi calcoli indipendenti.

- **Peer-to-peer**, è una rete che connette i dispositivi tra di loro, senza necessità di passare attraverso un server centrale:

- Non esiste un'entità che possa controllare la rete;
- Non esiste un punto di rottura singolo;
- Nel caso di Whatsapp, quando i server dell'azienda non funzionano il servizio non è più disponibile.

In una rete peer-to-peer, fintanto che i dispositivi sono connessi ad internet e funzionanti, niente può causare problemi.



Blockchain:

In una rete peer-to-peer, la mancanza di autorità di un solo ente rende necessario un insieme di regole e strumenti che i dispositivi possano utilizzare per comunicare tra di loro.

Nelle reti con un server centrale è il server a fare una crittografia dei messaggi inviati e ricevuti, a disporli in ordine cronologico nel proprio database e trasmetterli secondo certi criteri ai giusti dispositivi; in un sistema peer-to-peer, questo compito può essere svolto dalla **blockchain**.

La **blockchain** è una catena di blocchi contenenti dati organizzate opportunamente crittografati.

Le attività svolte all'interno della rete, come inviare denaro o ricevere documenti, vengono crittografate e comunicate a tutti i dispositivi della rete; i dispositivi organizzano i dati in ordine cronologico e sistematico, fino a formare un blocco.

Il blocco deve rispettare precise regole matematiche, altrimenti sarà evidente che qualcuno ha manomesso i dati o che le registrazioni dei dati sono avvenute nell'ordine sbagliato.

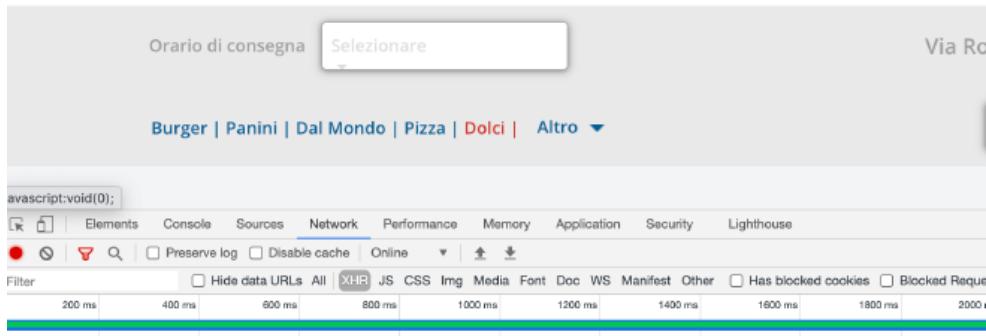
Se il blocco rispetta tali regole viene *validato* ed aggiunto alla catena.

Il problema fondamentale alla base delle reti peer-to-peer e della regolamentazione tramite blockchain è la **fiducia** (Es. quando inviamo un bonifico, ci fidiamo della nostra banca; quando inviamo un messaggio, ci fidiamo di Whatsapp e così via...)

La blockchain elimina completamente la necessità di fiducia alla base del sistema:

- Non occorre fidarsi di alcuna istituzione, non c'è autorità che tiri le fila della rete e le "regole" sono definite in partenza, applicate verificate attraverso algoritmi matematici;
- I protocolli blockchain possono eliminare molti dei problemi connessi all'era digitale.

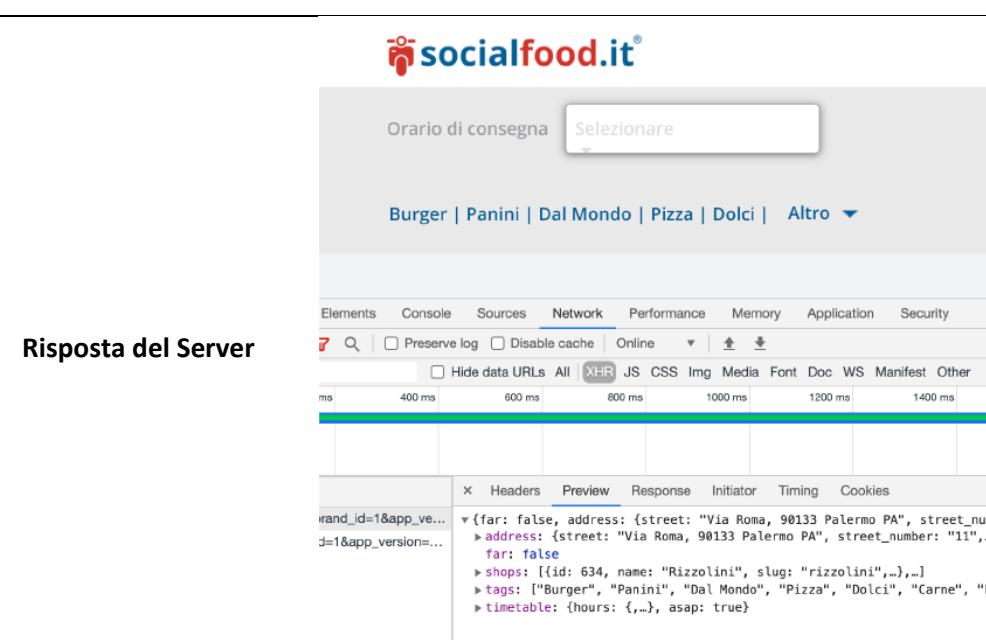
Esempio riepilogativo:

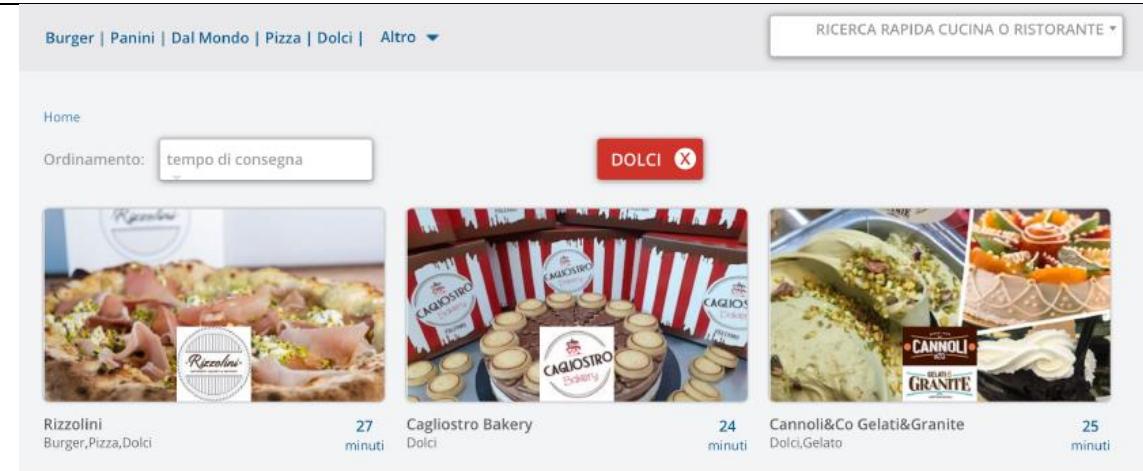


Richiesta del Client



Risposta del Server





Visualizzazione

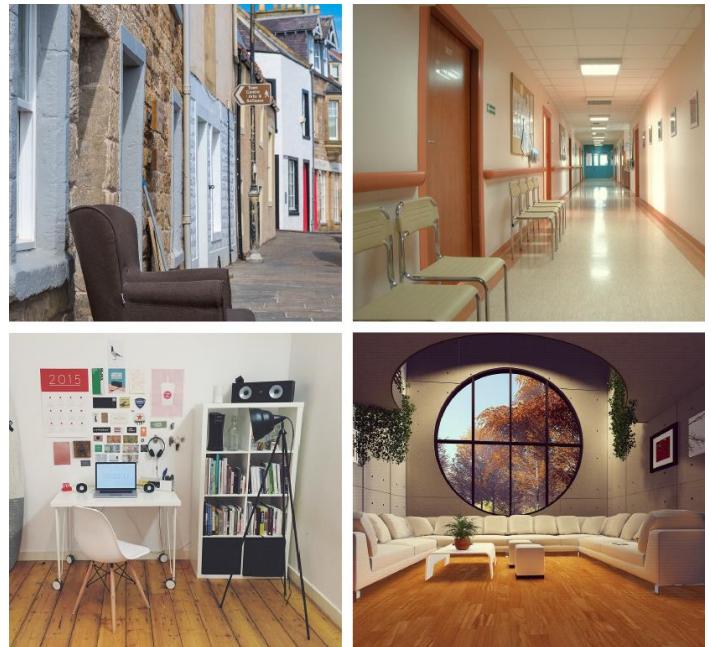
DESIGN PATTERNS

Supponiamo di voler leggere comodamente un libro, in quale dei quattro ambienti, qua accanto, ci sentiremmo a nostro agio?

Ciascuno ha le sue comodità, altre meno, ad esempio:

La poltrona sarà comoda e avrà una buona luce naturale, ma il luogo in cui si trova potrebbe essere affollato; le sedie nel corridoio sono sufficienti, ma l'ambiente ospedaliero e la luce artificiale non sono tra le scelte migliori; similmente la stanzetta, comodità sufficiente, luogo calmo, ma la luce è artificiale; infine il soggiorno ha: divano comodo, luce naturale e luogo silenzioso.

L'architetto C. Alexander introdusse l'idea di **linguaggio di pattern** nel libro "The Timeless Way of Building".



Un linguaggio di pattern è un insieme di regole che costituisce un vocabolario di progetto in un certo ambito o dominio, ad esempio, in un altro libro, "A Pattern Language", Alexander introduce un linguaggio speciale con sottolinguaggi per progettare città (per urbanisti) edifici (per architetti), e costruzioni in genere (per costruttori).

Schema descrittivo di un design pattern:

- **Contesto**, situazione problematica di progetto;
- **Problema**, insieme di forze in gioco in tale contesto;
- **Soluzione**, forma o regola da applicare per risolvere tali forze.

Es. 1

Pattern: Corridoio corto

- **Contesto:**
 - I corridoi lunghi e inutili rappresentano le cose peggiori dell'architettura moderna.
- **Problema:**
 - Ansia dei pazienti nei corridoi ospedalieri;
 - Corridoi più lunghi di 16/17 metri vengono concepiti come un disagio.
- **Soluzione:**
 - Corridoi brevi, simili a stanze, con tappeti o pavimenti in legno, mobili, librerie e ariose finestre.

Es. 2

In base all'esempio precedente, il nostro obiettivo/problema è un "punto luce"

Forze:

- Vogliamo star seduti e comodi in una stanza;
- Saremo attirati dalla luce.

Soluzioni:

- In soggiorno, almeno una finestra dev'essere un "punto luce";
- Se una poltrona è lontana da un punto luce occorre crearne uno.

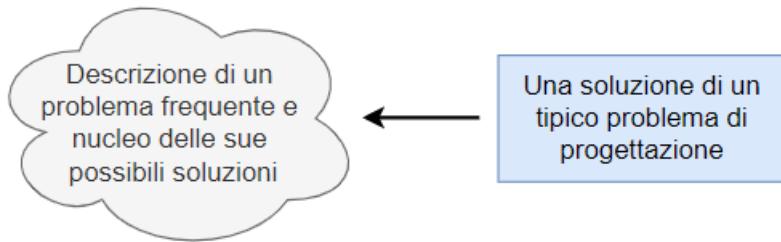
Si evince che in base ai quattro ambienti, il soggiorno è il migliore.

Un **pattern** è una soluzione ricorrente ad un problema standard, anche per lo sviluppo degli stessi software. Come si progetta un software? Tutti conosciamo:

- **Strutture dati;**
- **Algoritmi;**
- **Classi.**

Questi, quando si scrive un software, formano un **vocabolario progettuale**, tuttavia questi soli non bastano, esistono per questo dei livelli più alti di **design**.

Cos'è un design pattern?



Come già detto, ogni pattern descrive un *problema* che compare di continuo nel nostro ambito, e quindi descrive il *nucleo* di una *soluzione* a tale problema, in modo che si possa usare tale soluzione *un'infinità di volte*, senza *mai* farlo allo stesso.

Ogni pattern si presenta come descrizione di relazioni tra classi o tra oggetti per risolvere un problema progettuale in uno specifico contesto (es. il pattern Adapter permette l'interazione tra oggetti con interfaccia incompatibile).

Progettare da zero è costoso, i pattern ci aiutano a promuovere il riuso di soluzioni note, abilitando così architetture software, cioè strutture riusabili di componenti.

I pattern permettono di sfruttare l'esperienza collettiva dei progettisti esperti, catturano esperienze reali e ripetute di sviluppo di sistemi SW, e promuovono buone prassi progettuali.

I pattern si possono combinare per costruire architetture software con proprietà specifiche.

Struttura dei design patterns:

- **Nome**, è un nome simbolico che descrive sinteticamente un problema di progettazione, le sue soluzioni, e le conseguenze della soluzione scelta;
- **Problema (o intento)**, descrive quando è utile applicare il pattern. Spiega il problema che risolve ed il suo contesto;
- **Soluzione (concettuale)**, illustra gli elementi del progetto, le loro relazioni e collaborazioni. Non fornisce un'implementazione ma una descrizione astratta del modo in cui una configurazione di elementi (classi e oggetti) può risolvere il problema di progettazione;
- **Conseguenze**, sono i risultati ed i vincoli che derivano dall'applicazione del pattern. Le conseguenze sono importanti al fine di poter valutare soluzioni alternative e per comprendere/stimare i costi e i benefici dell'applicazione del pattern.

Classifichiamo i Design Pattern in tre modi:

- **Creational patterns**, forniscono vari oggetti meccanismi di creazione, che aumentano flessibilità in: *What, Who, How e When*, e riutilizzo di codice esistente.

Permettono inoltre di configurare un sistema con oggetti “prodotto” che variano ampiamente in struttura e funzionalità, la configurazione può essere: **statica** (compile-time) o **dinamica** (run-time).



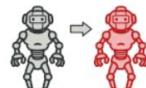
Factory Method, fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma consente alle sottoclassi di modificare il tipo di oggetti che verranno creati.



Abstract Factory, consente di produrre famiglie di oggetti correlati senza specificarne le classi concrete.



Builder, consente di costruire oggetti complessi passo dopo passo. Il modello consente di produrre diversi tipi e rappresentazioni di un oggetto utilizzando lo stesso codice di costruzione.

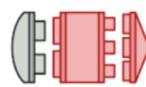


Prototype, consente di copiare oggetti esistenti senza rendere il tuo codice dipendente dalle loro classi.



Singleton, consente di assicurarti che una classe abbia solo un'istanza, fornendo al contempo un punto di accesso globale a questa istanza.

- **Structural patterns**, spiega come assemblare oggetti e classi in più grandi strutture mantenendo mentre queste strutture flessibili ed efficienti:



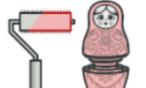
Adapter, consente agli oggetti con interfacce incompatibili di collaborare.



Bridge, consente di dividere una classe di grandi dimensioni o un insieme di classi strettamente correlate in due gerarchie separate, astrazione e implementazione, che possono essere sviluppate indipendentemente l'una dall'altra.



Composite, consente di comporre oggetti in strutture ad albero e quindi lavorare con queste strutture come se fossero oggetti singoli.



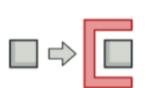
Decorator, consente di allegare nuovi comportamenti agli oggetti inserendo questi oggetti all'interno di speciali oggetti wrapper che contengono i comportamenti.



Facade, fornisce un'interfaccia semplificata per una libreria, un framework o qualsiasi altro insieme complesso di classi.



Flyweight, ti consente di adattare più oggetti alla quantità di RAM disponibile condividendo parti di stato comuni tra più oggetti invece di conservare tutti i dati in ciascun oggetto.



Proxy, consente di fornire un sostituto o un segnaposto per un altro oggetto. Un proxy controlla l'accesso all'oggetto originale, consentendo di eseguire qualcosa prima o dopo che la richiesta è arrivata all'oggetto originale.

- **Behavioral patterns**, riguardano gli algoritmi e l'assegnazione di responsabilità tra oggetti:



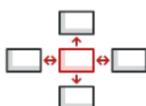
Chain of responsibility, consente di passare le richieste lungo una catena di gestori. Alla ricezione di una richiesta, ciascun gestore decide di elaborare la richiesta o di passarla al successivo gestore della catena.



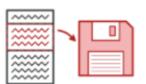
Command, trasforma una richiesta in un oggetto autonomo che contiene tutte le informazioni sulla richiesta. Questa trasformazione consente di passare le richieste come argomenti di metodo, ritardare o mettere in coda l'esecuzione di una richiesta e supportare operazioni annullabili.



Iterator, consente di attraversare gli elementi di una raccolta senza esporre la rappresentazione sottostante (elenco, stack, albero, ecc.).



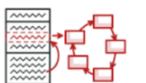
Mediator, consente di ridurre le dipendenze caotiche tra gli oggetti. Il modello limita le comunicazioni dirette tra gli oggetti e li costringe a collaborare solo tramite un oggetto mediatore.



Memento, consente di salvare e ripristinare lo stato precedente di un oggetto senza rivelare i dettagli della sua implementazione.



Observer, consente di definire un meccanismo di sottoscrizione per notificare a più oggetti eventuali eventi che si verificano all'oggetto che stanno osservando.



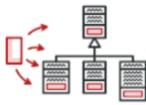
State, consente a un oggetto di modificare il suo comportamento quando cambia il suo stato interno. Sembra che l'oggetto abbia cambiato la sua classe.



Strategy, consente di definire una famiglia di algoritmi, inserirli in una classe separata e rendere i loro oggetti intercambiabili.



Template method, definisce lo scheletro di un algoritmo nella superclasse ma consente alle sottoclassi di sovrascrivere passaggi specifici dell'algoritmo senza modificarne la struttura.



Visitor, consente di separare gli algoritmi dagli oggetti su cui operano.

Nota: noi vedremo alcuni di questi design patterns.

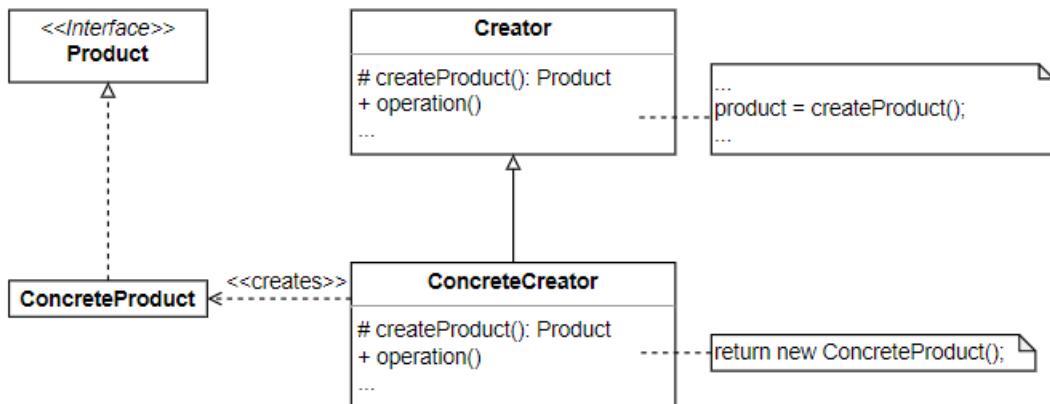
CREATIONAL DESIGN PATTERN

FACTORY METHOD

Definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la decisione del tipo di classe da istanziare.

Nonostante il nome, la creazione del prodotto non è la responsabilità principale del creatore.

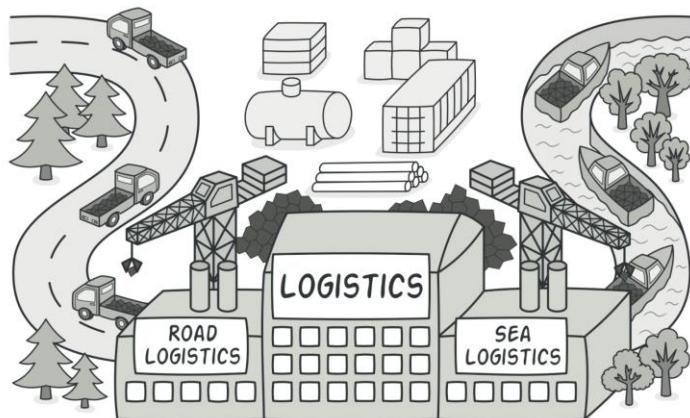
Di solito, la classe del creatore ha già una logica di core business relativa ai prodotti.



Il metodo factory aiuta a disaccoppiare questa logica dalle classi di prodotti concreti (es. una grande azienda di sviluppo software può avere un dipartimento di formazione per programmati. Tuttavia, la funzione principale dell'azienda nel suo complesso è ancora scrivere codice, non produrre programmati).

Problema:

Immaginiamo di creare un'applicazione per la gestione della logistica. La prima versione dell'app può gestire solo il trasporto su camion, quindi la maggior parte del codice risiede nella classe *Truck*.

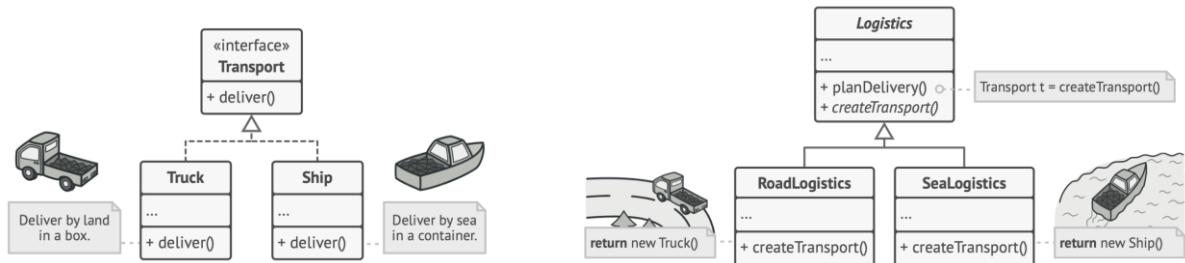


Dopo un po', l'app diventa piuttosto popolare. Ogni giorno riceviamo dozzine di richieste da parte di società di trasporto marittimo per incorporare la logistica marittima nell'app.

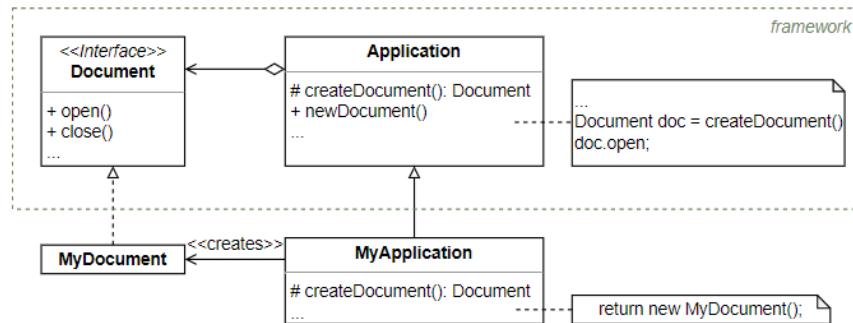
Ottimo, ma... per quanto riguarda il codice? Al momento, la maggior parte del codice è accoppiata alla classe *Truck*. L'aggiunta di navi all'app richiede la modifica dell'intera codebase. Inoltre, se in seguito decidiamo di aggiungere un altro tipo di trasporto all'app, probabilmente dovremo apportare nuovamente tutte queste modifiche.

Di conseguenza, ci ritroveremo con un codice piuttosto sgradevole, pieno di condizionali che cambiano il comportamento dell'app a seconda della classe degli oggetti di trasporto.

Soluzione:



I framework utilizzano classi astratte per definire e mantenere le relazioni tra gli oggetti e spesso sono anche responsabili della creazione di questi oggetti.

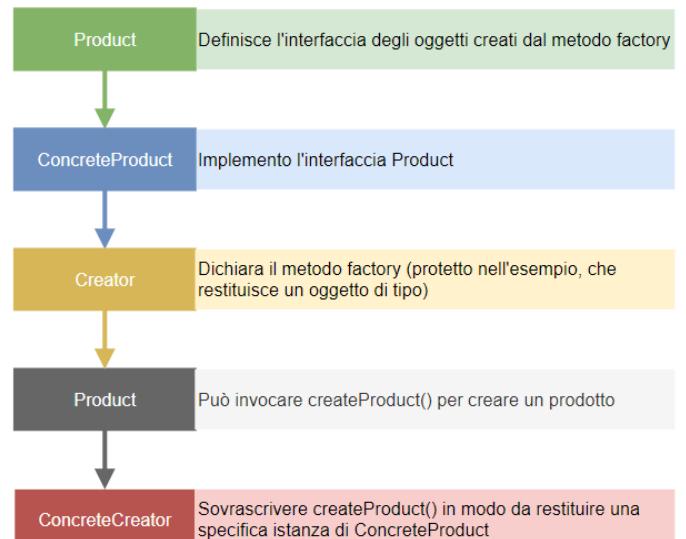


Si consideri un framework per applicazioni in grado di presentare più documenti agli utenti:

La classe *Application* è responsabile della gestione di oggetti conformi all'interfaccia *Document* e della loro creazione.

Application sa solo quando dovrà creare un nuovo documento ma non di che tipo è o come dovrà farlo.

La responsabilità è spostata all'esterno del framework: le sottoclassi di *Application* devono fornire l'implementazione del metodo factory *createDocument()* in modo da restituire un'istanza della classe appropriata.



Conseguenze:

- ✓ Elimina la necessità di riferirsi a classi dipendenti dall'applicazione all'interno del codice del framework;
- ✗ Gli utilizzatori potrebbero essere costretti a definire sottoclassi di Creator per creare un particolare oggetto ConcreteProduct;
- ✓ Fornisce un punto d'aggancio per le sottoclassi per la produzione di una versione specializzata di un prodotto;
- ✓ Connnette gerarchie di classi parallele;

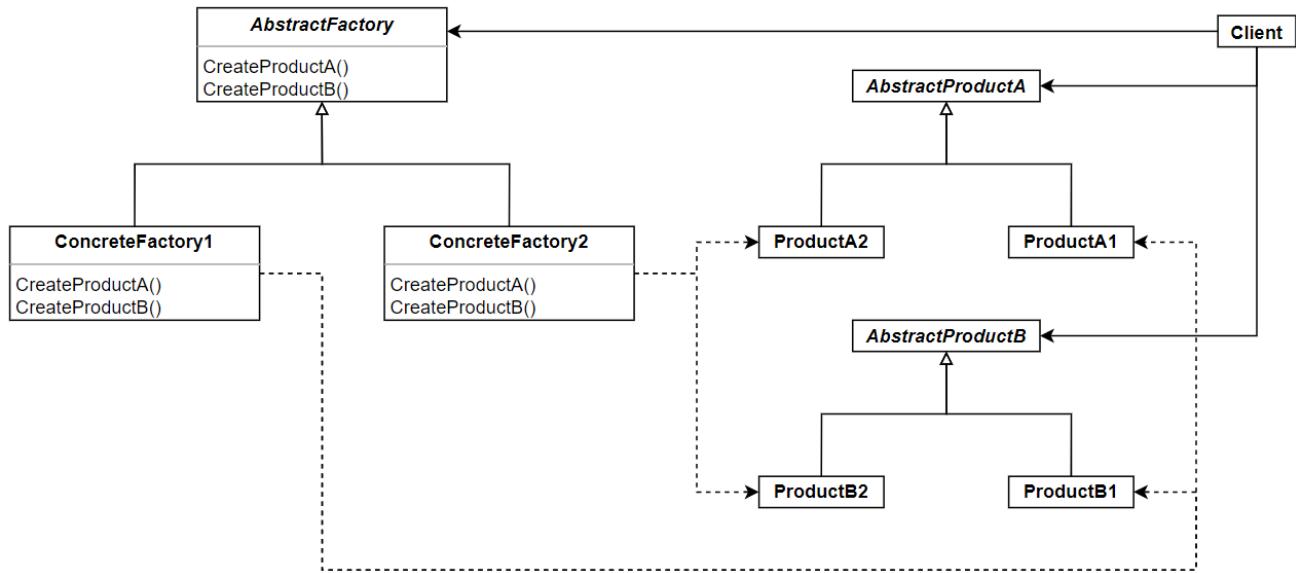
Riassumendo il Method Factory si utilizza quando:

una classe non può sapere in anticipo la classe di oggetti che deve creare;

una classe vuole che siano le sue sottoclassi a specificare gli oggetti da creare; le classi delegano la responsabilità ad una o più sottoclassi "aiutanti", si vuole localizzare la conoscenza di quale sottoclasse aiutante è il delegato.

ABSTRACT FACTORY

Lo scopo è quello di fornire un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.



Questo pattern si usa quando:

- Un sistema deve rimanere indipendente dal modo in cui i suoi prodotti vengono creati, composti e rappresentati;
- Un sistema deve essere configurato attraverso una o più famiglie di prodotti;
- Degli oggetti prodotto correlati e appartenenti alla stessa famiglia devono essere utilizzati insieme;
- Si vuole fornire una classe che rappresenti una libreria di prodotti, rivelando solo la loro interfaccia, non la loro implementazione.

Problema:

Vogliamo simulare un negozio di mobili, il codice è composto da classi che rappresentano:

- Una famiglia di prodotti correlati: Chair, Sofa, CoffeeTable;
- Diverse varianti di questa famiglia: Modern, Victorian, ArtDeco.

Serve un modo per creare singoli oggetti di arredo in modo che corrispondano ad altri oggetti della stessa famiglia.

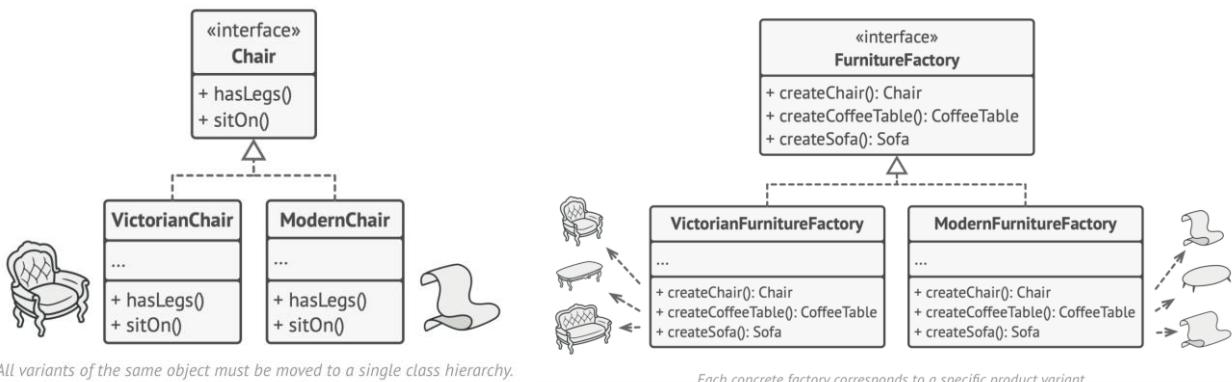


A Modern-style sofa doesn't match Victorian-style chairs.

Se i clienti ricevessero mobili non corrispondenti, potrebbero arrabbiarsi molto, inoltre, non si vuole modificare il codice esistente quando si aggiungono nuovi prodotti o famiglie di prodotti al programma.

I fornitori di mobili aggiornano i loro cataloghi molto spesso e non vorremmo cambiare il codice principale ogni volta che accade.

Soluzione:



Il client deve funzionare sia con le factory che con i prodotti tramite le rispettive interfacce astratte.
Supponiamo che il client desideri una factory per produrre una sedia:

- Il client non deve essere a conoscenza della classe della factory, né importa che tipo di sedia ottiene
- Che si tratti di un modello moderno o di una sedia in stile vittoriano, il client deve trattare tutte le sedie allo stesso modo, utilizzando l'interfaccia astratta della sedia;
- Con questo approccio, l'unica cosa che il client sa della sedia è che implementa in qualche modo il metodo `sitOn()`;
- Inoltre, qualunque variante di sedia venga restituita, corrisponderà sempre al tipo di divano o tavolino prodotto dallo stesso oggetto di fabbrica.

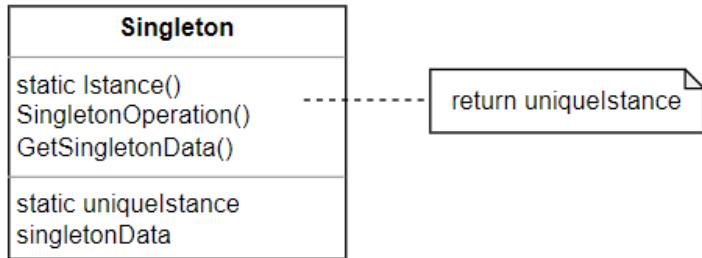
Conseguenze:

- ✓ **Isola le classi concrete**, aiuta a controllare le classi di oggetti che un'applicazione crea poiché una *factory* racchiude la responsabilità e il processo di creazione degli oggetti del prodotto, isolando così i client dalle classi di implementazione;
- ✓ **Rende facile lo scambio di famiglie di prodotti**, la *concrete factory* appare solo una volta in un'applicazione, ovvero dove viene istanziata. Ciò semplifica la modifica della *concrete factory* utilizzata da un'applicazione. Si possono utilizzare diverse configurazioni di prodotto semplicemente cambiando la *concrete factory*. Poiché una *abstract factory* crea una famiglia completa di prodotti, l'intera famiglia di prodotti cambia contemporaneamente;
- ✓ **Promuove la coerenza tra i prodotti**, quando gli oggetti del prodotto in una famiglia sono progettati per funzionare insieme, è importante che un'applicazione utilizzi gli oggetti di una sola famiglia alla volta. *AbstractFactory* rende facile da applicare questo concetto;
- ✓ **Supportare nuovi tipi di prodotti è difficile**, estendere le *abstract factory* per produrre nuovi tipi di prodotti non è facile, perché l'interfaccia *AbstractFactory* corregge l'insieme di prodotti che possono essere creati. Il supporto di nuovi tipi di prodotti richiede l'estensione dell'*AbstractFactory*, che implica la modifica della classe *AbstractFactory* e di tutte le sue sottoclassi.

SINGLETON

Assicura che una classe abbia una sola istanza, e fornisce un punto globale di accesso alla stessa

Un applicativo deve istanziare un oggetto che gestisce una stampante, questo oggetto deve essere unico, vale dire, deve esserci una sola istanza di esso, altrimenti potrebbero risultare dei problemi nella gestione della risorsa.



Il pattern si usa quando:

- Deve esserci esattamente un'istanza di una classe e deve essere accessibile ai client da un punto di accesso noto;
- L'unica istanza si deve poter estendere mediante sottoclassi e i client dovrebbero essere in grado di utilizzare un'istanza estesa senza modificare il proprio codice.

Problema:

Il problema è la definizione di una classe che garantisca la creazione di un'unica istanza all'interno del programma.

Soluzione:

Definisce una classe della quale è possibile istanziare un unico oggetto, tramite l'invocazione a un metodo della classe, incaricato della produzione degli oggetti.

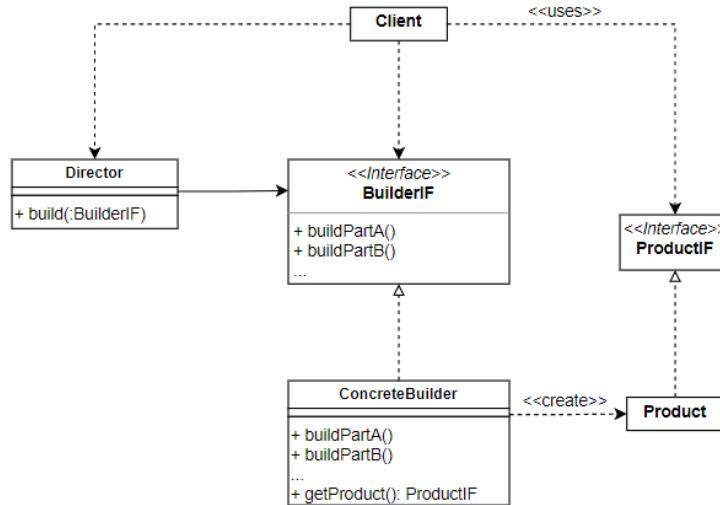
Diverse richieste di istanziare, comportano la restituzione di un riferimento allo stesso oggetto.

Conseguenze:

- ✓ **Accesso controllato all'istanza unica**, poiché la classe Singleton incapsula la sua unica istanza, può avere un controllo rigoroso su come e quando i client vi accedono;
- ✓ **Spazio dei nomi ridotto**, il modello Singleton è un miglioramento rispetto alle variabili globali. Evita di inquinare lo spazio dei nomi con variabili globali che memorizzano le sole istanze;
- ✓ **Permette l'affinamento delle operazioni e la rappresentazione**, la classe Singleton può essere sottoclasse ed è facile configurare un'applicazione con un'istanza di questa classe estesa. È possibile configurare l'applicazione con un'istanza della classe necessaria in fase di esecuzione;
- ✓ **Consente un numero variabile di istanze**, il modello rende facile cambiare idea e consente più di un'istanza della classe Singleton. Inoltre, è possibile utilizzare lo stesso approccio per controllare il numero di istanze utilizzate dall'applicazione. È necessario modificare solo l'operazione che **concede l'accesso all'istanza Singleton**.

BUILDER

Alcuni oggetti complessi richiedono di essere costruiti step by step.



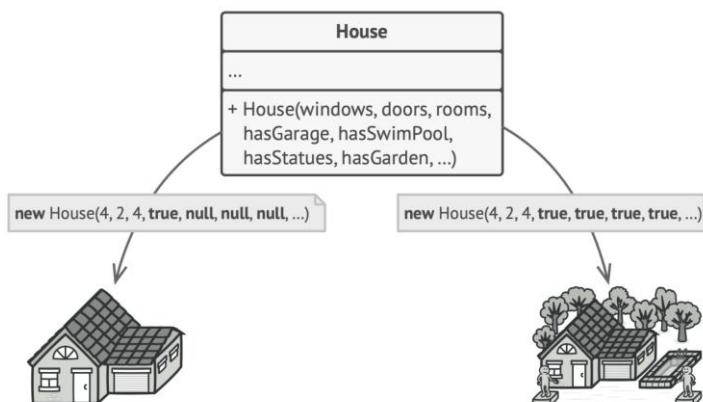
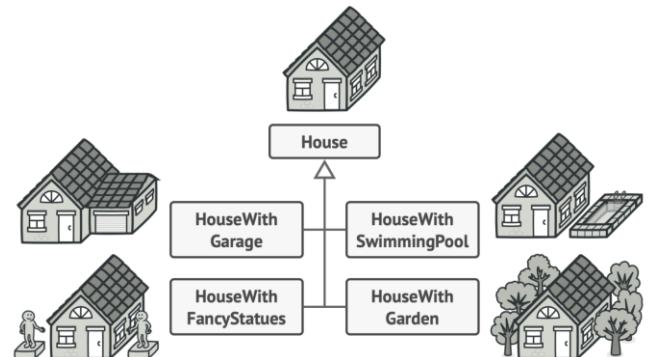
- **BuilderIF**: specifica l'interfaccia astratta che crea le parti dell'oggetto **Product**.
- **Concrete Builder**: costruisce e assembla le parti del prodotto implementando l'interfaccia **Builder**; definisce e tiene traccia della rappresentazione che crea.
- **Director**: costruisce un oggetto utilizzando l'interfaccia **Builder**.
- **Product**: rappresenta l'oggetto complesso e include le classi che definiscono le parti che lo compongono, includendo le interfacce per assemblare le parti nel risultato finale.

Il pattern si usa quando bisogna separare la costruzione di un oggetto complesso dalla sua rappresentazione, in modo tale che lo stesso processo di costruzione possa creare rappresentazioni diverse.

Problema:

Vogliamo costruire una casa che ha di base un certo tipo di struttura, ma può aggiungere altre caratteristiche nella stessa fase di creazione: un garage, una piscina, delle statuine, un giardino....

Il problema è capire quante sottoclassi dovremmo andare a creare per ogni possibile configurazione della casa.

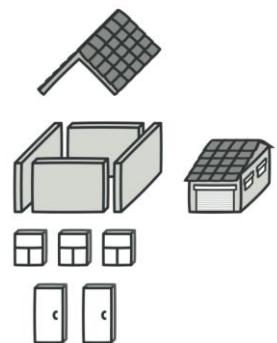
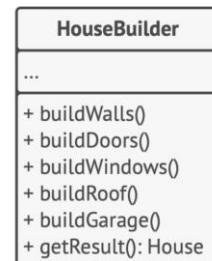


Un'ipotetica soluzione sarebbe la creazione di un costruttore con molti parametri, ma non tutti verrebbero utilizzati, e molti verrebbero trascurati per altri.

Soluzione:

Delegare a un *builder* la costruzione dell'oggetto complesso

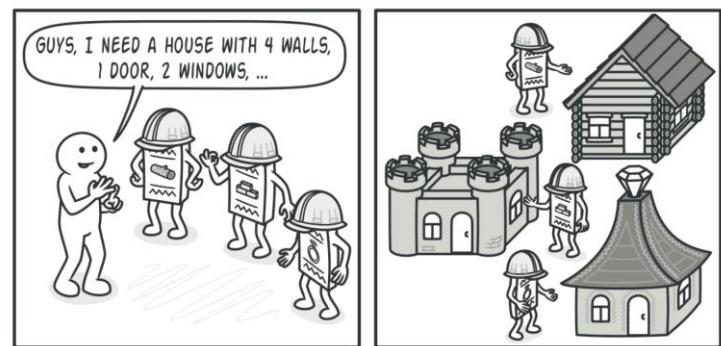
- Il pattern organizza la costruzione degli oggetti in una serie di passaggi (`buildWalls`, `buildDoor`, ecc.);
- Per creare un oggetto, si eseguono una serie di questi passaggi su un oggetto builder;
- La parte importante è che non è necessario chiamare tutti i passaggi, bensì solo quelli necessari per produrre una particolare configurazione di un oggetto.



Alcune delle fasi di costruzione potrebbero richiedere un'implementazione diversa quando è necessario creare varie rappresentazioni del prodotto (ad esempio, le pareti di una capanna possono essere costruite in legno, ma le mura del castello devono essere costruite con pietra).

In questo caso, è possibile creare diverse classi di builder che implementano lo stesso insieme di passaggi di creazione, ma in modo diverso; quindi, si può effettuare un insieme ordinato di chiamate al builder per produrre diversi tipi di oggetti.

Il builder avrà cura di evitare che altri oggetti possano accedere all'oggetto da costruire prima che questo sia completo.

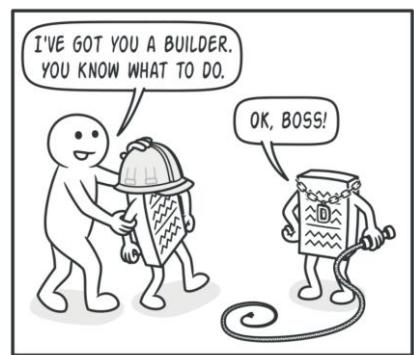


È possibile isolare una serie di chiamate al builder necessarie per costruire un prodotto all'interno di una classe separata chiamata **Director**.

La classe Director definisce l'ordine in cui eseguire le fasi di costruzione, mentre il builder fornisce l'implementazione per quelle fasi.

L'utilizzo del Director non è obbligatorio, tuttavia, consente di isolare varie routine di costruzione in modo da poterle riutilizzare nel programma.

Inoltre, la classe Director nasconde completamente i dettagli della costruzione del prodotto dal client, che in tal modo deve solo associare un Builder a un Director, avviare la costruzione con il Director e ottenere il risultato dal Builder.



Conseguenze:

- ✓ **Consente di variare la rappresentazione interna di un prodotto:**
 - L'interfaccia del Builder consente di nascondere la rappresentazione, la struttura interna dell'oggetto Product ed il processo di costruzione;
 - Per modificare la rappresentazione del Product basta definire un nuovo tipo di Builder.
- ✓ **Isola il codice per la costruzione e la rappresentazione:**
 - Migliora la modularizzazione incapsulando il modo di costruire e rappresentare un oggetto complesso;
 - Per i clienti non è necessario conoscere le classi che definiscono la struttura interna del prodotto poiché esse non appaiono nell'interfaccia del Builder.
- ✓ **Consente un migliore controllo del processo di costruzione:**
 - A differenza di altri pattern creazionali che costruiscono il prodotto nella sua interezza, in questo caso il prodotto viene costruito passo dopo passo sotto il controllo del Director.

Utilizzare un builder al posto di un costruttore con molti parametri:

I metodi factory statici ed i costruttori condividono una limitazione: non scalano bene quando ci sono molti parametri.

Si consideri una classe che rappresenti le etichette delle informazioni nutrizionali di un prodotto alimentare, ci sono alcuni campi che devono essere necessariamente specificati quali: dimensione di ciascuna porzione, numero di porzioni per contenitore e calorie per porzione

Molti altri campi sono opzionali : grassi totali, grassi saturi, colesterolo, sodio e così via... Molti prodotti specificano valori diversi da zero solo per alcuni dei campi opzionali.

```
public class NutritionFacts {
    private final int servingSize; // (mL) required
    private final int servings; // (per container) required
    private final int calories; // optional
    private final int fat; // (g) optional
    private final int sodium; // (mg) optional
    private final int carbohydrate; // (g) optional

    public NutritionFacts(int servingSize, int servings) { this(servingSize, servings, 0); }
    public NutritionFacts(int servingSize, int servings, int calories) { this(servingSize, servings, calories, 0); }
    public NutritionFacts(int servingSize, int servings, int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize = servingSize; this.servings = servings; this.calories = calories;
        this.fat = fat; this.sodium = sodium; this.carbohydrate = carbohydrate;
    }
}
```

Che tipo di costruttore o factory method occorre usare? Di solito, la soluzione adottata è fornire più costruttori:

- Un costruttore solo con i parametri richiesti;
- Un costruttore con un parametro opzionale, uno con due e così via;
- Fino ad un costruttore con tutti i parametri.

Quando occorre creare un'istanza si sceglie il costruttore con il minor numero di parametri che abbia almeno quelli che si vogliono specificare: `NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);`

Di solito ci sono più parametri di quelli che si vogliono specificare ma si è costretti a fornirli ugualmente... la situazione peggiora al crescere del numero di parametri.

I costruttori telescopici funzionano ma il codice cliente risulta difficile da scrivere e ancor più difficile da leggere in quanto , chi leggendo si chiede il significato di ciascun parametro, deve calcolare con cura la sua posizione.

Lunghe sequenze di parametri dello stesso tipo può dare origine a bachi difficili da rilevare a tempo di esecuzione.

Come soluzione alternativa (basata su Builder), si introduce un costruttore senza parametri per creare un oggetto, mentre i parametri richiesti si specificano invocando gli appositi metodi setter.

Non ha nessuno degli svantaggi del costruttore telescopico, tuttavia, poiché il processo di costruzione è diviso in varie chiamate l'oggetto creato potrebbe trovarsi in uno stato inconsistente. La classe non ha modo di verificare la consistenza dei parametri.

Preclude la possibilità di rendere gli oggetti della classe immutabili.

La soluzione migliore consiste nel ricorrere all'uso del pattern Builder:

la classe `NutritionFacts` definisce un builder come inner class statica `Builder`;

La classe `Builder` riproduce nei suoi campi quelli della classe `NutritionFacts`;

Il costruttore di `Builder` riceve solo i parametri obbligatori;

La classe `Builder` introduce un metodo per ciascun parametro opzionale.

```
public class NutritionFacts {
    // Parameters initialized to default values (if any)
    private int servingSize = -1; // Required; no default value
    private int servings = -1; // idem
    private int calories = 0; private int fat = 0;
    private int sodium = 0; private int carbohydrate = 0;
    public NutritionFacts() { }

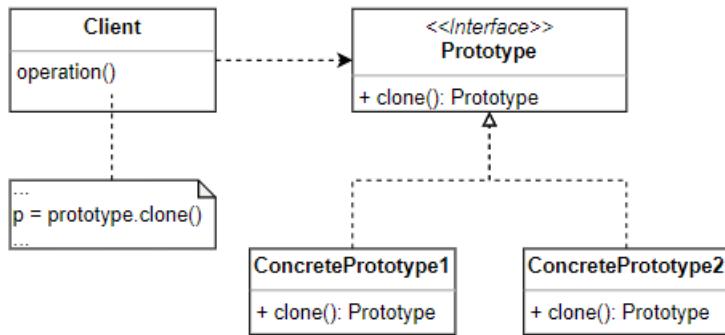
    public void setServingSize(int val){ servingSize = val; }
    public void setServings(int val){ servings = val; }
    public void setCalories(int val){ calories = val; }
    public void setFat(int val){ fat = val; }
    public void setSodium(int val){ sodium = val; }
    public void setCarbohydrate(int val){ carbohydrate = val; }
}
```

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Questi metodi settano il parametro e restituiscono il builder stesso, il metodo `build` crea l'oggetto fornendo atomicamente tutti i parametri per mezzo del builder.

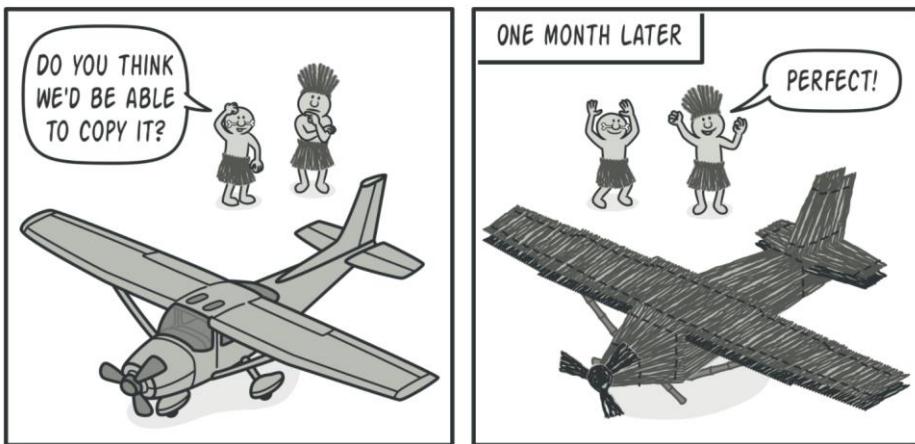
PROTOTYPE

Serve a effettuare la copia di oggetti senza rendere il codice dipendente dalle classi a cui appartengono gli oggetti, quindi, come faremmo a fare una copia esatta di un oggetto?



Problema:

Il problema che ci poniamo è come ottenere una copia esatta di un prodotto, copiarlo dall'esterno non sempre è possibile, né tantomeno garantisce la sua vera struttura.



Soluzione:

Vi è un'interfaccia comune per tutti gli oggetti che supportano il cloning.

L'interfaccia consente di effettuare la copia dell'oggetto senza duplicare il codice all'interno della classe concreta.

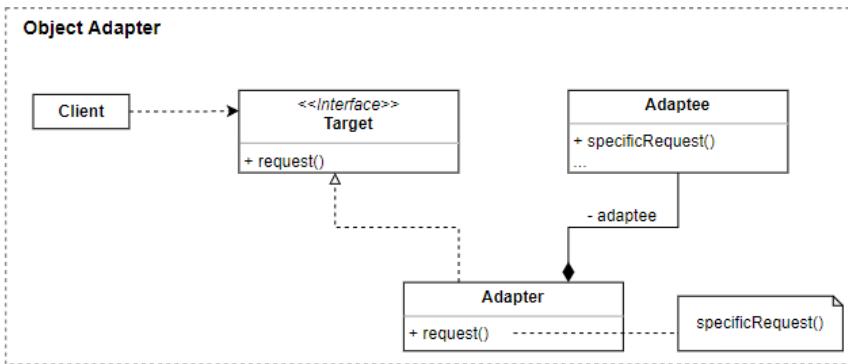
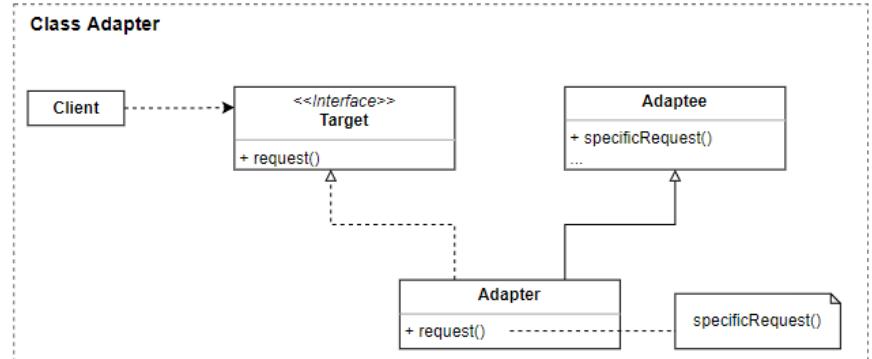
In genere, l'interfaccia contiene un solo metodo di cloning, la cui implementazione sarà simile in tutte le classi, si possono in tal modo creare oggetti che presentano configurazioni diverse e, quando è necessario copiarne uno, si effettua il clone del prototipo anziché creare un oggetto del tutto nuovo

STRUCTURAL DESIGN PATTERN

ADAPTER

Anche noto come Wrapper, ha lo scopo di convertire l'interfaccia di una classe in un'altra interfaccia richiesta dal client, consente a classi diverse di cooperare quando ciò non sarebbe possibile a causa dell'incompatibilità delle loro interfacce.

- **Target (Figure 2D)**: definisce l'interfaccia specifica del dominio utilizzata dal client;
- **Client**: collabora con oggetti compatibili con l'interfaccia Target;
- **Adaptee (XXXTriangle)**: individua un'interfaccia che deve essere adattata;
- **Adapter (Triangle)**: adatta l'interfaccia Adaptee all'interfaccia Target aborazioni.

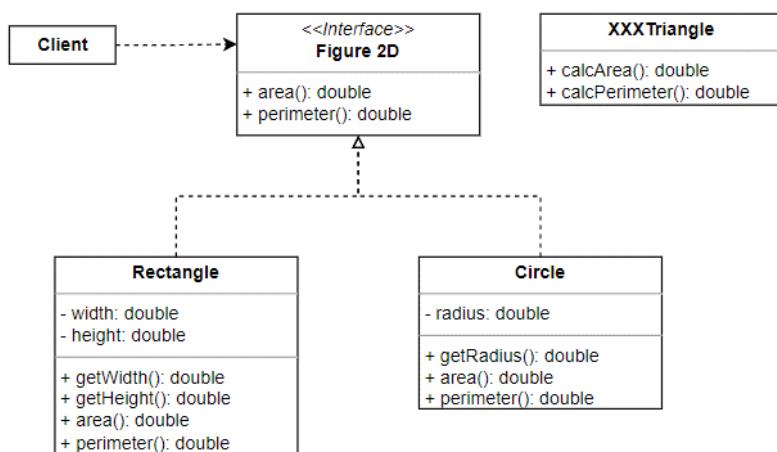


I client invocano le operazioni su un'istanza di Adapter.

L'adapter a sua volta invoca operazioni di Adaptee per soddisfare la richiesta.

Problema:

A volte una classe preesistente, progettata per essere riutilizzata, non può essere riusata solo perché la sua interfaccia non è compatibile con quella specifica richiesta dall'applicazione.



Nell'esempio, la classe **XXXTriangle** non può essere riusata dove ci si aspetta l'interfaccia **Figure2D** perché non è compatibile con essa.

Modificare **XXXTriangle** per renderla conforme a **Figure 2D** non è una buona soluzione: la si legherebbe ad un contesto specifico ed inoltre il sorgente potrebbe non essere disponibile.

Soluzione:

Si introduce una classe *Triangle* che sia al contempo erede di *XXXTriangle* e conforme all'interfaccia *Figure 2D*. L'implementazione dei metodi di *Figure 2D* sfrutta quelli di *XXXTriangle*.

L'ereditarietà non è sempre possibile, nel caso di Java: .

- Se *XXXTriangle* è final;
- Se *Figure 2D* fosse una classe e non un'interfaccia.

Si introduce una classe *Triangle* che implementa l'interfaccia *Figure2D* e le cui istanze incapsulano un oggetto di tipo *XXXTriangle*.

L'implementazione dei metodi di *Figure 2D* delega l'esecuzione all'oggetto incapsulato, evitando così i problemi di ereditarietà, diventando sempre applicabile.

Conseguenze:

Class Adapter:

- ✓ Detta l'interfaccia di *Adaptee* all'interfaccia *Target* basandosi su una classe concreta;
- ✗ Non può essere utilizzata se *Adaptee* è astratta oppure è un'interfaccia;
- ✓ Consente ad *Adapter* di sovrascrivere parte del comportamento di *Adaptee* essendo una sua sottoclasse;
- ✓ Non occorrono ulteriori direzioni per raggiungere l'oggetto adattato.

Object Adapter:

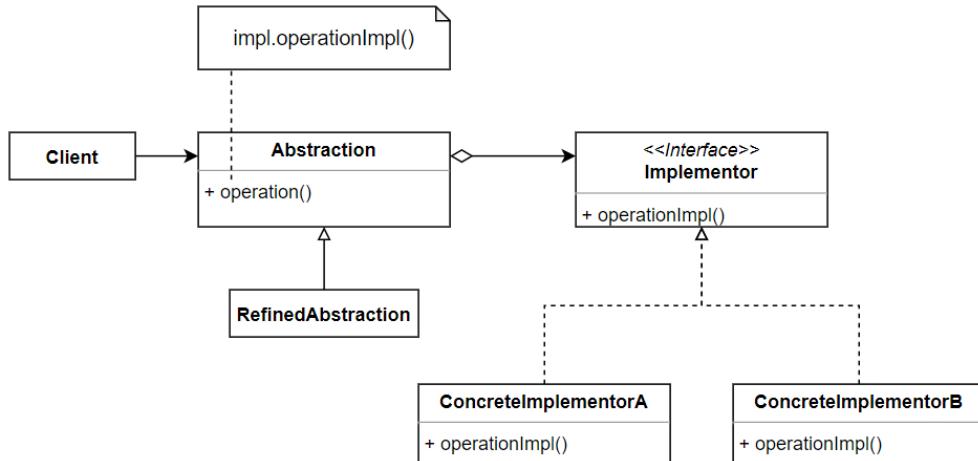
- ✓ Permette ad un singolo *Adapter* di operare con *Adaptee* e le sue sottoclassi, se esistono. Può in tal caso aggiungere funzionalità a tutti gli *Adaptee*;
- Rende difficile sovrascrivere il comportamento di *Adaptee* non essendo una sua sottoclasse;
- ✗ Aggiunge un livello di direzione per raggiungere l'oggetto adattato.

BRIDGE

Pattern strutturale, noto anche come Handle/Body, disaccoppia un'astrazione dalla sua implementazione in modo che le due possano variare indipendentemente l'una dall'altra.

Si vuole avere la possibilità di estendere sia le astrazioni che le implementazioni per mezzo dell'ereditarietà, si possono combinare astrazioni e implementazioni in vario modo ed estendere le indipendentemente dalle altre.

I cambiamenti nell'implementazione di un'astrazione non devono avere impatto sui client, si vuole condividere una stessa implementazione fra più oggetti nascondendo questa condivisione ai client.



- **Abstraction:** specifica l'interfaccia dell'astrazione. Mantiene un riferimento ad un oggetto di tipo **Implementor**;
- **RefinedAbstraction:** estende l'interfaccia di **Abstraction**;
- **Implementor:** definisce l'interfaccia per le classi che implementano l'astrazione.
Non deve corrispondere esattamente all'interfaccia di **Abstraction**: **Implementor** fornisce le operazioni base, mentre **Abstraction** definisce operazioni di più alto livello implementate sfruttando quelle di base.
- **ConcreteImplementor:** definisce un'implementazione concreta dell'interfaccia **Implementor**.

Quando un'astrazione può avere una tra più implementazioni possibili, in genere si risolve il problema ricorrendo all'ereditarietà. L'astrazione viene definita da un'interfaccia o da una classe astratta e le sottoclassi concrete la implementano in modi differenti.

Tale approccio non è flessibile poiché l'ereditarietà lega un'implementazione a un'astrazione in modo permanente. Ciò rende difficile modificare, estendere e riusare astrazioni ed implementazioni in modo indipendente.

Es.

Nello sviluppo di una libreria di classi per supporto allo sviluppo di applicazioni matematiche, si definisce la possibilità di gestire due tipologie di matrici numeriche, come raffinamento dell'astrazione Matrix, che fornisce le operazioni generali di gestione di dati matriciali:

- La matrice completa, nella quale tutte le celle sono rappresentate come oggetti all'interno di essa (CompleteMatrix);
- La matrice sparsa, che contiene solo i valori diversi da zero (SparseMatrix).

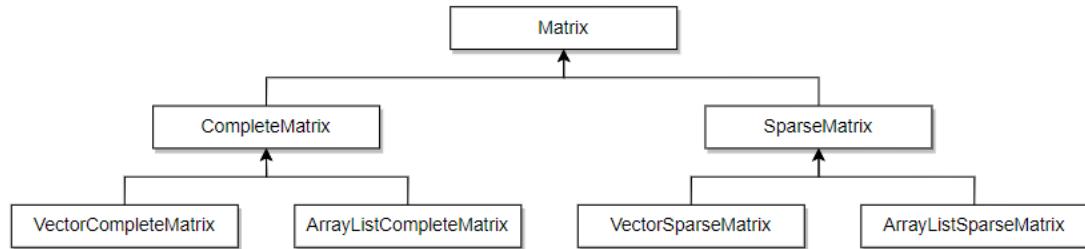
Indipendentemente del tipo di matrice da utilizzare, le celle della matrice (che saranno rappresentate da oggetti aventi come attributi le proprie coordinate e il valore da tenere) dovranno essere immagazzinate in qualche tipologia di collezione d'oggetti.

Assumiamo che gli sviluppatori della libreria considerino due tipi di collezioni per implementare le matrici, tra le diverse tipologie di collezioni fornite da Java: *java.util.Vector* e *java.util.ArrayList*.

Problema:

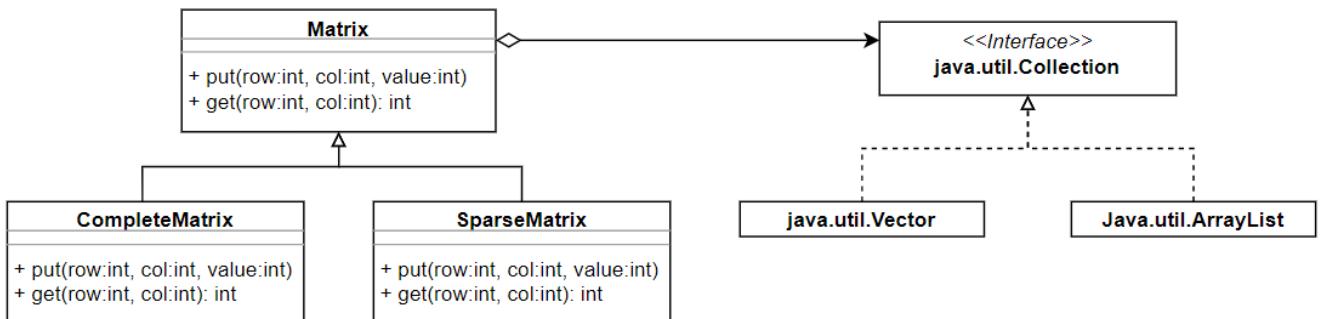
Sebbene intuitiva, questa gerarchia non è efficiente perché ogni volta che è necessario aggiungere un nuovo tipo di matrice, bisogna creare due classi specifiche, una per ogni tipo d'implementazione (ArrayList o Vector)

D'altro canto, se si decide utilizzare una nuova implementazione, sarà necessario aggiungere una nuova classe per ogni tipo di matrice esistente.



Il problema consiste nella definizione di un modo di rappresentare e gestire efficientemente entrambe le gerarchie, in modo tale che una possa variare senza avere impatto sull'altra

Soluzione:



Il pattern Bridge suggerisce la separazione dell'astrazione (gerarchia di Matrix) dall'implementazione (gerarchia di Collection), in gerarchie diverse, legando oggetti della seconda a quelli della prima, tramite una relazione di composizione.

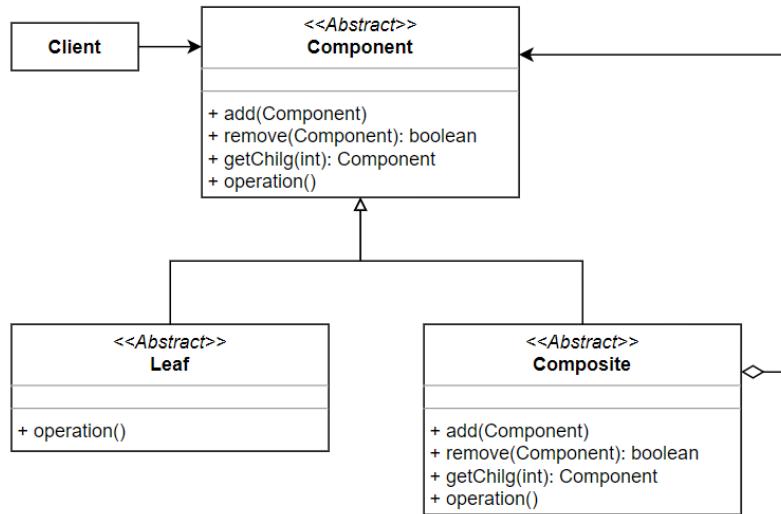
In questo modo ogni oggetto della gerarchia di Matrix sarà configurato con un particolare oggetto Collection da utilizzare.

COMPOSITE

È un Pattern strutturale che consente la costruzione di gerarchie di oggetti composti, gli oggetti composti possono essere formati da oggetti singoli, oppure da altri oggetti composti.

Questo pattern è utile nei casi in cui si vuole:

- Rappresentare gerarchie di oggetti tutto-parte;
- Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti.



- Component: classe astratta Component

- Dichiara una interfaccia comune per oggetti singoli e composti;
- Implementa le operazioni di default o comuni tutte le classi.

- Leaf: classe SinglePart

- Estende la classe Component, per rappresentare gli oggetti che non sono composti (foglie);
- Implementa le operazioni per questi oggetti.

- Composite: classe CompoundPart

- Estende la classe Component, per rappresentare gli oggetti che sono composti;
- Immagazzina al suo interno i propri componenti;
- Implementa le operazioni proprie degli oggetti composti, e particolarmente quelle che riguardano la gestione dei propri componenti;

- Client: in questo esempio sarà il programma principale

- Utilizza gli oggetti singoli e composti tramite l'interfaccia rappresentata dalla classe astratta Component.

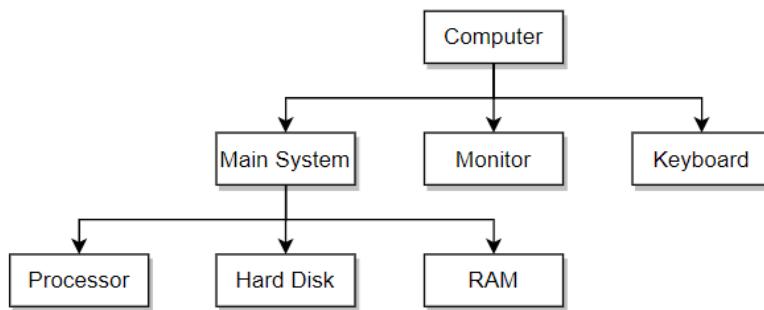
Es.

Nel magazzino di una ditta fornitrice di computer ci sono diversi prodotti, quali computer pronti per la consegna, e pezzi di ricambio (o pezzi destinati alla costruzione di nuovi computer)

Dal punto di vista della gestione del magazzino, alcuni di questi pezzi sono pezzi singoli (indivisibili), altri sono pezzi composti da altri pezzi: ad esempio, il “monitor”, la “tastiera” e la “RAM” sono pezzi singoli, mentre il “main system” è un pezzo composto da tre pezzi singoli (“processore”, “disco rigido” e “RAM”). Un altro esempio di pezzo composto è il “computer”, che si compone di un pezzo composto (“main system”), e due pezzi singoli (“monitor” e “tastiera”).

Problema:

Il problema è la rappresentazione omogenea di tutti gli elementi presenti del magazzino, sia dei singoli componenti, sia di quelli composti da altri componenti.



Soluzione:

Definisce la classe astratta componente (Component) che deve essere estesa in due sottoclassi:

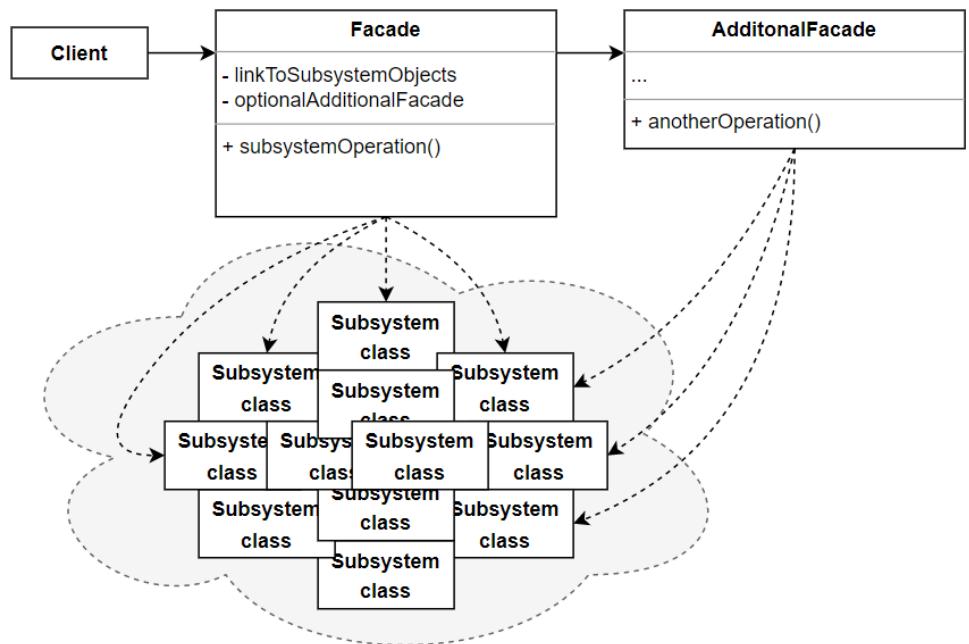
- Una che rappresenta i singoli componenti (Leaf);
- Un'altra (Composite) che rappresenta i componenti composti, e che si implementa come contenitore di componenti.

Il fatto che la seconda è un contenitore di componenti consente di immagazzinare al suo interno sia componenti singoli, sia altri contenitori (dato che entrambi sono stati dichiarati come sottoclassi di componenti).

FACADE

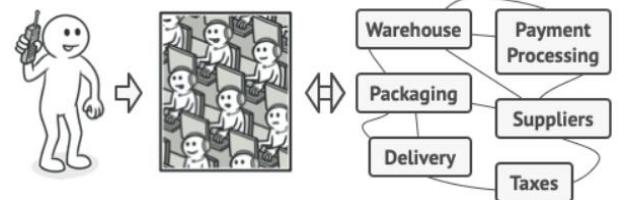
Fornisce un'interfaccia unificata per un insieme di interfacce presenti in un sottosistema; ovvero un'interfaccia di livello più alto che rende il sistema più semplice.

- **Facade:** conosce le classi nel sottosistema che sono responsabili di gestire una richiesta;
- **Classi del sottosistema:** implementano le funzionalità del sottosistema e non hanno alcuna conoscenza dell'esistenza del Facade: non hanno alcun riferimento ad esso.



Problema:

Immagina di dover far funzionare il tuo codice con un ampio insieme di oggetti che appartengono a una libreria o un framework sofisticato. Normalmente, dovresti inizializzare tutti quegli oggetti, tenere traccia delle dipendenze, eseguire metodi nell'ordine corretto e così via.



Placing orders by phone.

Di conseguenza, la logica aziendale delle tue classi diventerebbe strettamente accoppiata ai dettagli di implementazione delle classi di terze parti, rendendone difficile la comprensione e la manutenzione.

Soluzione:

Una facciata è una classe che fornisce una semplice interfaccia a un sottosistema complesso che contiene molte parti mobili. Una facciata potrebbe fornire funzionalità limitate rispetto all'utilizzo diretto del sottosistema. Tuttavia, include solo quelle funzionalità che interessano davvero ai clienti.

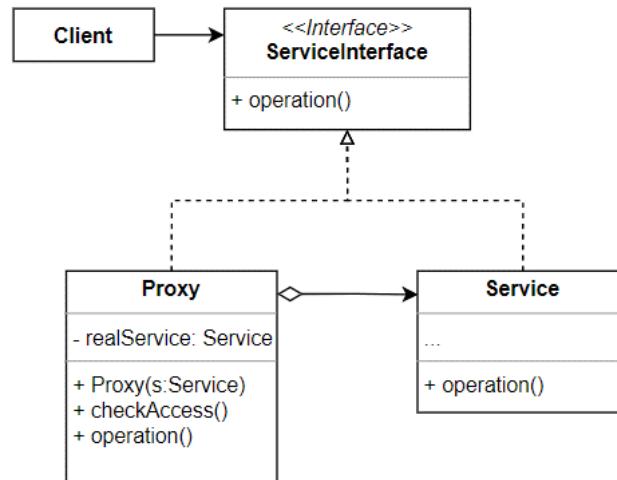
Conseguenze:

- ✓ Nasconde ai client componenti del sottosistema, riducendo il numero degli oggetti con cui interagiscono;
- ✓ Promuove un basso accoppiamento tra un sottosistema ed i suoi client. i componenti del sottosistema sono accoppiati tra di loro (alta coesione interna). La riduzione della dipendenza è di fondamentale importanza nei sistemi di grandi dimensioni.
- ✓ In particolare, riduce la dipendenza di compilazione se le classi del sottosistema sono modificate non è necessario ricompilare il codice dei client;
- ✓ non impedisce alle applicazioni client di utilizzare le classi del sottosistema qualora sia necessario;

PROXY

Noto anche come Surrogate, fornisce un surrogato o un segnaposto di un altro oggetto per controllare l'accesso all'oggetto.

- **Proxy**: mantiene un riferimento che gli consente di accedere all'oggetto di tipo Service di cui è il surrogato. Ha la stessa interfaccia di ServiceInterface. Controlla l'accesso all'oggetto rappresentato e può essere responsabile della sua creazione e dell'eliminazione;
- **ServiceInterface**: definisce l'interfaccia comune per Service e Proxy consentendo di usare un proxy ovunque ci si attende un Service;
- **Service**: definisci l'oggetto reale rappresentato dal Proxy.



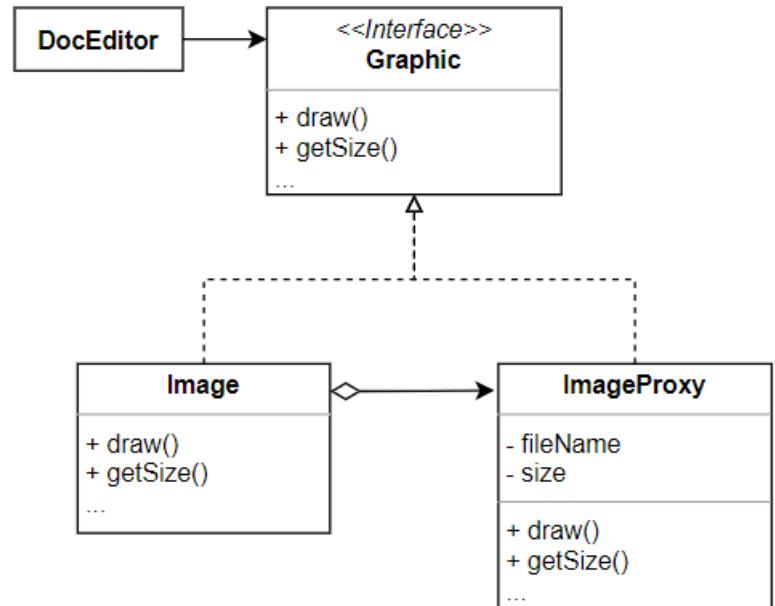
Problema:

Una ragione per effettuare un controllo sull'accesso a un oggetto può essere quella di rinviare il costo della sua creazione fino a che non è necessario.

Ad esempio, si consideri un editor che consente di rappresentare anche immagini all'interno dei documenti. Per velocizzare il caricamento in memoria dei documenti può essere opportuno ritardare il caricamento delle immagini fino a quando non è necessario visualizzarle.

Soluzione:

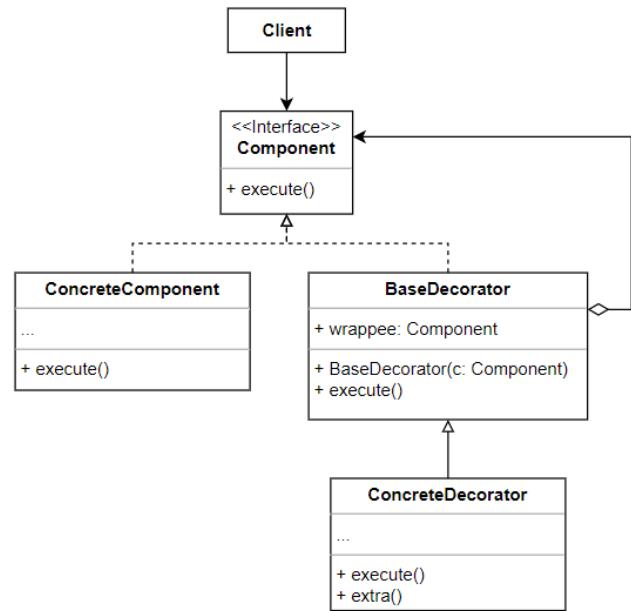
La soluzione consiste nell'utilizzo all'oggetto surrogato dell'immagine, ad esempio che occupi lo stesso spazio.



DECORATOR

Aggiunge dinamicamente responsabilità addizionali ad un oggetto, fornisce un'alternativa flessibile all'uso dell'ereditarietà come strumento per l'estensione delle funzionalità.

- **Component:** dichiara l'interfaccia comune sia per i wrapper che per gli oggetti di cui è stato eseguito il wrapping;
- **BaseDecorator:** ha un campo per fare riferimento a un oggetto avvolto. Il tipo del campo dovrebbe essere dichiarato come l'interfaccia del componente in modo che possa contenere sia componenti concreti che decoratori;
- **ConcreteComponent:** è una classe di oggetti che vengono avvolti. Definisce il comportamento di base, che può essere modificato dai decoratori;
- **ConcreteDecorator:** definiscono comportamenti extra che possono essere aggiunti dinamicamente ai componenti. I decoratori concreti sovrascrivono i metodi del decoratore di base ed eseguono il loro comportamento prima o dopo aver chiamato il metodo genitore.



Problema:

Si pensi ad un modello di oggetti che rappresenta gli impiegati (Employee) di una azienda.

Tra gli impiegati, ad esempio, esistono gli Ingegneri (Engineer) che implementano le operazioni definite per gli impiegati, secondo le proprie caratteristiche.

Questi cambiamenti di tipologia di alcuni impiegati coinvolgono modifiche delle responsabilità definite per gli oggetti, alterandone le esistenti o aggiungendone nuove.

Per questa ragione, sarebbe di interesse definire un modo per aggiungere dinamicamente nuove responsabilità ad oggetto specifico, eventualmente con la ulteriore possibilità di toglierle.

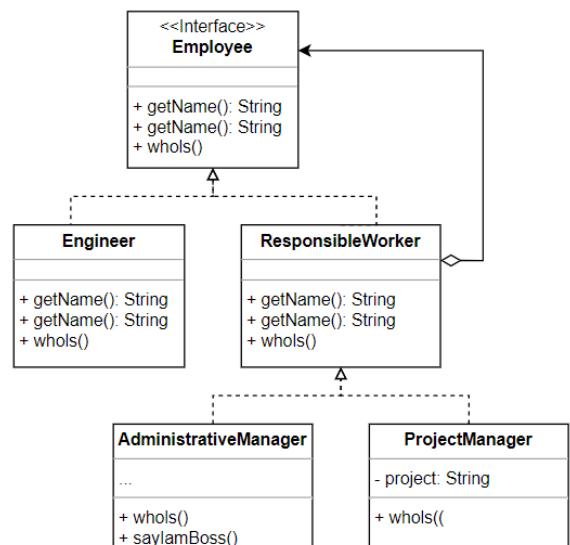
Soluzione:

Il pattern suggerisce la creazione di wrapper classes (Decorator) che racchiudono gli oggetti ai quali si vuole aggiungere le nuove responsabilità.

Questi ultimi oggetti, insieme ai Decorator devono implementare una interfaccia comune, in modo che l'applicazione possa continuare ad interagire con gli oggetti decorati.

Per una stessa interfaccia possono esserci più Decorator, ad esempio, per investire i ruoli di capoufficio e di responsabile di un progetto.

Il fatto che Decorator e oggetti decorati implementino la stessa interfaccia, consente anche l'applicazione di un Decorator ad un altro oggetto già decorato, ottenendo in questo modo la sovrapposizione di funzioni (ad esempio, un impiegato potrebbe essere investito come capoufficio e responsabile di un progetto contemporaneamente).



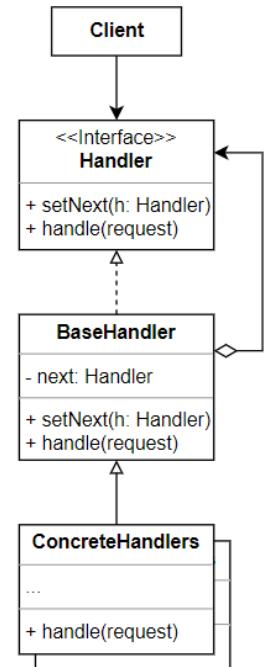
BEHAVIORAL DESIGN PATTERN

CHAIN OF RESPONSIBILITY

Consente di separare il mittente di una richiesta dal destinatario, in modo di consentire a più di un oggetto di gestire la richiesta.

Gli oggetti destinatari vengono messi in catena, e la richiesta trasmessa attraverso questa catena fino a trovare un oggetto in grado di gestirla.

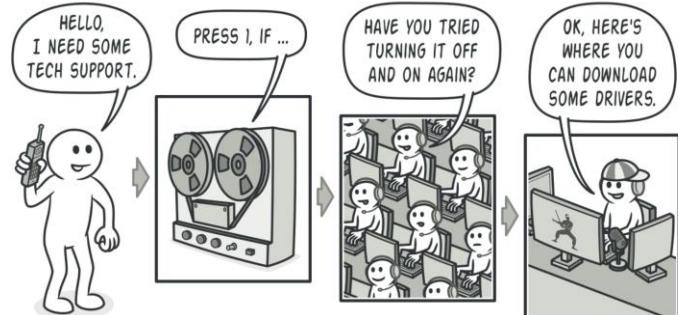
- **Handler:** dichiara l'interfaccia, comune a tutti i gestori concreti. Di solito contiene un solo metodo per gestire le richieste, ma a volte può avere anche un altro metodo per impostare il gestore successivo sulla catena.
- **BaseHandler:** è una classe facoltativa in cui è possibile inserire il codice boilerplate comune a tutte le classi del gestore.
- **ConcreteHandlers:** contengono il codice effettivo per l'elaborazione delle richieste. Al ricevimento di una richiesta, ciascun gestore deve decidere se evaderla e, in aggiunta, se passarla lungo la catena.



Problema:

In analogia con il caso reale possiamo immaginare un sistema di servizio di supporto.

Inizialmente sentiremo la voce robotica dell'autore responder, dei problemi tipici riscontrati, dopodiché, se nessuna di queste opzioni è valida allora veniamo messi in contatto con l'operatore, che si limita a dare suggerimenti da manuale. Infine, se neanche l'operatore soddisfa la nostra richiesta allora veniamo messi in contatto a uno degli ingegneri che sarà in grado o meno di soddisfare la nostra richiesta.



A call to tech support can go through multiple operators.

Soluzione:

Chain of responsibility propone la costruzione di una catena di oggetti responsabili della gestione delle richieste pervenute dal client.

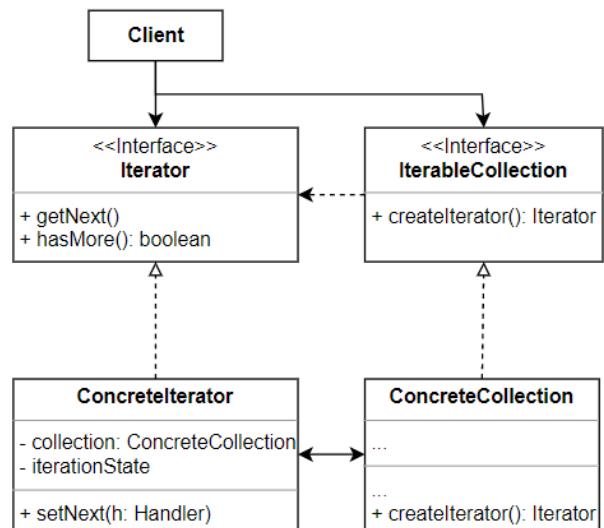
Quando un oggetto della catena riceve una richiesta, analizza se corrisponde a lui gestirla, o, altrimenti, la inoltra all'oggetto successivo nella catena.

In questo modo, gli oggetti a cui arriva la richiesta devono soltanto interfacciarsi con l'oggetto che si trova più basso nella catena di responsabilità.

ITERATOR

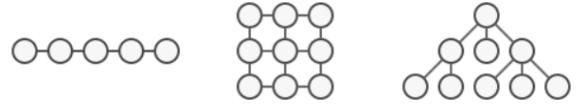
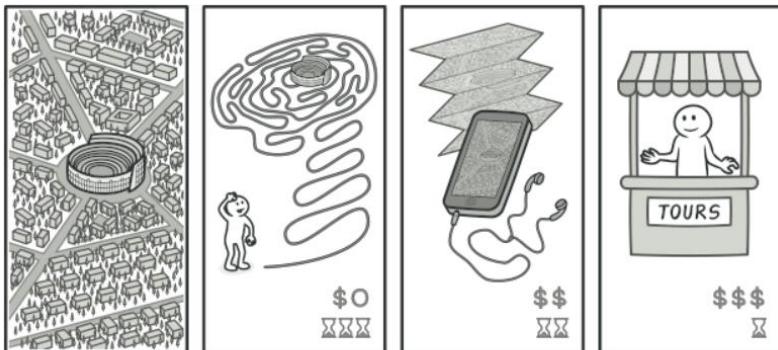
Consente di attraversare gli elementi di una collezione rendendo trasparente la rappresentazione della collezione (elenco, pila, albero, ecc.)

- **Iterator**: interfaccia ListIterator, specifica l'interfaccia per accedere e percorrere la collezione;
- **ConcreteIterator**: oggetto che implementa l'interfaccia ListIterator che tiene traccia della posizione corrente all'interno della collezione;
- **IterableCollection**: interfaccia List, specifica una interfaccia per la creazione di oggetti Iterator;
- **ConcreteCollection**: classe LinkedList, crea e restituisce una istanza di iterator.



Problema:

Il percorso di un viaggiatore è rappresentato come una collezione ordinata di oggetti, dove ogni oggetto rappresenta un luogo visitato.



Vari tipi di collezioni.

La collezione può essere implementata in base a un array, una linked list o qualunque altra struttura.

Una applicazione potrebbe essere interessata a poter accedere agli elementi di questa collezione in una sequenza particolare, ma senza dover interagire direttamente con il tipo di struttura interna usata per la rappresentazione.

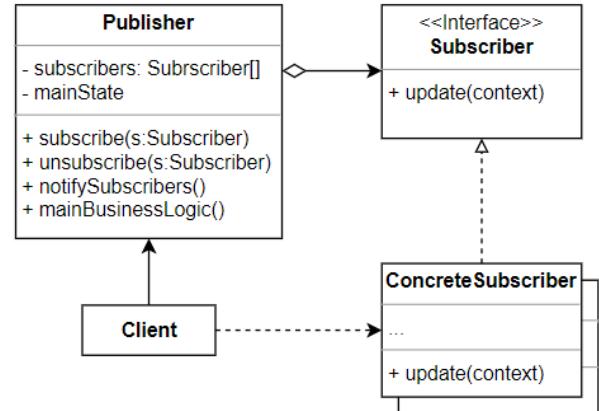
Soluzione:

La soluzione suggerita è l'implementazione di un oggetto che consenta l'acceso e scorimento della collezione, e che fornisca una interfaccia standard verso chi è interessato a percorrerla e ad accedere agli elementi.

OBSERVER

Consente di notificare a diversi oggetti cosa succede a un oggetto che tutti loro stanno «osservando»

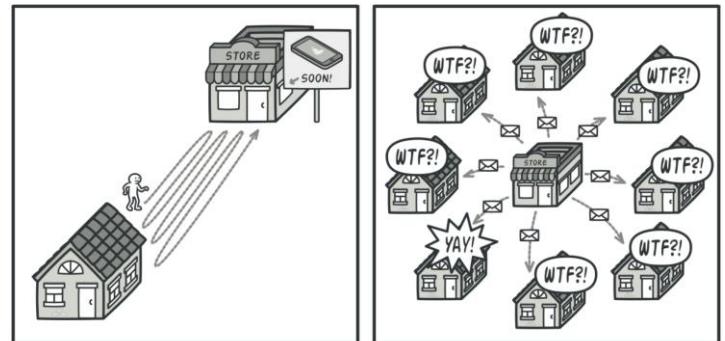
- **Publisher:** emette eventi di interesse per altri oggetti. Questi eventi si verificano quando l'editore cambia il suo stato o esegue alcuni comportamenti;
- **Subscriber:** dichiara l'interfaccia di notifica. Nella maggior parte dei casi, consiste in un unico update metodo. Il metodo può avere diversi parametri che consentono all'editore di trasmettere alcuni dettagli dell'evento insieme all'aggiornamento;
- **ConcreteSubscriber:** eseguono alcune azioni in risposta alle notifiche emesse dall'editore. Tutte queste classi devono implementare la stessa interfaccia in modo che l'editore non sia accoppiato a classi concrete.



Problema:

Immaginiamo di avere due oggetti: Customer e Store. Il cliente è molto interessato a una particolare marca di prodotto (es. un nuovo modello di iPhone) che dovrebbe essere disponibile presto nel negozio.

Il cliente può visitare il negozio tutti i giorni e verificare la disponibilità dei prodotti. Ma mentre il prodotto è ancora in viaggio, la maggior parte di questi viaggi sarebbe inutile.



Visiting the store vs. sending spam

D'altra parte, il negozio potrebbe inviare tonnellate di e-mail (che potrebbero essere considerate spam) a tutti i clienti ogni volta che diventa disponibile un nuovo prodotto. Ciò salverebbe alcuni clienti da viaggi infiniti al negozio. Allo stesso tempo, sconvolgerebbe altri clienti che non sono interessati ai nuovi prodotti.

Sembra che abbiamo un conflitto. O il cliente perde tempo a controllare la disponibilità del prodotto o il negozio spreca risorse avvisando i clienti sbagliati.

Soluzione:

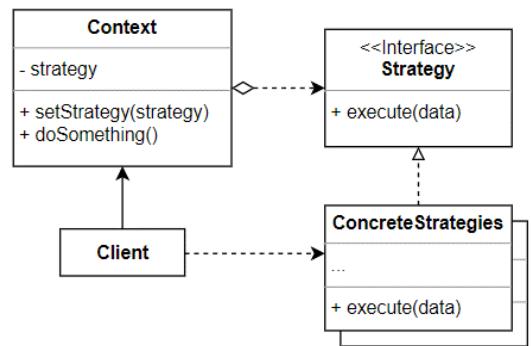
Il pattern suggerisce di aggiungere un meccanismo di sottoscrizione alla classe publisher in modo che i singoli oggetti possano sottoscrivere o annullare la sottoscrizione a un flusso di eventi provenienti da quel publisher.

STRATEGY

Consente la definizione di una famiglia di algoritmi, incapsula ciascuno di essi e li rende intercambiabili tra di loro.

Questo permette di modificare gli algoritmi in modo indipendente dai client che li utilizzano.

- **Context:** mantiene un riferimento a una delle strategie concrete e comunica con questo oggetto solo tramite l'interfaccia della strategia;
- **Strategy:** è comune a tutte le strategie concrete. Dichiara un metodo che il contesto utilizza per eseguire una strategia;
- **ConcreteStrategy:** implementano diverse varianti di un algoritmo utilizzato dal contesto.

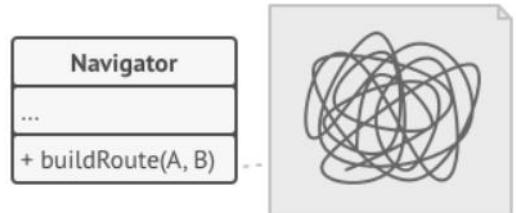


Problema:

Supponiamo di avere un app di navigazione per viaggiatori occasionali. L'app s'incentra su una mappa che aiuta gli utenti a orientarsi rapidamente in qualsiasi città.

Una delle funzionalità più richieste dell'app è la pianificazione automatica del percorso, l'utente inserisce un indirizzo e vede il percorso più veloce verso la destinazione.

La prima versione dell'app può solo costruire i percorsi su strade, via via aggiungiamo opzioni per costruire percorsi a piedi, per i mezzi pubblici, ciclisti....



The code of the navigator became bloated.

Mentre dal punto di vista aziendale l'app è stata un successo, la parte tecnica causa molti grattacapi. Ogni volta che aggiungiamo un nuovo algoritmo di routing, la dimensione della classe principale del navigatore raddoppia. Ad un certo punto, il programma è troppo difficile da mantenere.

Qualsiasi modifica a uno degli algoritmi, che si trattasse di una semplice correzione di bug o di un leggero aggiustamento del punteggio di strada, ha influenzato l'intera classe, aumentando la possibilità di creare un errore nel codice già funzionante.

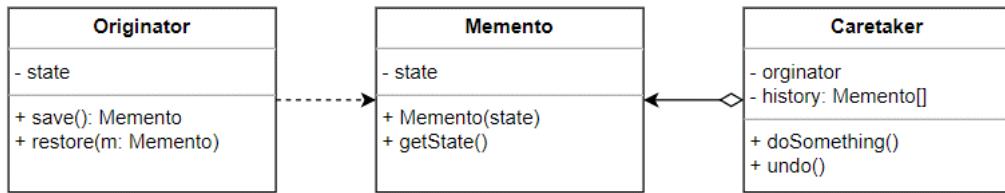
Soluzione:

Questo pattern suggerisce l'incapsulamento della logica di ogni particolare algoritmo, in apposite classi **ConcreteStrategy** che implementano l'interfaccia che consente agli oggetti **Context** di interagire con loro.

Questa interfaccia deve fornire un accesso efficiente ai dati del **Context**, richiesti da ogni **ConcreteStrategy**, e viceversa.

MEMENTO

È un pattern che consente di ripristinare lo stato precedente di un oggetto senza rivelare i dettagli interni della sua implementazione.



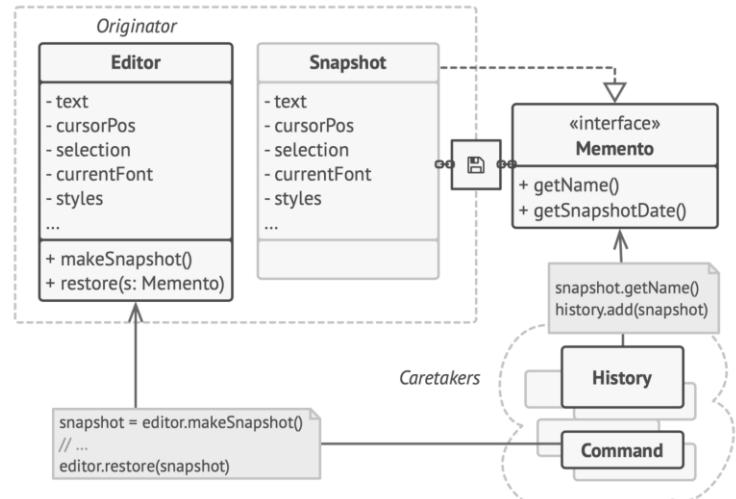
- **Originator**: può produrre istantanee del proprio stato, nonché ripristinare il proprio stato dalle istantanee quando necessario;
- **Memento**: è un oggetto valore che funge da istantanea dello stato dell'originatore. È pratica comune rendere immutabile il memento e passargli i dati una sola volta, tramite il costruttore;
- **Caretaker**: sa non solo "quando" e "perché" catturare lo stato dell'originatore, ma anche quando lo stato dovrebbe essere ripristinato.

Problema:

Si immagini di creare un'applicazione per gestire un editor di testo; oltre alla semplice modifica del testo, l'editor può formattare il testo, inserire immagini in linea, ecc....

Si vorrebbe anche consentire agli utenti di annullare qualsiasi operazione eseguita sul testo.

Per l'implementazione, si sceglie di adottare l'approccio «diretto»: prima di eseguire qualsiasi operazione, l'applicazione registra lo stato di tutti gli oggetti e lo salva in memoria. Quando un utente decide di annullare un'azione, l'applicazione recupera l'ultima istantanea dalla cronologia e la utilizza per ripristinare lo stato di tutti gli oggetti.



Soluzione:

Il pattern Memento delega la creazione degli snapshot di stato al proprietario effettivo di quello stato, l'oggetto originatore. Invece di altri oggetti che tentano di copiare lo stato dell'editor dall'"esterno", la stessa classe dell'editor può creare l'istantanea poiché ha pieno accesso al proprio stato.

Il modello suggerisce di memorizzare la copia dello stato dell'oggetto in un oggetto speciale chiamato **memento**.

I contenuti del «ricordo» non sono accessibili a nessun altro oggetto tranne che a quello che lo ha prodotto.

Altri oggetti devono comunicare con i ricordi utilizzando un'interfaccia limitata che può consentire il recupero dei metadati dell'istantanea (ora di creazione, il nome dell'operazione eseguita, ecc.), ma non lo stato dell'oggetto originale contenuto nell'istantanea.

CAPITOLO 3: Sviluppo del software

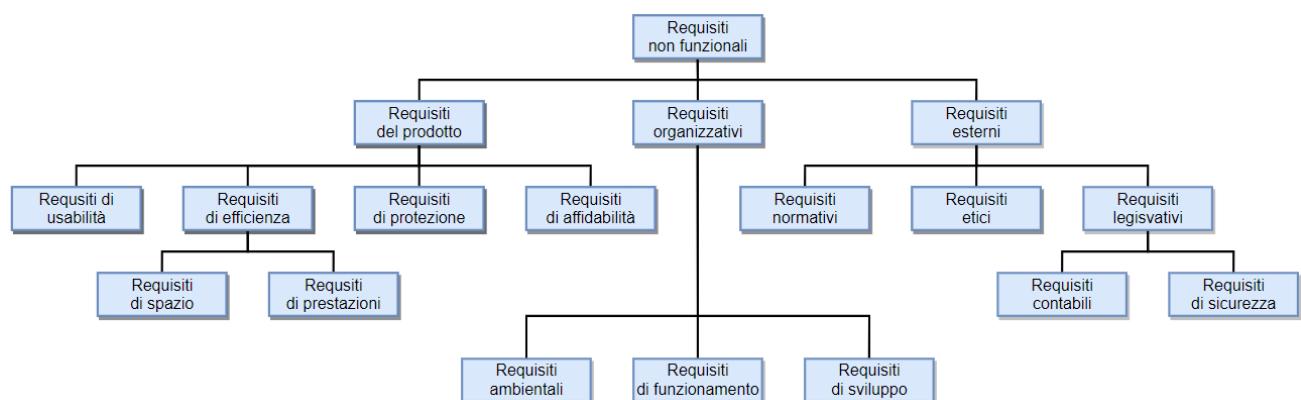
Stabiliamo a priori che studieremo il processo di sviluppo del software nel caso della metodologia agile, in particolare con riferimento a Scrum.

I REQUISITI

Prima di parlare di come si sviluppa un software, partiamo dalla base del suo ciclo di sviluppo.

Differiamo, innanzitutto, due tipi di requisiti:

- **I requisiti non funzionali**, devono essere rispettati a prescindere da come sono implementate le funzionalità del sistema, ma indirettamente influenzano questo processo;



Misurabilità:

Proprietà	Misura
Velocità	Transazioni elaborate al secondo Tempi di risposte a utenti/eventi Tempo di refresh dello schermo
Dimensione	Megabyte/Numeri di chip ROM
Facilità d'uso	Tempo di addestramento Numero di maschere d'aiuto
Affidabilità	Tempo medio di malfunzionamento Probabilità di indisponibilità Tasso di malfunzionamenti Disponibilità
Robustezza	Tempo per il riavvio dopo un malfunzionamento Percentuale di eventi che causano malfunzionamenti Probabilità di corruzioni dei dati dopo un malfunzionamento
Portabilità	Percentuale di istruzioni dipendenti dal sistema target Numero di sistemi target

- I **requisiti funzionali**, si riferiscono a specifiche funzionalità che bisogna implementare nel sistema: con quell'input deve essere restituito un determinato output, come il sistema deve gestire specifiche situazioni, ecc....

Adottando la metodologia agile si fa un disegno un generale del sistema, e di qualche parte più specifica ove e se necessario (es. sistema molto complesso).

Per ciascuna porzione del sistema:

- Formulazione della User Story;
- Disegno delle classi;
- Preparazione dei test;
- Implementazione guidata dai test per ciascuna delle unità;

Ogni **User Story** descrive **cosa** fare, per **chi** e **perché** in maniera semplice, comprensibile allo stesso modo per il cliente e per gli sviluppatori.

Il punto di vista è quello di chi richiede la nuova funzionalità, la quantità di informazioni è quella indispensabile per consentire al team di sviluppo di fare una stima di massima del lavoro richiesto per la realizzazione.

Ci sono diversi modi di scrivere una User Story ma solitamente questa contiene un nome, una breve descrizione e la specifica dei criteri di accettazione e delle condizioni per cui la story possa ritenersi completata.

Un modello può essere **WHO, WHAT, WHY**:

As a <type of user>, I want <some goal> so that <some reason>

Le User Story seguono le caratteristiche del modello I.N.V.E.S.T.:

- **Independent**: devono essere indipendenti tra loro;
- **Negotiable**: devono essere "negoziabili" ed aperte ai contributi di tutti;
- **Valuable**: devono portare valore aggiunto al cliente;
- **Estimable**: devono essere stimabili, non in maniera esatta, ma abbastanza da poter permettere una pianificazione di massima per l'implementazione;
- **Small**: devono essere piccole, in modo da riuscire a realizzare la funzionalità in massimo un paio di settimane di lavoro:
 - Devono essere piccole perché in questo modo le stime sono più precise;
 - Se la story è troppo complessa (Epica), la si scomponete in più story.
- **Testable**: devono poter essere testate e devono avere informazioni su come eseguire i test.

Alle User Story è importante elaborare degli **acceptance criteria**, ovvero dei criteri che devono essere utilizzati per valutare se una storia è stata implementata correttamente e pienamente.

Si tratta quindi delle condizioni che il prodotto SW deve rispettare per essere accettato dall'utente e/o cliente.

Gli acceptance criteria sono fondamentali per capire quando l'obiettivo viene raggiunto.

Le User Story devono essere visibili a tutto il team.

EPIC	USER STORY	ACCEPTANCE CRITERIA
As an Acquisition Gateway User, I need to access the Acquisition ordering platform behind a secure login	As an Acquisition Gateway User, I need to select an Auction product in the Acquisition ordering platform so that I can bid on it.	Ensure the Acquisition Gateway User is able to: <ul style="list-style-type: none"> - log in to Acquisition Gateway - navigate to the Auction page - able to select a product(s) to bid on
	As an Acquisition Gateway User, I need to review my previous bids in the Acquisition ordering platform so that I can remove expired bid	Ensure the Acquisition Gateway User is able to: <ul style="list-style-type: none"> - log in to Acquisition Gateway - navigate to a page to review items previously bid upon - select one, or multiple, expired bids - remove expired bids

Full User Story example

SCRUM

Scrum è una metodologia agile, incrementale e iterativa, per lo sviluppo di prodotti, applicazioni e progetti. È un framework, ovvero una modalità strutturata e pianificata, su cui è possibile costruire un sistema.

Esso, inoltre, è un **processo empirico**, ovvero basato sul concetto che la conoscenza deriva dall'esperienza e che le decisioni vadano prese alla luce di ciò che si conosce.

I tre pilastri che sostengono l'empirismo sono: *trasparenza, ispezione e adattamento*.

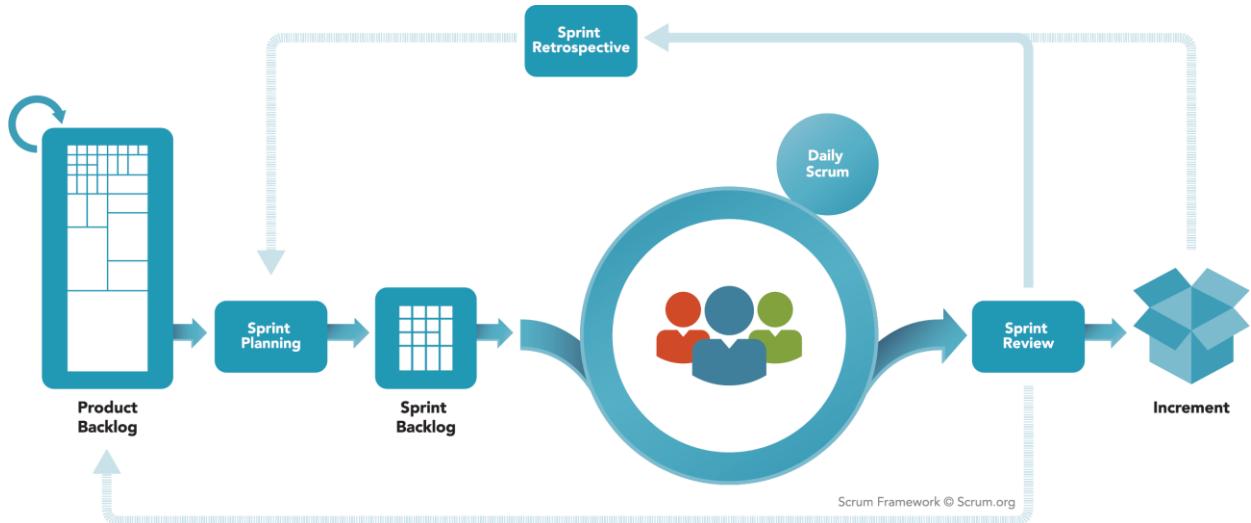
La "squadra" che compone questa metodologia è lo **Scrum Team**, un team di sviluppo è cross-funzionale e auto-organizzato, l'idea è utilizzare il pieno potenziale di ciascuno entro i propri ambiti di competenza, condividendo le responsabilità. A differenza dei modi di lavorare tradizionali non esiste un **Project Manager** che ha autorità su tutto e decide tutto.

Il vantaggio è notevole poiché coinvolgendo più persone rimuovi il potenziale collo di bottiglia dell'avere una sola persona chiamata a visionare e prendere decisioni su tutti gli aspetti del progetto.

Si può dire che in un Scrum Team esiste una leadership condivisa e essere leader vuol dire trovare il problema e mettere insieme le persone giuste per risolverlo. Chiunque nel team abbia la capacità di vedere un problema in anticipo, ha anche la possibilità di avere un ruolo da leader per la soluzione.

Scrum struttura il proprio **sviluppo in cicli** chiamati **Sprint** che durano da una a quattro settimane, al termine dei quali il team di sviluppo rilascia funzionalità immediatamente testabili.

I cicli sono **timeboxed**, il che significa che hanno durata fissa nel tempo, non possono essere estesi e terminano anche se il lavoro non è stato ultimato.



I cicli si suddividono in tre macro-fasi:

- **Pianificazione in Scrum**, all'inizio di ogni Sprint, durante un evento chiamato **Sprint Planning**, il team seleziona i propri task da una lista di attività prioritizzate (**Product Backlog**) e si impegna a completare tutte le attività selezionate entro il termine dello Sprint;

Il fine ultimo non è completare il maggior numero di attività possibile, ma produrre incrementi di software utilizzabili, raggiungendo lo Sprint Goal, ovvero l'obiettivo che si desidera ottenere durante l'iterazione: questo obiettivo è concordato tra team di sviluppo e Product Owner;

È importante sottolineare come tutto ciò che viene pianificato all'inizio dell'iterazione è una specie di contratto e non viene cambiato in corsa;

Lo scope, inteso come l'insieme di **User Story** necessarie a realizzare lo Sprint Goal, è fisso per tutta la durata del ciclo.

- **Esecuzione in Scrum**, il team si confronta quotidianamente mediante il Daily Stand-up o **Daily Scrum Meeting**, fondamentale al fine di ottenere una piena condivisione degli obiettivi e dei risultati;

Capita spesso che durante il daily standup vengano fuori idee, problemi e possibili dipendenze verso altri team (o fornitori):

- Se eseguita correttamente, il meeting dura un massimo di quindici minuti;
- Tutte le discussioni, come la soluzione degli eventuali problemi o impedimenti riscontrati va rimandata ad altre sedi.

Al termine dello Sprint il team rilascia ciò che è stato completato (Done), vale a dire tutto ciò che rispetta un elenco di requisiti predefiniti (Definition of Done);

Per esempio, nel caso di una normale applicazione software potrebbe voler dire una funzionalità integrata, funzionante, testata e rilasciabile;

- **Conclusione di uno Sprint**, lo Sprint si conclude con due eventi fondamentali: lo **Sprint Review** e il **Retrospective Meeting**.

- Lo Sprint Review è un meeting informale in cui il team valida il lavoro svolto insieme al Product Owner e a eventuali stakeholder;
- Il Retrospective Meeting, che in genere è svolto al termine del Review, serve al team per riflettere sullo Sprint appena concluso ed è un "luogo sicuro" dove ognuno può esprimere le proprie opinioni (in maniera costruttiva);

Entrambi questi eventi giocano un ruolo fondamentale e sono strettamente connesse a tutti e tre i **pilastri dell'empirismo**, in quanto promuovono trasparenza sul processo e sul lavoro svolto; ma anche ispezione e adattamento, attraverso un'analisi critica di ciò che ha funzionato durante l'iterazione appena conclusa e ciò che invece richiede miglioramenti.

Ciascuna fase ha delle componenti fondamentali che compongono gli Sprint, è importante che tali siano ben chiare affinché, altrettanto, i membri del team sappiano come gestirsi:

- **User Story e Acceptance Criteria**, le prime non sono nient'altro che i requisiti che, il programmatore, il cliente, l'utente finale... un qualsiasi attore, nel ruolo in cui ricopre, richiede nel sistema. Questi, poi, hanno bisogno di un'effettiva descrizione passo per passo di quello che devono fare, ovvero le Acceptance Critieria.
- **Product Backlog**, è un elenco di attività (anche le stesse User Story) e di feature ordinato per priorità.

Il Product Backlog è di proprietà del Product Owner, ed è redatto in seguito all'identificazione di una **vision** e alla stesura di una **Roadmap**.

Nonostante lo sviluppo agile si basi su piccole iterazioni, la Roadmap è uno strumento fondamentale per la definizione di obiettivi di medio e lungo termine.

Il Product Backlog viene costantemente rivisto e riordinato dal Product Owner in base alle necessità degli utenti o del cliente, le aspettative degli stakeholder, nuove idee, o in seguito alle esigenze di mercato ma anche in base a suggerimenti da parte del team.

All'interno di un Product Backlog trovano spazio diversi tipi di attività, come per esempio:

- Nuove feature e Change Request;
- Bug;
- Compiti tecnici (esempio: setup ambienti);
- Indagini e acquisizione competenze.

Es.

ID	Priority	Type	Item	Theme	SP	Status	Link to Bug Tracker/Wiki
1	1	Story	As a user I want to login so that I can access the user area	User Account	3	Done	http://bogus.agileway.it/issue/1102
2	1	Story	As a user I want to logout so that I can browse the website anonymously	User Account	1	In Progress	http://bogus.agileway.it/issue/1103
5	2	Bug	CSS is not working on login module		1	Not Started	http://bogus.agileway.it/issue/1100
3	2	Epic	As a user I want to register so that I can have an account	User Account		Not Started	
4	3	Investigation	Payment gateway			On Hold	

Per quanto riguarda i **bug**, dovrebbero anch'essi far parte del Product Backlog, nonostante ci siano pareri contrastanti in merito: Scrum non suggerisce di per sé alcuna pratica, ma dal momento che ciascun "defect" origina una modifica al prodotto originario, ha perfettamente senso includerli come attività nel Product Backlog.

Anche i compiti più tecnici, le **indagini** e l'acquisizione di nuove competenze, trovano posto all'interno del backlog in quanto vanno pianificati insieme al team.

Se queste attività non si pianificano è molto facile dimenticarsene o non trovare lo spazio adeguato all'esecuzione.

- **Planning Poker**, è un metodo che si basa sulla dimensione relativa di un requisito in **termini di impegno (effort)** previsto per l'implementazione dello stesso; si gioca a planning poker in gruppo.

L'unità di misura utilizzata per la stima sono gli **Story Point (SP)**, si raggiunge la stima di ciascuna attività sfruttando il principio della “saggezza della folla” e all'unanimità (consensus).

Il risultato, espresso in effort necessario all'implementazione di una funzionalità, è concettualmente molto vicino a un'indicazione di tempo piuttosto che di complessità, dato che ciò che conta per gli stakeholder, e dal punto di vista del business, è il time to market.

Nel caso di Scrum, Product Owner e Scrum Master, a meno che non siano a loro volta sviluppatori, non partecipano attivamente alle stime ma facilitano il processo.

Ogni membro del team di sviluppo ha in mano un mazzo di carte con una sequenza di numeri, la sequenza raccomandata, è ricavata dalla Successione di Fibonacci: 0, 0.5, 1, 2, ..., 100. In alternativa si può anche utilizzare la sequenza originale di Fibonacci o le taglie delle T-Shirt: XS, S, M, L, XL,

In molti mazzi di carte per l'agile poker si trovano carte aggiuntive:

- Il simbolo di infinito (∞), è una chiara indicazione che la User Story andrebbe divisa in parti più piccole per poter essere stimata, ma va da sé che anche punteggi troppo alti potrebbero indicare la stessa necessità;
- Il punto interrogativo, è una richiesta di maggiori informazioni;
- La tazzina di caffè, indica che si è a corto di carburante.

Il primo step è definire una baseline: si sceglie una User Story dal backlog e la si analizza insieme al fine di definirne una stima dell'effort, il Product Owner introduce brevemente l'attività da svolgere, quando tutti hanno un'idea abbastanza chiara su come procedere per la parte di implementazione, il moderatore (Scrum Master) invita i partecipanti a scegliere una carta dal proprio mazzo e a mostrarla tutti insieme.

Il conto alla rovescia, o **showdown**, è un metodo efficace.

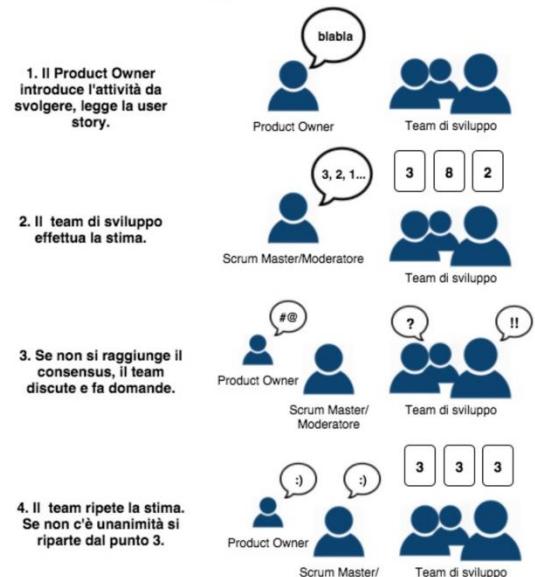
È importante che le carte vengano mostrate tutte insieme e che in fase di discussione non si faccia riferimento ai valori delle carte.

In caso di discrepanze il gruppo riprende la discussione sul requisito cercando di capire i diversi punti di vista e i motivi che hanno portato a valutazioni discordanti. Una volta chiariti i dubbi, ciascuno sviluppatore ripete la stima scegliendo una nuova carta dal mazzo, o quella mostrata in precedenza in caso volesse confermare la propria stima precedente. Si procede così fino a quando il team non raggiunge il consensus, ovvero l'unanimità, e tutti, o quasi, mostrano la stessa carta... il moderatore può negoziare il consenso se la discussione rischia di inoltrarsi fino a notte fonda.

Una volta terminato il primo giro si procede con la stima delle restanti User Story seguendo lo stesso identico procedimento, quando si discute l'effort delle attività successive si ha un riferimento abbastanza preciso.

Si può (anche non ma è consigliata) ridefinire la baseline a ogni Sprint Planning o utilizzare sempre la stessa: in questo caso è importante che tutti gli sviluppatori abbiano ben chiari i dettagli dell'implementazione di quel determinato requisito.

Così facendo si potrà avere un'indicazione più precisa su qual è la velocity del team.

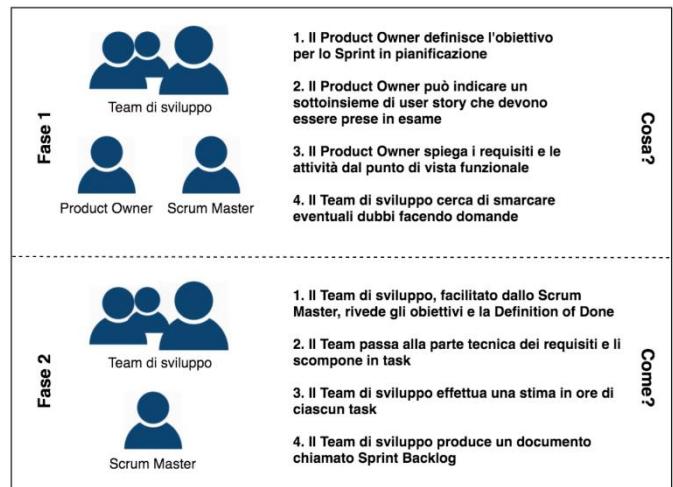


- **Sprint Planning**, è un meeting in cui il team pianifica il lavoro che deve essere svolto e portato a termine durante lo Sprint.

A differenza del Refinement, in cui si ragiona a livello di insieme (User Story), lo Sprint Planning prevede un maggior livello di dettaglio e una suddivisione in task.

Il risultato dello Sprint Planning è lo Sprint Backlog ovvero l'elenco di attività che il team prevede di portare a termine entro la conclusione dello Sprint.

Allo Sprint Planning meeting partecipa tutto il team.



Cosa si include nella pianificazione? Il prerequisito per una buona pianificazione è avere un **Product Backlog** in ordine e priorizzato correttamente dal Product Owner.

- Approccio 1: I requisiti che devono essere inclusi nella pianificazione possono essere **scelti dal team a partire dallo Sprint Goal** (es. Completare il modulo di registrazione) che viene espresso dal Product Owner all'inizio del meeting.

Il team seleziona quindi tutte le User Story che sono applicabili al fine di completare l'obiettivo.

- Approccio 2: Il Product Owner indica esplicitamente un **sottoinsieme di User Story che devono essere incluse** in fase di Sprint Planning e che si trovano in cima al Product Backlog.

Questo approccio può essere dettato da:

- Necessità di business;
- Bisogno di portarsi avanti con attività che dovranno essere completate in iterazioni successive;
- Richiesta specifica da parte del Team di sviluppo;
- Altre ragioni.

Anche se si sceglie questo approccio, è bene ricordare che è il team di sviluppo che decide quanto è possibile aggiungere allo Sprint Backlog.

In ogni caso è sempre bene specificare uno Sprint Goal in modo da **creare un obiettivo** che, se raggiunto, rende lo Sprint ultimato con successo.

Quanto si include nella pianificazione? Di norma si tende a pianificare un numero totale di story point il più vicino possibile alla media della **velocity**, ovvero il dato che rappresenta quanto il team è mediamente in grado di portare a compimento nell'arco di un'iterazione.

Nel caso in cui le User Story selezionate dal team di sviluppo fossero di molto inferiori al dato della velocity o alle ore di sviluppo ideali (IEH), il team può **confrontarsi con il Product Owner** e valutare se procedere con altre attività con priorità alta presenti nel Product Backlog.

Al contrario, se le User Story superano il monte ore a disposizione per lo Sprint, potrebbe rendersi necessaria una **revisione** delle ore di sviluppo ideali (IEH).

Potrebbe anche rendersi necessario aprire un dialogo con il Product Owner e trovare un compromesso, magari **rivedendo lo scope** e ridefinendo quali sono le componenti fondamentali al fine del completamento dello Sprint e il raggiungimento dello Sprint Goal.

Le funzionalità non necessarie potranno così essere pianificate in Sprint successivi.

- **Sprint Backlog**, funziona in maniera analoga al *Product Backlog* e, come già detto, è il risultato dello *Sprint Planning* e comprende un elenco ordinato di attività che il team seleziona e che si impegna di portare a termine entro la fine dello *Sprint*.

All'interno dello Sprint Backlog sono presenti diverse informazioni come (vedi figura come esempio):

- Elenco delle User Story che fanno parte dello Sprint;
- Stima dell'effort per ciascuna User Story, in Story Point;
- Task per ogni User Story;
- Stima in ore per ciascun task;
- Acceptance criteria;
- Link a risorse esterne.

Può anche includere:

- Persona incaricata al task;
- Grafici;
- Sprint Goal.

Goal:	Completare User Account			IEH	192						
ID	Item/Acceptance Criteria	Item details/Task	SP	Owner	Planned Hours	Lun	Mar	Mer	Gio	Ven	Lun
1	As a user I want to login so that I can access the user area	http://bogus.agileway.it/issue/1102	3								
	- Verify that user area is accessible - Verify that welcome is shown - Verify that UI changes on header - Verify that username is "remembered" - Welcome message is displayed	Create login form UI Create login form controller Create login business logic Implement "remember me" check Write Selenium test case Manual test on device		Paolo Paolo Max Franco Giovanni Franco	3 1 4 1 3 2	3 1 2 1 3 2	2 1 2 1 3 2	2 1 1 1 2 2	1 1 1 1 2 2		
2	As a user I want to logout so that I can browse the website anonymously	http://bogus.agileway.it/issue/1103	1								
	- Verify that user area is not accessible - Verify if the cookie is gone - Verify that UI changes on header - User is redirected to "home"	Create logout business logic Implement logout button CSS Manual test on device		Franco Giovanni Max	2 0.5 2	2 0.5 2	2 0.5 2	2 0.5 2	2 0.5 2	0.5 0.5 2	0.5 0.5 2

Le User Story sono inserite nello Sprint Backlog in ordine di **priorità**, le più importanti prima e le meno importanti dopo.

A differenza del Product Backlog, dove le stime sono in Story Point, nello Sprint Backlog le stime di ciascun task sono espresse in **ore di sviluppo**, se un task è più grande di una giornata di lavoro e un buon motivo per dividerlo in più task più piccoli.

Una buona idea è includere lo Sprint Goal da qualche parte, in modo che il team non lo perda di vista

Per lo Sprint Backlog si possono utilizzare tool open

Deve poter essere aggiornato **da più persone**, e infatti responsabilità di ciascuno sviluppatore tenere aggiornato il conto delle ore.

Una giornata lavorativa è di solito composta da 8 ore, ma è anche vero che non tutte e otto le ore vengono utilizzate per sviluppare, molto tempo se ne va in riunioni, oppure per leggere le mail, o per parlare al telefono.

Quando si effettua la pianificazione durante lo Sprint Planning, è buona norma considerare 6 ore di sviluppo ideali (Ideal Engineering Hours) al giorno, lasciando uno spazio di due ore per tutte le altre attività.

La matematica per capire quante ore di sviluppo si hanno a disposizione per ciascuno Sprint è quindi molto semplice.

Le IEH vengono **moltiplicate** per i giorni di lavoro che costituiscono lo Sprint e per ciascuno sviluppatore allocato nell'iterazione.

Nel caso di uno Sprint di due settimane i giorni lavorativi sono 10, e in caso di un team composto da 4 sviluppatori (il Product Owner e lo Scrum Master in genere non sono sviluppatori e non vengono conteggiati) si ha un totale di 240 ore.

$$\text{Ore sviluppo} = \#n_{giorni_sprint} \times IEH \times \#n_{sviluppatori}.$$

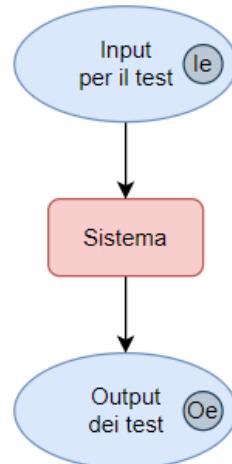
I TEST

Abbiamo già detto che la metodologia agile Scrum si punta, ad ogni Sprint, di fornire delle funzionalità immediatamente testabili.

Tuttavia, queste funzionalità vanno verificate, per questo esistono i Test.

I nostri obiettivi principalmente sono due:

- Dimostrare a sviluppatore e cliente che il SW soddisfa i requisiti:
 - Deve esserci un test per ciascun requisito? **SI**.
 - Deve esserci un test per ciascuna funzionalità? **SI, ANCHE COMBINATI**.
- Scoprire eventuali input o sequenze di input per i quali il comportamento del SW è errato, indesiderato o non conforme alle specifiche:
 - Presenza di *bug*;
 - Possibili crash o interazioni indesiderate con altri sistemi;
 - Calcoli errati o danneggiamento dei dati.



"I test possono dimostrare soltanto la presenza di errori, non la loro assenza"

Edsger Dijkstra

Convalida VS Verifica

- **Convalida:** *"Stiamo realizzando il prodotto corretto?"*
Il SW soddisfa le specifiche e offre le funzionalità richieste.
- **Verifica:** *"Stiamo realizzando il prodotto correttamente?"*
Il SW soddisfa i requisiti funzionali e non funzionali.

Esiste un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga *prima* di quella del software che deve essere sottoposto a test, e che lo sviluppo del software applicativo sia orientato *esclusivamente* all'obiettivo di passare i test automatici precedentemente predisposti.

Tale modello è chiamato **Test Driven Development (TDD)**.

Il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto **ciclo TDD**:

- Nella prima fase, **fase rossa**, il programmatore scrive un test automatico per la nuova funzione da sviluppare, che deve *fallire* in quanto la funzione non è stata ancora realizzata;
- Nella seconda fase, **fase verde**, il programmatore sviluppa la quantità *minima* di codice necessaria per passare il test;
- Nella terza fase, **fase grigia o refactoring**, il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualità.

Supporta la *Continuous Integration (CI)*: allineamento *frequente* (ovvero "molte volte al giorno") dagli ambienti di lavoro degli sviluppatori verso l'ambiente condiviso (*mainline*).

Una buona norma (best practice) che ogni sviluppatore dovrebbe effettuare è quello di eseguire **test di unità** per assicurarsi che la singola unità di sviluppo assolva le sue funzioni e rispetti i requisiti:

i test di unità possono essere sviluppati in maniera indipendente dallo sviluppatore, ma è buona norma avere un prodotto e aderire ad una pratica comune:

Simple use:

- Testare la singola classe con un main che valuti la bontà dei metodi;

Limiti:

- Non può essere ripetuto nel tempo;
- Non segue alcuno standard.

Eric Gamma (**Design Pattern**) e Kent Beck (Extreme Programming) introdussero il progetto **JUnit** (che vedremo in particolar modo), il cui scopo è quello di avere un framework per sviluppare test di unità secondo un semplice modello.

I testi di unità sono una metodologia che permette di verificare il corretto funzionamento di singole unità di codice in determinate condizioni.

Nel caso di Java (e più in generale nella programmazione orientata agli oggetti) si considera come unità di codice un metodo o un gruppo di metodi che implementano una singola funzionalità.

Il test di unità di un'applicazione può essere organizzato in:

- **Test Case**, test che verifica una singola unità di codice;
- **Test Suite**, gruppo di Test Case che verificano funzionalità correlate.

Il test di unità va progettato ed eseguito dallo sviluppatore contemporaneamente allo sviluppo stesso del codice per quella unità.

Se la progettazione dei casi di test è un lavoro duro e difficile, l'esecuzione dei casi di test è un lavoro noioso e gramo. L'automatizzazione dell'esecuzione dei casi di test (**Testing Automation**) porta innumerevoli vantaggi:

- Tempo risparmiato (nell'esecuzione dei test);
- Affidabilità dei test (non c'è rischio di errore umano nell'esecuzione dei test);
- Si può migliorare l'efficacia a scapito dell'efficienza sfruttando il fatto che l'esecuzione è poco costosa;
- Riuso (parziale) dei test a seguito di modifiche nella classe.

I componenti di un test JUnit sono:

- **Classe di test**, che deve estendere (ereditare da) una classe denominata *TestCase* della libreria;
- **Metodi di test**, ognuno ha un nome che inizia con la parola test (in minuscolo), inoltre possono essere realizzati un numero qualsiasi di metodi di test.

Una classe di test, contiene anche:

- Un metodo *setup ()*, che viene eseguito prima dell'esecuzione di ogni test, utile per settare precondizioni comuni a più di un caso di test;
- Un metodo *teardown ()*, che viene eseguito dopo ogni caso di test, utile per resettare le postcondizioni.

La struttura di un metodo di test è suddivisa in quattro parti:

- **Inizializzazione precondizioni**, limitatamente alle precondizioni tipiche del singolo caso di test, le altre potrebbero essere nel setup;
- **Inserimento valori di input**, tramite chiamate a metodi set oppure tramite assegnazione di valori ad attributi pubblici;
- **Codice di test**, esecuzione del metodo da testare con gli eventuali parametri relativi a quel caso di test;
- **Valutazione delle asserzioni**, controllo di espressioni booleani (asserzioni) che devono risultare vere se il test dà esito positivo, ovvero se i dati di uscita e/o le postcondizioni riscontrati sono diversi da quelli attesi.

Ma come si trasforma un test in JUnit?

- **Precondizioni**, tramite il metodo setup vengono eseguite delle operazioni mirate a far diventare vere le precondizioni.
- Al termine del metodo vengono poste delle asserzioni che valutano le precondizioni: se non fossero verificate, i test non saranno eseguiti.
- **Input**, tramite settaggi diretti di attributi oppure chiamate a metodi set (che si suppongono corretti e senza necessità di essere testati).
 - **Test**, esecuzione del metodo da testare con gli eventuali ulteriori parametri di input relativi a quel caso di test.
 - **Output**, controllo di espressioni booleane relative alla coincidenza dei valori di output ottenuti con quelli osservati (asserzioni).
 - **Postcondizioni**, valutazione di espressioni booleane (asserzioni) relative alle postcondizioni.

Si hanno poi le **asserzioni** che, come già detto, effettuano controlli su espressioni booleane, le quali possono essere vere o false.

I risultati attesi sono documentati con delle asserzioni esplicite, non con delle stampe che comunque richiedono dispendiose ispezioni visuali dei risultati. Se l'asserzione è:

- Vera: il test è andato a buon fine;
- Falsa: il test è fallito ed il codice testato non si comporta come atteso, quindi, c'è un errore a tempo dinamico.

Le asserzioni sono utilizzate sia per verificare gli oracoli che le postcondizioni; potrebbero essere utilizzate anche per verificare le precondizioni: se la precondizione fallisce, il test non viene proprio eseguito (e viene riportato come fallito).

Alcune delle asserzioni più note sono:

<code>assertNull ()</code>	Verifica se l'oggetto è nullo
<code>assertTrue ()</code>	Verifica se la condizione è vera
<code>assertFalse</code>	Verifica se la condizione è falsa
<code>assertEquals ()</code>	Verifica se due oggetti sono uguali
<code>fail ()</code>	Fallisce il test

Nota: queste asserzioni sono generali, ciascuna ha molteplici varianti con molteplici firme nei metodi.

Elenco completo: http://junit.sourceforge.net/javadoc_40/org/junit/Assert.html

Es.

```
package calcolatrice;

public class Calcolatrice {
    public Calcolatrice() {};
    public int somma(int a, int b) {
        return a + b;
    }
}
```

```
package calcolatrice;

import junit.framework.TestCase;

public class calcolatriceTest extends TestCase {
    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    //Test method for 'calcolatrice.calcolatrice.somma (int, int)'
    public void testSomma() {
        calcolatrice c = new calcolatrice();
        int a = 5, b = 7;
        int s = c.somma(a,b);
        assertEquals("Somma non corretta!", 12, s);
    }
}
```

In particolare, le asserzioni:

- Nel **Before** corrisponde alla precondizione: se non è verificata il test non viene eseguito;
- Nell'**After** corrisponde alla postcondizione: se non è verificata, il test ha rilevato un malfunzionamento;
- Nel **Test** corrisponde all'oracolo: se non è verificata, il test ha rilevato un malfunzionamento.

Da notare che il test viene eseguito solo se la precondizione è verificata, mentre l'After è eseguito in ogni caso.

```
public class calcolatriceTest {
    private Calcolatrice c;

    @Before
    public void setUp() throws Exception {
        c = new Calcolatrice();
        assertNotNull(c);
    }

    @After
    public void tearDown() throws Exception {
        c = null;
        assertNull(c);
    }

    @Test
    public void testSomma () {
        assertEquals("Somma Sbagliata", 12, c. somma (5,7))
    }
}
```

Se una classe è in associazione/dependency con altre e JUnit rileva l'errore, esso potrebbe dipendere dall'altra classe (se non è stata testata adeguatamente) oppure dall'integrazione.... JUnit è uno strumento che è in grado di risolvere SOLO le problematiche relative al testing di unità, da solo qualche utile indicazione rispetto al testing di integrazione!

Si è dato per scontata la correttezza del codice delle classi di test... se avessimo voluto esserne sicuri al massimo avremmo potuto fare il test di unità delle classi di test stesse (paradossale!)

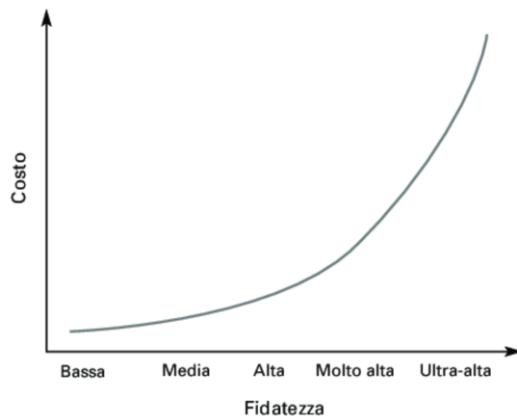
Il codice delle classi di test è comunque estremamente lineare e ripetitivo: la possibilità di sbagliare è ridotta.

SISTEMI FIDATI

La fidatezza di un sistema informatico è una proprietà del sistema che ne riflette l'attendibilità. Attendibilità non è nient'altro che il grado di fiducia che un utente ha nell'aspettarsi che il sistema funzionerà e che il sistema non "fallirà" durante il normale utilizzo.

Non ha senso esprimere la fidatezza numericamente, piuttosto, termini relativi come "non affidabile", "molto affidabile" e "ultra affidabile" possono riflettere il grado di fiducia che potremmo avere in un sistema.

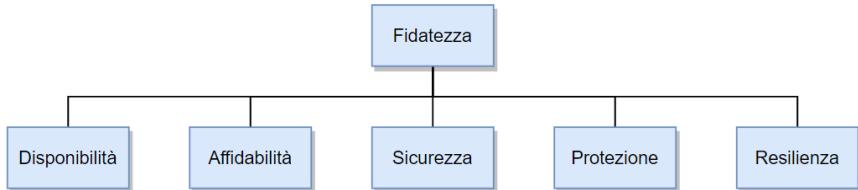
Ovviamente più un sistema è affidabile più tale avrà un costo maggiore:



Affinché anche i processi SW siano fidati questi devono rappresentare cinque caratteristiche:

- **Verificabile**, il processo dovrebbe essere comprensibile a persone diverse da quelle che prendono parte al processo, che possono verificare che gli standard di progettazione sono applicati e che possono dare suggerimenti per il suo miglioramento.
- **Differente**, il processo dovrebbe includere attività di verifica e convalida ridondanti e diverse.
- **Documentabile**, il processo dovrebbe avere un modello di processo definito che stabilisce le attività del processo e la documentazione da produrre durante queste attività.
- **Robusto**, il processo dovrebbe essere in grado di correggere gli errori derivanti dalle sue singole attività.
- **Standardizzato**, dovrebbe essere disponibile una serie completa di standard di sviluppo che definisce come il software deve essere prodotto e documentato.

Tornando ai sistemi, come rappresentato in figura, le dimensioni della fidatezza di un sistema sono principalmente cinque:



- **Disponibilità**, la capacità del sistema di fornire i servizi quando servono.
- **Affidabilità**, la capacità del sistema di fornire i servizi secondo le specifiche.

Sia l'affidabilità che la disponibilità possono utilizzare tre metriche:

1. **Probability Of Failure On Demand, POFOD**, definisce la probabilità che la richiesta di un servizio provochi il fallimento di un sistema;
2. **Rate Of Occurrence Of Failures, ROCOF**, definisce il numero di fallimenti del sistema che probabilmente si verificheranno in un certo periodo di tempo (per esempio, in un'ora) o in un certo numero di esecuzioni del sistema;
3. **Availability, AVAIL**, indica la probabilità che un sistema potrà essere operativo quando viene effettuata la richiesta di un servizio del sistema.

Il termine **fault-tolerant** indica un approccio alla fidatezza in cui i sistemi includono appositi meccanismi per continuare le operazioni, anche dopo che si è verificato un errore del software o un guasto dell'hardware, e il sistema si trova in uno stato di errore. I meccanismi fault-tolerant identificano e correggono questo stato di errore del sistema, in modo tale che, quando si verifica un guasto, non si abbia un fallimento del sistema. Questi meccanismi sono richiesti nei sistemi a sicurezza o a protezione critica, e quando il sistema non può passare a uno stato sicuro dopo che è stato rilevato un errore.

- **Sicurezza**, la capacità del sistema di operare senza provocare danni ingenti.

Cos'è innanzitutto un rischio, un rischio può presentarsi in varie forme le quali sono:

- **Incidente**, o contrattempo, un evento o una sequenza di eventi non pianificati che possono portare morte o infortuni alle persone o danni alle cose e all'ambiente;
- **Danno**, la misura delle perdite risultanti da un incidente, le perdite variano dalla morte di diverse persone, alle ferite lievi o ai danni alle cose;
- **Pericolo**, una condizione che può causare un incidente;
- **Probabilità del pericolo**, la probabilità che si verifichino eventi che possono creare situazioni pericolose. I valori di probabilità tendono a essere arbitrari, ma variano da "probabile" a "non plausibile";
- **Gravità del pericolo**, una stima del danno peggiore che potrebbe risultare da un particolare pericolo. La gravità può variare da "catastrofica", quando molte persone perdono la vita, a "lieve", quando si hanno solo danni lievi. Se è possibile la morte di un singolo individuo, la gravità del pericolo è "molto alta";
- **Rischio**, una misura della probabilità che il sistema provochi un incidente. Il rischio è stimato considerando la probabilità e la gravità del pericolo, e la probabilità che il pericolo causi un incidente.

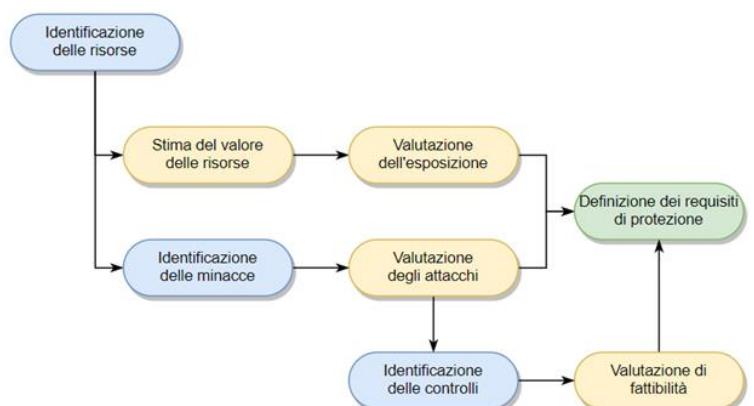
Come si è appena visto il garanzia della sicurezza del sistema è assolutamente necessario, per questo si hanno dei processi per la garanzia di quest'ultima:

1. **Analisi e monitoraggio dei rischi**, i rischi vengono tracciati partendo da un'analisi preliminare fino al test e alla convalida del sistema;
2. **Revisioni della sicurezza**, sono attività che si svolgono durante tutto il processo di sviluppo.
3. **Certificazione della sicurezza**, viene certificata la sicurezza dei componenti critici. Questa attività riguarda un gruppo esterno al team di sviluppo del sistema che esamina le prove disponibili e decide se il sistema o un suo componente può essere considerato sicuro prima che sia consegnato agli utenti.

- **Protezione**, la capacità del sistema di proteggersi contro intrusioni accidentali o deliberate, i quali sono:
 - **Risorsa**, qualcosa che merita di essere protetta. La risorsa può essere lo stesso sistema software o i dati utilizzati dal sistema;
 - **Attacco**, un hacker sfrutta una vulnerabilità del sistema con lo scopo di provocare qualche danno alle risorse del sistema. Gli attacchi possono essere portati dall'esterno del sistema (attacchi esterni) o da utenti autorizzati del sistema (attacchi interni);
 - **Controllo**, una misura protettiva che riduce la vulnerabilità del sistema. La crittografia può essere un esempio di controllo che riduce le vulnerabilità di un sistema che ha un controllo degli accessi debole;
 - **Esposizione**, possibile perdita o danno a un sistema di calcolo. Il termine può riferirsi alla perdita o al danneggiamento dei dati oppure al tempo e alle energie spese per ripristinare le funzionalità del sistema dopo una violazione della protezione;
 - **Minaccia**, circostanza che ha il potenziale di causare perdite o danni. Il termine identifica una vulnerabilità del sistema che è sottoposto a un attacco;
 - **Vulnerabilità**, un punto debole in un sistema informatico che può essere sfruttato per arrecare perdite o danni al sistema.

Per dire se un sistema è protetto, prima di tutto si fa un processo di valutazione preliminare, in base allo schema qui presente, per difendersi dalle seguenti minacce:

- **Minacce di intercettazione**, un hacker guadagna l'accesso alle risorse del sistema;
- **Minacce di interruzione**, un hacker rende indisponibile una parte del sistema;
- **Minacce di modifica**, un hacker danneggia una risorsa del sistema;
- **Minacce di falsificazione**, un hacker inserisce informazioni false nel sistema.



Dopo la prima valutazione si passa ai requisiti:

1. Requisiti di identificazione, specificano se un sistema deve identificare gli utenti prima di interagire;
2. Requisiti di autenticazione, specificano come identificare gli utenti;
3. Requisiti di autorizzazione, specificano i privilegi e le autorizzazioni di accesso degli utenti identificati;
4. Requisiti di immunità, specificano come un sistema dovrebbe proteggersi da virus, worm e minacce simili;
5. Requisiti di integrità, specificano come evitare di danneggiare i dati;
6. Requisiti di identificazione delle intrusioni, specificano quali meccanismi utilizzare per scoprire gli attacchi portati al sistema;
7. Requisiti di non-ripudio, specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento;
8. Requisiti di riservatezza specificano come deve essere mantenuta la riservatezza delle informazioni;
9. Requisiti di controllo della protezione specificano, come può essere controllato l'utilizzo di un sistema;
10. Requisiti di protezione della manutenzione del sistema, specificano come un'applicazione può impedire che siano fatte delle modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione.

Una tecnica nota e sicura è l'**Architettura di Protezione a Più Strati**, nella quale si dividono in più strati le varie risorse e vulnerabilità con le relative tecniche di protezione.

- **Resilienza**, la capacità del sistema di resistere e riprendersi dopo un evento dannoso.

Le attività di resilienza correlate nel rilevamento e nel recupero da problemi di sistema sono quattro:

1. **Riconoscimento**, il sistema o i suoi operatori dovrebbero essere in grado di riconoscere i sintomi di un problema che potrebbe portare al malfunzionamento del sistema.

Idealmente, questo riconoscimento dovrebbe essere possibile prima che si verifichi l'errore.

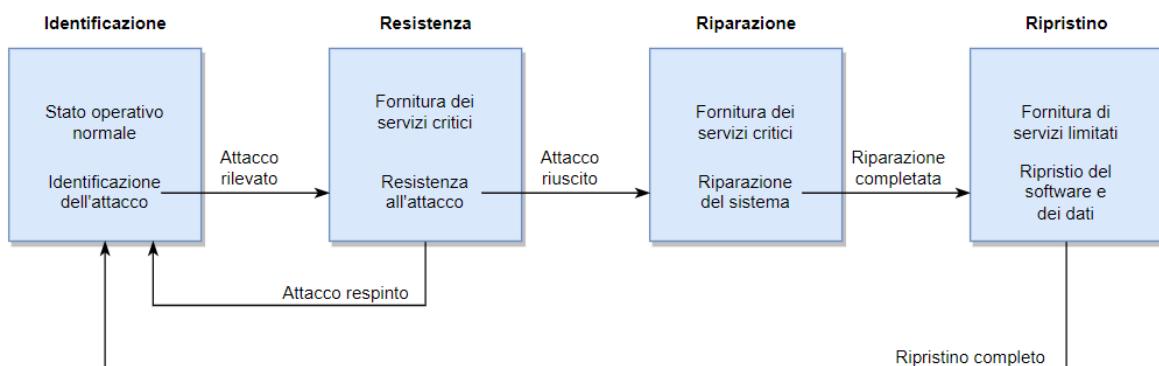
2. **Resistenza**, se vengono rilevati i sintomi di un problema o segni di un attacco informatico presto, allora possono essere invocate strategie di resistenza che riducono la probabilità che il sistema fallirà.

Queste strategie di resistenza possono concentrarsi sull'isolamento di parti critiche del sistema in modo che non siano interessati da problemi altrove. La resistenza include:

- **Resistenza proattiva**, in cui le difese sono incluse in un sistema per intrappolare i problemi e
- **Resistenza reattiva**, in cui vengono intraprese azioni quando viene scoperto un problema.

3. **Ripristino**, se si verifica un guasto, lo scopo dell'attività di ripristino è quello di assicurarlo i servizi di sistema critici vengono ripristinati rapidamente in modo che gli utenti del sistema non siano gravemente colpiti dal guasto.

4. **Ripristino**, in questa attività finale, vengono ripristinati tutti i servizi di sistema e il normale funzionamento del sistema può continuare.



Una delle buone pratiche della resilienza è la monitorizzazione degli eventi in base a tutto quello che circonda il sistema, in modo da prevederli e resistergli.

