SOFTWARE DESIGN

Andare a definire il concetto di qualità per un software non è proprio semplice, perché pe un software non è possibile definire degli obiettivi precisi, questi dipendono da cosa ha bisogno chi richiede il software
La qualità viene definita sulla base di alcune caratteristiche si possono attribuire al codice come:

- Funzionalità
- Reliabulity
- Usabilità
- Efficienza
- Manutenibilità
- Portabilità
- Component replacebility

Partendo da queste caratteristiche si definiscono delle metriche per misurare queste, e sulla base di queste è definita la qualità di un software. È necesarrio capire quindi come aggregare queste misure.

La **progettazione del software** (il processo) è la costruzione di astrazioni di dati e calcolo e l'organizzazione di queste astrazioni in un'applicazione software funzionante - Martin P. Robillard
Quindi questo vuol dire andare a definire che struttura dati usiamo per il contesto, come gestiamo le eccezione ecc. Si tratta di un processo euristico e iterativo guidato dalla conoscenza della materia e che porta a scrivere un codice che sia progettualmente pulito in modo da avere meno errori ed è possibile modificare con più semplicità.
L'obiettivo del software design è quello di andare a definire in modo **chiaro** il contesto che andiamo a modellare, definendo i requisiti e constraint del dominio.
Il design può essere inteso come un insieme di decisioni per modellare il problema e che ci permettono di **catturare la conoscenza del dominio**. Per poter fare questo le soluzione sono descritte tramite dei modelli che vengono chiamati **design patter** e che furono introdotti dalla GoF, i quattro cristi che hanno dato via e evoluto il movimento agile.
Crearono un catalogo di 23 design patter, dando soluzioni a problemi ricorrenti.
I pattern non sono implementazioni concrete, ma "schemi che defiscono un certo modo d struttura il codice per poter rendere il codice pulito e comprensibile.

DESIGN PATTER

Un design pattern ha 4 componenti principali:

- **nome del pattern**, è il modo con cui in una o due parole possiamo riferirci ad un problema, la sua soluzione e le conseguenze della sua applicazione.
- **problema** descrive quando è possibile applicare il pattern. Esso cioè esplicita il contesto.
  contesto insieme di condizioni e situazioni in cui un pattern è applicabile
- **soluzione**, descrive gli elementi che fanno parte del progetto, le loro relazioni, responsabilità e collaborazioni. Essa non descrive una specifica implementazione in quanto un pattern deve poter essere applicabile in diverse situazioni concrete.
- **conseguenze**, sono il risultato di trade-offs derivanti dall'applicazione del pattern. Esse sono di importanza critica nella valutazione di alternative di progetto e per la comprensione di costi e benefici relative all'applicazione del pattern.

I pattern sono classificai sulla base del

- **purpose**, cosa fa
- **scope**, se si applica alle classi o agli oggetti

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

Creational Patterns
Gestiscono la **creazione degli oggetti**, cercando di astrarre il processo di istanziazione. Lo scopo è rendere il sistema indipendente dalla modalità di creazione, composizione e rappresentazione degli oggetti.

Structural Patterns
Si occupano della **composizione delle classi e degli oggetti**, facilitando la creazione di strutture flessibili ed efficienti.

Behavioral Patterns
Si concentrano sugli **algoritmi e sulla comunicazione** tra oggetti, migliorando l'interazione e la responsabilità tra componenti.

Principi della OOP alla Base dei Design Pattern
I principali concetti su cui si basano i design pattern sono quattro: **incapsulamento, astrazione, ereditarietà e polimorfismo**.

Incapsulamento il principio secondo cui **i dettagli interni di un oggetto devono essere nascosti** (**Information Hiding**) al mondo esterno, esponendo solo ciò che è strettamente necessario attraverso un'interfaccia pubblica. Questo consente di proteggere lo stato interno dell'oggetto da modifiche non autorizzate e di mantenere sotto controllo la complessità del sistema.

Astrazione permette di **modellare concetti complessi nascondendone i dettagli non rilevanti**, concentrandosi solo sulle caratteristiche fondamentali. Questo principio guida la progettazione di interfacce e classi astratte, che permettono di definire comportamenti generici che possono poi essere specializzati da classi concrete.

Ereditarietà consente di **riutilizzare il codice** definito in una classe base in una o più classi derivate. Questo facilita la costruzione di gerarchie di classi in cui le specializzazioni si basano su un comportamento generico già definito. Nei design pattern, questo principio è spesso usato per **delegare la responsabilità di alcune operazioni a sottoclassi**.

Polimorfismo consente di **trattare oggetti di classi diverse come se appartenessero alla stessa interfaccia**, sfruttando la possibilità di invocare metodi in modo dinamico a seconda del tipo reale dell'oggetto.

I Principi SOLID
Oltre ai principi fondamentali dell'OOP, molti design pattern si ispirano ai **principi SOLID**, un insieme di linee guida introdotte da Robert C. Martin (Uncle Bob) per facilitare la progettazione di software mantenibile ed estensibile:

1. **Single Responsibility Principle (SRP)** – Ogni classe o metodo dovrebbe avere **una sola responsabilità** o motivo di cambiamento.
2. **Open/Closed Principle (OCP)** – Le entità software dovrebbero essere **aperte all'estensione ma chiuse alla modifica**.
   Le classi sono **aperte** alle estensioni tramite l'implementazione di interfacce o l'ereditarietà, ma **chiuse** alle modifiche, poiché il comportamento esistente rimane inalterato. Favorisce la **modularità** del codice e la **facilità di estensione del sistema**. Possiamo aggiungere

nuovi tipi di veicoli senza dover modificare il codice esistente, garantendo una maggiore stabilità e riducendo il rischio di introdurre nuovi bug.

3. **Liskov Substitution Principle (LSP)** – Le classi derivate devono poter sostituire le classi base **senza alterare il comportamento previsto**. Questo principio è fondamentale per garantire che l'uso del polimorfismo sia corretto e prevedibile.
4. **Interface Segregation Principle (ISP)** – È meglio avere **interfacce specifiche e limitate** piuttosto che una singola interfaccia generica e pesante.
5. **Dependency Inversion Principle (DIP)** – Le classi dovrebbero **dipendere da astrazioni, non da classi concrete** riducendo l'accoppiamento tra le componenti del sistema.

ANTIPATTER

Come esistono le buone idee, esistono anche le idee di merda… i bad smells, o **antipattern**, soluzioni che sembrano ottime, ma fanno cagare.

ENCAPSULATION

Il **PRIMITIVE OBSESSION** è un antipattern in cui si usa eccessivamente tipi primitivi (come stringhe, interi, booleani, ecc.) per rappresentare concetti complessi o entità del dominio del problema, invece di creare tipi o classi specifiche per quei concetti.
Questo è un problema perché ciò rende il codice meno chiaro e meno espressivo, aumenta il rischio di errori perché non c'è nessuna validazione o comportamento incapsulato o controllo sui valori che può assumere.
Quindi quando dobbiamo rappresentare oggetti della realtà nel codice conviene creare un classe che li modelli per avere più controllo su di esso. Creando una classe incapsuliamo anche la logica e di funzionamento con tutte le informazioni ad essere relative.

L'**Escaping References** accade quando un oggetto "lascia scappare" i riferimenti ai suoi dati interni, cioè quando dà accesso diretto a qualcosa che dovrebbe invece tenere nascosto e sotto controllo.
Per esempio, immagina che una classe abbia una lista di cose importanti, tipo le transazioni di un conto bancario. Se questa lista viene restituita direttamente quando qualcuno la chiede, allora quella persona può cambiarla a piacimento, anche senza passare dai metodi sicuri della classe.
Questo è un problema perché così perdi il controllo su cosa succede dentro l'oggetto. La lista potrebbe essere modificata in modo sbagliato, causando errori o dati incoerenti.

Non basta dichiarare una variabile di istanza privata per evitare l'escape reference, perché anche usando i getter, potremmo degradare l'incapsulamento. Avere una classe a cui si accede solo tramite i getter e i setter è una debolezza delle progettazione, noto come **INAPPROPRIATE INTIMACY**, perché le classi passano troppo tempo a esplorare le parti private delle altre. BAH! Sono gli oggetti che dovrebbero interagire tra loro usando i metodi che hanno a disposizione.

La soluzione? Non dare mai direttamente i tuoi dati "interni", ma invece restituisci una copia di quei dati. In questo modo chi riceve la lista può modificarla quanto vuole, ma senza influenzare quella originale dentro l'oggetto. Oppure, puoi usare metodi specifici per far fare le modifiche solo in modo controllato. Così, mantenere tutto sotto controllo diventa più facile e il codice rimane più robusto.

Un'altra soluzione è l'**immutabilità** in modo da rendere le classi immodificabili nel loro stato interno, se non tramite i suoi metodi.

Design by Contract, è un approccio in cui ogni componente software è visto come un **contratto** e stabilisce cosa ogni parte deve garantire e cosa si aspetta in cambio.Un contratto, nel contesto del software, si basa su tre elementi:

- **Precondizioni:** sono le condizioni che devono essere vere prima che un'operazione venga eseguita. Il chiamante ha la responsabilità di assicurarsi che queste condizioni siano soddisfatte. Se le precondizioni non sono rispettate, il comportamento del metodo è indefinito.
- **Postcondizioni:** sono le condizioni che il metodo garantisce al termine della sua esecuzione, a patto che le precondizioni fossero vere. Se il metodo ha completato correttamente, allora queste condizioni devono sempre risultare vere.
- **Invarianti:** sono condizioni che devono rimanere sempre vere durante l'intera vita di un oggetto. Esse definiscono lo stato valido dell'oggetto e aiutano a mantenere la coerenza interna indipendentemente dalle operazioni eseguite.

TYPES E INTERFACE

Sappiamo che cos'è un interfaccia nella OOP. Un particolarità di queste sottovalutata è la possibilità di gestire il polimorfismo e anche l'ereditarietà multipla che in java non è permessa.
Nelle classi che progettiamo il fatto di mettere i metodi direttamente nella classe, causa il problema dell'accoppiamento con la sua definizione. Un principio importante di progettazione è il **disaccoppiamento del comportamento** dalla definizione della classe per rendere il tutto più indipendente dal tipo di servizio che deve essere implementato. L'interfaccia permette di definire un tipo, obv con la caratteristica che l'interfaccia non si può instanziare, ma le sue classi concrete sì. La relazione che si viene a creare fra l'interfaccia e la sua classe concreta, è una relazione **polimorfica di sottotipo**, quindi è possibile assegnare ad una variabile definita come tipo interfaccia, una qualunque classe concreta che implementa l'interfaccia.
Il polimorfismo permette:

- l'**accoppiamento debole**, perché il codice che utilizza un insieme di metodi vincolato ad una specifica implementazione, si definisce a runtime, quindi si possono chiamare solo i metodi in essa definiti
- l'**estensibilità** perché possiamo facilmente aggiungere nuove implementazioni

L'interfaccia può servire anche a definire un comportamento per le classi che le implementano. Tipo Comparable<T>. Per rispettare il principio ISP le interfacce devono implementare pochi metodi, e in questo cosa per definire i comportamenti solo uno.
Se non usassimo una interfaccia dovremmo modificare in modo pesante il codice, questo antipattern è noto come **SWITCH STATEMENT**. La soluzione è quella di creare una classe che implementa l'interfaccia e un certo comportamento. Queste sono dette anche **function object** che solo classi con solo funzioni. Tutto ciò può essere evitato con:

- **nested class**, si definisce la classe nello scope di un'altra
- **anonymous class**, una classe senza nome che viene definita e istanziata direttamente nel punto in cui serve
- **lambda expression**, una forma compatta di scrivere funzioni anonime, usata per implementare **interfacce funzionali**, cioè interfacce con un solo metodo astratto.

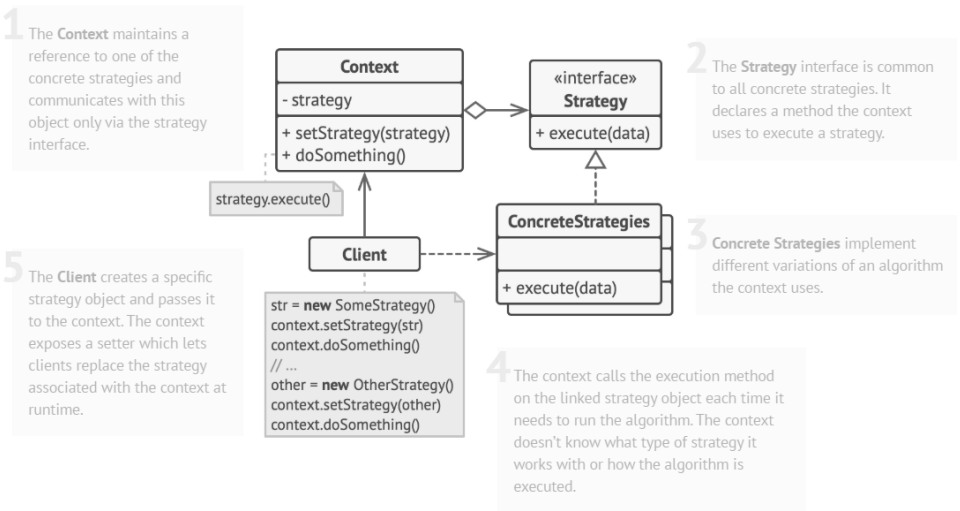
Queste possono essere passate ad altre funzione, che oltre a prendere dati prendono funzioni, comportamenti diversi che vengono applicati sui dati, in modo che venga eseguita a un certo punto.

Questo viola l'incapsulamento perché usano il costruttore o gli altri modi mostriamo lo stato della classe. Un modo per rafforzarlo è lo **STATIC FACTORY METHOD** che nasconde la logica di creazione. Consiste nell'implementare un metodo statico che restituisce un'istanza di quella classe.

- I metodi factory statici possono restituire lo stesso tipo che implementa il metodo, un sottotipo e anche primitive, offrendo una gamma più flessibile di tipi restituiti
- I metodi factory statici possono essere metodi a istanziazione controllata, con il pattern Singleton, lo Strategy.
- Il factory method possono essre usati per creare oggetti **immutabili**, impedendo la modifica dello stato dopo la creazione. Questo rinforza ulteriormente l'incapsulamento.

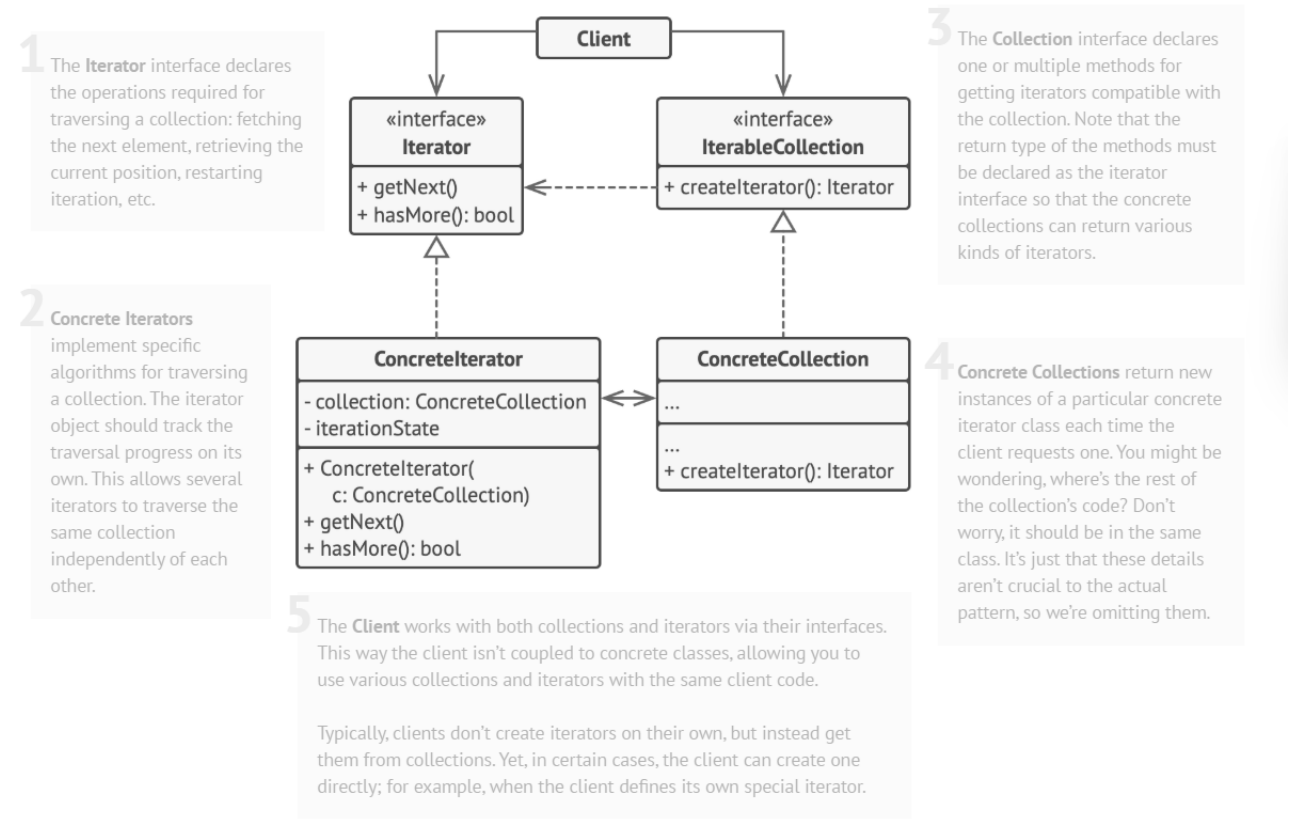
STRATEGY PATTER

Il fatto che alle objects function possiamo passare comportamenti simili, ma diversi, è possibile grazie allo **strategy pattern** che permette di fornire una famiglia di algoritmi che implementa la stessa interfaccia, ma hanno un'implementazione diversa. La scelta dell'algoritmo dipende dal contesto che viene passato del client, quindi è determinato a runtime.



ITERATOR PATTERN

Un requisito di design per la progettazione delle strutture dati è l'accesso ai dati che contiene, ma senza violare il principio dell'information hiding. Una soluzione migliore di ritornare una copia della collezione è quella di fornire un metodo di accesso a questi dati, un **iteratore** che nasconde la struttura interna di come sono memorizzati i dati. Questo fortifica l'incapsulamente, tranne nel caso ci siamo il metodo remove(). L'**iterator pattern** fa proprio questo.



DEPENDENCY INJECTION

Quando creiamo delle classi questi possono avere bisogno di oggetti di altre classi, quindi nel costruttore creiamo un'istanza dell'oggetto che ci server per completare la classe. SI viene a creare un **dipendenza**. (Pure le classi si drogano...bha). Questo è un problema perché lo leghiamo alla classe che viene iniettata e se questa cambia comportamento, definizione ecc, dobbiamo modificare ogni classe che ha una sua istanza. E pensa che palle.
Una soluzione può essere quella di usare le lambda, ma ci sono gli stessi problemi. Allora dobbiamo cercare di disaccoppiare le classi e lo si può fare **iniettando** la dipendenza nella classe. (Ora la droghiamo noi). Lo si può fare facendo stabile al client cosa usare. Ci sono anche altre soluzioni [VEDI DOPO]

OBJECT STATE

Si può avere una visione duplice dello stato di un programma.

> **statica**, che avviene a compile-time, dove il codice viene analizzato e tradotto in un formato eseguibile. In questa fase, il compilatore conosce le dichiarazioni delle classi, i metodi disponibili, le gerarchie di ereditarietà e i tipi associati alle variabili.
> **dinamica**, che avviene a run-time, in cui il comportamento effettivo del programma dipende da molti fattori che non sono visibili durante la compilazione, come i dati di input, le condizioni che si verificano nei vari rami del codice, lo stato interno degli oggetti e, soprattutto, il tipo reale degli oggetti creati in memoria.

Queste due non è detto che coincidano sempre. Questo è dovuto al fatto al polimorfismo. Possiamo usare per rappresentare i possibili stati di un oggetto con i sequence, state e object diagrams.

Importante è definire l'**object state** che si può avere di un oggetto. Possiamo dire che un oggetto avrà uno **stato concreto**, cioè i valori che possono assumere le variabili che definiscono una classa. Bisogna considerare anche il suo **state space**, che rappresenta tutti i possibili valori che può assumere. Ovviamente non sempre è possibile rappresentarli tutti allora ci è utile definire un sottoinsieme di queste, lo **abstract space**. Questo deve essere significato e deve cogliere gli aspetti importanti di come lo stato cambia e quali sono i suoi possibili passaggi di stato.
Lo state diagram può aiutare a fare questo e ci mostra il ciclo di vita di un oggetto.

> **PRINCIPIO DI DESIGN** evitare cicli di vita troppo complessi e minimizzare
> Lo spazio degli statti un oggetto

Alcuni stati nello spazio degli stati potrebbero essere una conseguenza di come un oggetto è progettato o implementato, senza che ci sia utilità per un oggetto in quello stato in un dato sistema software. In questi casi, eliminare alcuni stati dal ciclo di vita di un oggetto può sembrare una buona idea, ma se se un giorno ne avessimo bisogno? Si cade nell'antipattern della **SPECULATIVE GENERALITY**, cioè ci preoccupiamo del futuro, ma non serve, se serviranno verranno aggiunte in seguito, mo pensa al presente.
Sbagliato è anche cachare informazioni nelle variabili di istanza, ma in realtà dipende da situazione a situazione. Se usiamo un metodo per accedere a quello stato, ma si sa che quella informazione non verrà mai modificato, allora la si può

storare nella classe. Se può essere modificata nel corso del ciclo di vita dell'oggetto allora teniamo il metodo per accedere a quello stato. È un trade-off che dobbiamo tenere conto, se risparmiare un po' di tempo a costo di un po' di memoria extra, o viceversa. Il principio generale è che le informazioni non dovrebbero essere memorizzate in un oggetto a meno che non contribuiscano in modo univoco al valore intrinseco rappresentato dall'oggetto, questo antipattern è il **TEMPORARY FIELD**.
I campi temporanei e tutto il codice che li gestisce possono essere inseriti in una classe separata tramite **Extract Class**. In altre parole, si crea una classe che ha quella informazione e si usa un metodo per poter accedere a quelle informazioni.

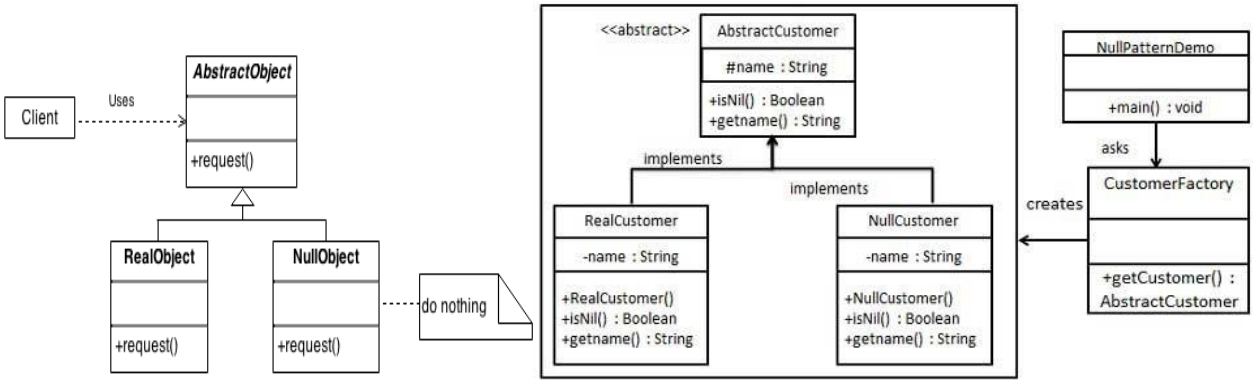
Questo metodo di refactoring contribuirà a mantenere l'aderenza al principio di responsabilità unica.

Introdurre **NULL OBJECT** e integrarlo al posto del codice condizionale utilizzato per verificare l'esistenza dei valori dei campi temporanei. Ma prima parliamo della **nullability**, la possibilità di assegnare valori null alle variabili per indicare l'assenza di valore. Prima non si poteva, fu introdotto da Tony Hoare, con l'obiettivo di rendere i riferimenti più sicuro. Ma è stata una idea del cazzo, perché ha introdotto altri problemi come, NullPointerException, SegmentationFault, difficoltà nella verifica ecc...
Per evitare di ampliare lo spazio di stato di un oggetto con riferimenti nulli, una buona pratica consigliata è progettare le classi in modo che i riferimenti nulli non vengano utilizzati. Ci sono vari modi per farlo.

- Assegnare i valori nel costruttore
- Verificare se i valori sono nulli e nel caso lanciare exception
- Design by contract
- modi migliori...

Gli **optional type**, sono delle classi wrapper che contengono un istanza del tipo che wrappiamo oppure non lo contengono. Come il gatto di Schrödinger, non sai cosa contiene finchè non unwrappi. Quindi il programmatore vedendo un costrutto del genere fa più attenzione a fare controlli ecc.
Un problema di questa soluzione e che perdiamo il tipo delle variabili, quindi dovremmo creare dei metodi per ottenere i tipi effettivi.

Ritornando al **NULL OBJECT**, permette di risolvere questo problema e la nullability, prevedendo un oggetto che rappresenta il valore null per quella classe. Anche questo si basa sul polimorfismo.
Consiste nell'aggiungere un metodo per determinare su un oggetto è nullo o meno e a secondo della condizione restituire o null o no. Oppure implementare una factory che nasconde tutto il funzionamento



Per minimizzare ancora di più lo spazio degli stati, possiamo oltre ad usare una classe Enumerate, stabilire anche un vincolo sul numero di volte che una variabile può essere modificata. In java si può usare final, per dire che il campo una volta assegnato non può essere più modificato e se si provasse a farlo avremmo un errore a compilation time. Da notare che la variabile final, contine un riferimento, ed è questo che non può essere modificato, ma lo stato dell'oggetto riferito sì. I campi final però non rendono immutabili gli oggetti referenziati. A volte si usa la parola chiave final per chiarire che una variabile non deve essere modificata dopo la sua inizializzazione. Questo può aiutare a rendere il codice più leggibile, specialmente quando ci si trova in contesti più lunghi o complessi dove il rischio di errore o confusione è maggiore. Ma questo non dovrebbe essere mai fatto per via dell'antipattern **LONG METHOD** che stabilisce che i metodi lunghi devono essere evitati.

Tre concetti da tenere a mente quando si progettano i cicli di vita degli oggetti sono quelli di identità, uguaglianza e unicità.
L'**identità** si riferisce al fatto che ci si riferisce a un oggetto specifico, anche se questo non si trova in una variabile, cioè l'identità di un oggetto si riferisce solitamente alla sua posizione di memoria o riferimento/puntatore.
L'**uguaglianza** tra due oggetti deve essere definita dal programmatore perché il significato dell'uguaglianza non può sempre essere dedotto dalla progettazione della classe dell'oggetto
L'**unicità**, il fatto di avere una solo istanziazione per quella classe. Se si può garantire che gli oggetti di una classe siano unici, non è più necessario definire l'uguaglianza, perché in questo caso specifico, l'uguaglianza diventa equivalente all'identità e possiamo confrontare gli oggetti utilizzando l'operatore

==. Garanzie di unicità sono quasi impossibili da ottenere in Java a causa di meccanismi come la **metaprogrammazione** e la **serializzazione**.

FLYWEIGHT PATTER

Il Flyweight consiste nel condividere parti comuni dello stato tra oggetti, evitando di duplicare dati che possono essere riutilizzati. Gli oggetti flyweight sono spesso **immutabili** e leggeri, e servono a rappresentare solo lo stato intrinseco, mentre lo stato estrinseco viene passato dall'esterno quando serve. Invece di creare milioni di oggetti simili, si condividono quelli uguali per risparmiare memoria.
Per gestire la creazione di oggetti di una classe, si crea una classe flyweight. Le cui istanze sono chiamate oggetti flyweight. I tre componenti principali necessari per realizzare questo vincolo sono:

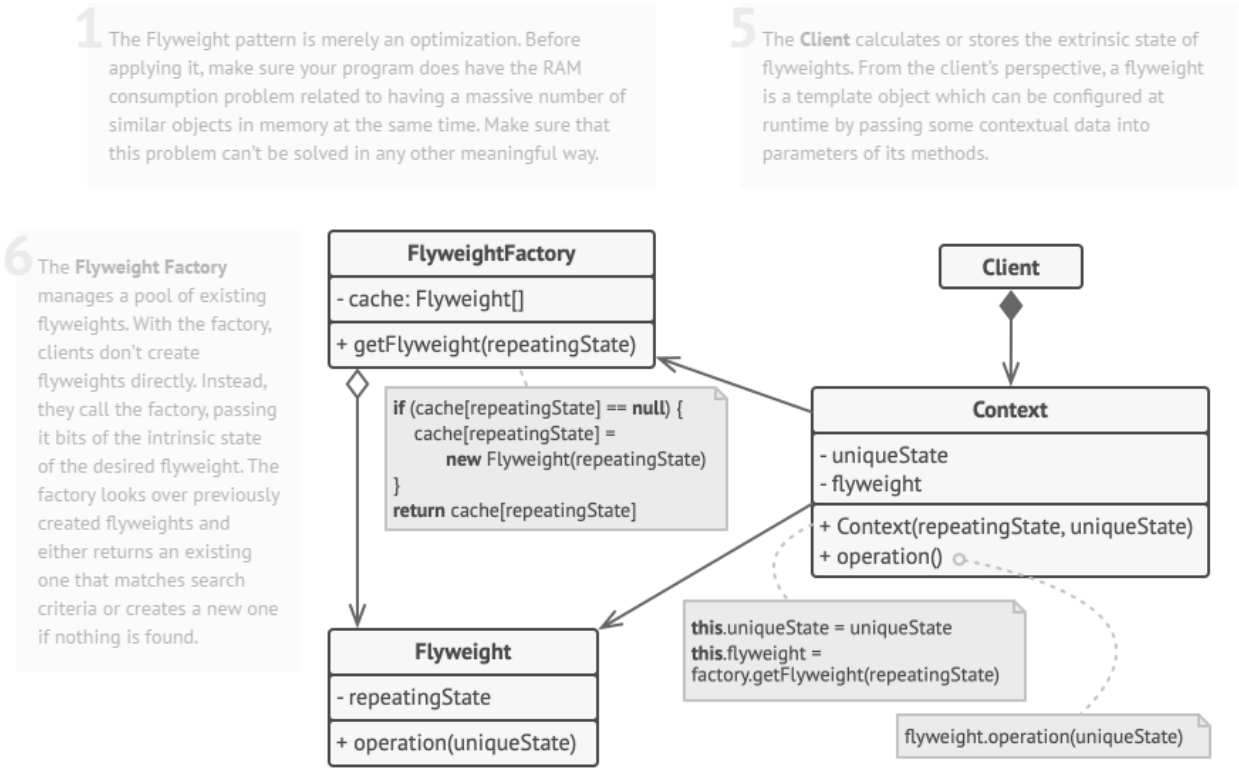
1. Un costruttore privato per la classe flyweight, in modo che i client non possano controllare la creazione di oggetti della classe;
2. Un archivio statico di oggetti flyweight che conserva una collezione di oggetti flyweight;
3. Un metodo di accesso statico che restituisce l'oggetto flyweight univoco corrispondente a una chiave di identificazione. Il metodo di accesso in genere verifica se l'oggetto flyweight richiesto esiste già nell'archivio, lo crea se non esiste già e restituisce l'oggetto univoco.

Quando lo **spazio degli stati** di una variabile è **limitato e ben definito**, è possibile inizializzare gli oggetti tramite un **blocco statico**, che viene eseguito al primo caricamento della classe. In alternativa, si può usare una **struttura dati statica**. Questo approccio è **statico** perché l'archivio dei flyweight è condiviso a livello di classe, e non ha senso dover accedere a un oggetto tramite un'altra istanza dello stesso tipo. Questa combinazione di archivio statico e metodo di accesso viene spesso chiamata **flyweight factory**.
Per facilitare l'accesso ai vari oggetti flyweight, si può implementare un **metodo factory** che gestisce un pool di istanze esistenti. Il metodo accetta lo **stato intrinseco** richiesto, cerca un oggetto corrispondente nel pool e, se lo trova, lo restituisce; altrimenti, crea una nuova istanza, la aggiunge al pool e poi la restituisce.Questo metodo factory può essere collocato in vari posti, a seconda del contesto:

- In un **contenitore di flyweight** dedicato
- In una **classe factory** separata
- Oppure come **metodo statico** all'interno della stessa classe flyweight

Un aspetto importante da considerare nell'implementazione del pattern FLYWEIGHT è se pre-inizializzare l'archivio flyweight o se farlo in modo lazy, creando oggetti man mano che vengono richiesti tramite il metodo di accesso.

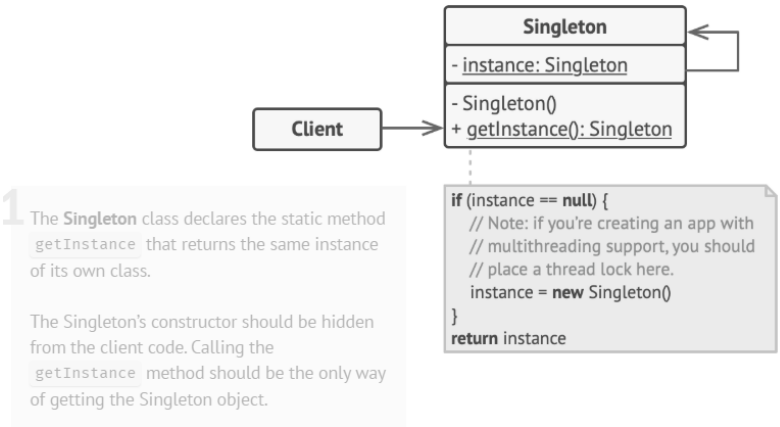
Dipende dal contesto. Il pattern FLYWEIGHT è utile quando si devono gestire oggetti flyweight immutabili, anche se ci possono essere errori se implementato in modo scorretto.



SINGLETON PATTERN

Il **Singleton** viene utilizzato quando si desidera che una determinata classe abbia una sola istanza durante tutta l'esecuzione del programma, e che tale istanza sia facilmente accessibile da qualsiasi punto del codice.

1. Un costruttore privato per la classe singleton, in modo che i client non possano creare più oggetti;
2. Una variabile globale per contenere un riferimento alla singola istanza dell'oggetto singleton.
3. Un metodo di accesso, solitamente chiamato instance(), che restituisce il singleton in istanza. Il metodo di accesso è facoltativo, poiché è anche possibile implementare il pattern dichiarando l'istanza globale come costante pubblica.



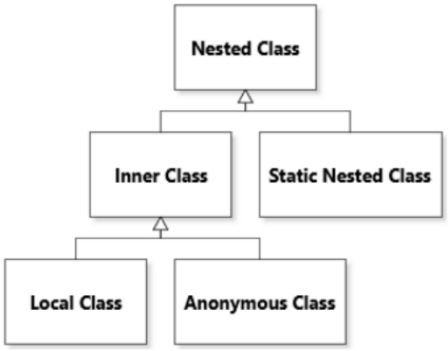
Il pattern SINGLETON differisce da FLYWEIGHT in quanto cerca di garantire che esista una singola istanza di una classe, anziché istanze uniche di una classe. Gli oggetti Singleton sono tipicamente con stato e mutabili, mentre gli oggetti flyweight sono immutabili.
Un errore tipico nell'implementazione del pattern SINGLETON è quello di memorizzare un riferimento a un'istanza della classe in un campo statico chiamato INSTANCE o qualcosa di simile, senza prestare la dovuta attenzione a impedire al codice client di creare autonomamente nuovi oggetti. In questo caso, l'uso del nome Singleton è pericolosamente fuorviante, poiché gli utenti del codice potrebbero fare affidamento sul fatto che la classe produca una singola istanza quando in realtà non è così.
Il modo classico per impedire l'istanziazione è rendere privato il costruttore della classe.

> Bloch propone un trucco controverso, ovvero l'utilizzo di un tipo enumerato. Questo funziona perché il compilatore impedisce l'istanziazione di tipi enumerati. È un po' confusionario, meglio attenersi a un costruttore privato per garantire il vincolo di singola istanza.

Il SINGLETON, è controverso, perché ci sono più svantaggio che vantaggi.Un **singleton** è un'istanza globale accessibile da qualsiasi punto del programma, ma questo lo rende facile da abusare, causando dipendenze nascoste e codice difficile da mantenere. Inoltre, essendo responsabile del proprio ciclo di vita, è difficile da testare.
Un'alternativa migliore è spesso l'**iniezione di dipendenza**, che consente di passare un'unica istanza ai componenti che ne hanno bisogno. Tuttavia, non impedisce la creazione di più istanze: questo va garantito con attenzione progettuale e buona documentazione. L'iniezione, però, aiuta a gestire e propagare in modo controllato la stessa istanza.

Un problema particolare nella gestione dello stato è dovuto dall'uso di classi annidate in Java. Le classi annidate possono essere suddivise in

- **classi annidate statiche**, una classe definita all'interno di un'altra classe con la parola chiave static. Non ha un riferimento implicito all'istanza della classe esterna e può essere usata senza creare un oggetto della classe esterna.
- **classi interne**, una classe definita all'interno di un'altra classe senza la parola chiave static e questo la lega ad ogni istanza della istanza della classe esterna potendo accedere direttamente alle sue variabili. A loro si suddividono in:
    - classi anonime
    - classi locali.



Le **classi locali** sono classi che si definiscono all'interno di un metodo di una classe esterna. La loro visibilità è limitata proprio a quel metodo, quindi non possono essere usate al di fuori di esso. Un'altra caratteristica importante è che queste classi possono accedere ai parametri del metodo che le contiene, purché questi parametri siano dichiarati come final .
Un caso particolare di classi locali sono le **classi anonime**. Queste sono classi locali senza un nome, definite direttamente all'interno di un'espressione. Vengono spesso usate quando si vuole creare rapidamente un'implementazione di un'interfaccia o di una classe astratta, senza dover scrivere una nuova classe con un nome specifico.

Alernativa per implementare le **function object** è usare questi costrutti. Spesso si vuole **definire un comportamento** che dipende da **qualche valore locale** al momento della sua creazione.

"Ma se il metodo finisce, quella variabile non dovrebbe sparire?"

Esiste un concetto chiamato **closure** una funzione che può **ricordare e usare le variabili del contesto in cui è stata creata**, anche dopo che quel contesto è scomparso. In Java, quando usiamo **classi anonime** che fanno riferimento a variabili locali del metodo esterno, il compilatore salva automaticamente quelle variabili come campi interni nell'oggetto che viene creato.

COMPOSITION

Nella OOP i vari costrutti possono relazionarsi fra loro o tramite **composizione** o tramite l'**ereditarietà**. Queste due tecniche servono a stabilire come le parti di un software interagisco fra loro.

La **composizione** si riferisce alla relazione **parte di** tra classi in cui il ciclo di vita degli oggetti contenuti è strettamente associato al ciclo di vita dell'oggetto contenitore. La composizione è un tipo di associazione in cui una classe **contiene** il riferimento ad un'altra. La composizione è diversa dall'**aggregazione** con cui si intende un insieme di oggetti (liste...). La composizione è utile per segmentare un classe che altrimenti sarebbe troppo complessa e ciò violerebbe il principio della separazione delle responsabilità. Si incorre in un anti pattern detto **GOD CLASS**, una classe che fa tutto, sa tutto degli altri oggetti ecc, ma questo è sbagliato perché saremo constretti ad usare sempre questa classe per ogni cosa e ciò incrementa le dipendeze che ci sono nel programma essendo che non la si può estendere, creare un interfaccia ecc..
Viola anche la

> **Law of Demeter (LoD) – *"Principio del minimo contatto"***
> **Violazione**: Una *God Class* spesso accede direttamente a molte parti del sistema, diventando un nodo centrale che conosce troppo del resto dell'applicazione.

Per evitare tutto ciò si usa il meccanismo delle **delega**, cioè si delegano ad altri oggetti le azioni che si devono fare, essendo loro specializzati ad operare su quell'oggetto per quella azione. In questo modo si evita che un oggetto assuma troppe responsabilità ed è possibile implementare strategie, comportamenti o operazioni intercambiabili.

Una proprietà della composizione è la **transitività**, cioè un oggetto composto da altri oggetti può, a sua volta, essere un componente o un delegato di un altro oggetto padre.
Quando si lavora con queste gerarchie non è importante sapere si si una la composizione come tale o come delega, perché essendo strutture ricorsive possono essere viste in entrambi i modi: **come una relazione strutturale** (composizione) oppure **come una relazione comportamentale** (delegazione). Ciò che conta è **la responsabilità** che ciascun oggetto si assume e **il grado di accoppiamento** che si instaura tra le varie componenti del sistema.

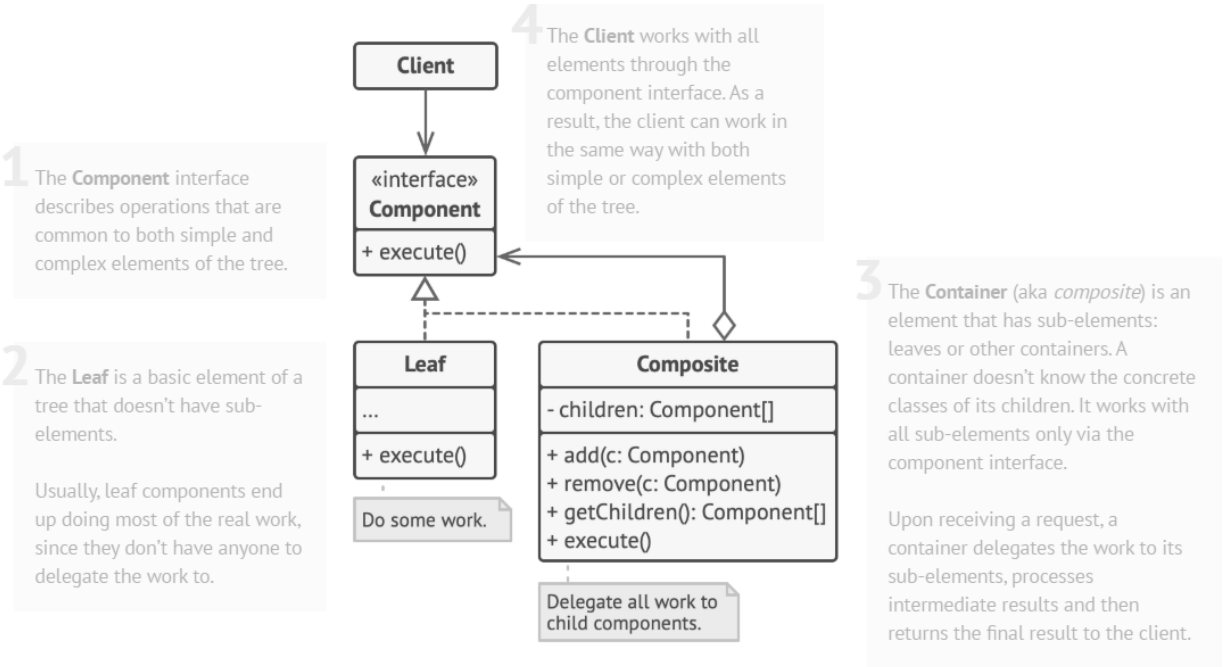
COMPOSITE PATTERN

Per poter creare una composizione esiste un design pattern**, COMPOSITE PATTERN**, che permette di trattare gruppi di oggetti come se fossero un oggetto singolo.
Un modo per realizzarla è quella di creare un interfaccia comune e di farla estendere da tutte le possibili implementazioni e di sfruttare le interfaccia e il polimorfismi, ma un problema è che il numero di implementazioni può esplodere e si dovrebbero avere già tutte le implementazioni disponibili essendo definite staticamente e non a runtime. Una soluzione migliore è stabilita dal COMPOSITE PATTERN.

Il pattern presenta tre elementi/ruoli, **componente, composito** e **foglia**.
L'elemento composito ha due caratteristiche:

- Aggrega diversi oggetti di tipo componente. L'utilizzo del tipo di interfaccia componente è importante, poiché consente composite per comporre qualsiasi altro tipo di elemento, inclusi altri compositi.
- Implementa l'interfaccia del componente. Questo è ciò che consente agli oggetti compositi di essere trattati dal resto del codice esattamente allo stesso modo degli elementi foglia.

Il codice client dovrebbe dipendere principalmente dal tipo di componente e non manipolare direttamente i tipi concreti, per poter trattare tutti gli oggetti nello stesso modo.



L'implementazione del metodo dell'interfaccia permette di iterare, navigare attraverso l'albero delle componenti e questo avviene perché i composite implementano anch'essi l'interfaccia Component e **delegano** il comportamento ai loro figli.

Un problema di implementazione da considerare è come aggiungere al composite le istanze del componente che compone, può essere fatto in due modi principali.

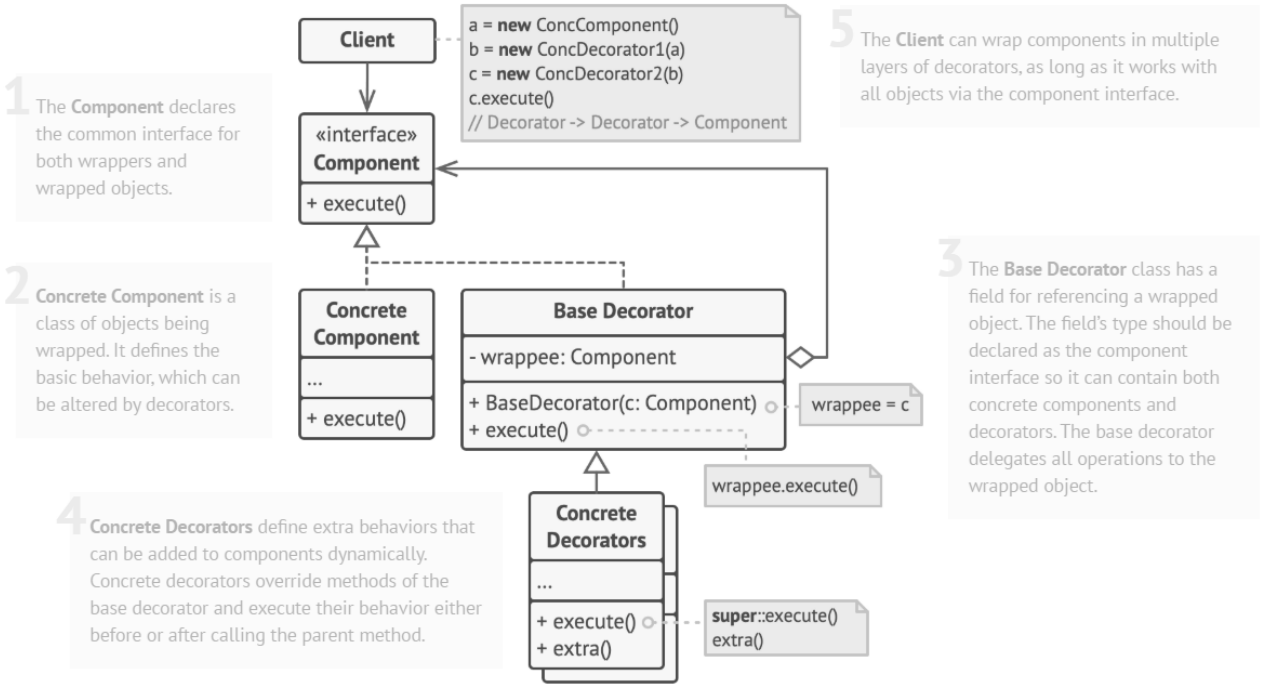
1. Avere un metodo nel composite che prenvede l'aggiunto del figlio (eventualmente anche la modifica, il getChild e la rimozione), ma s deve includere il metodo add nel componente?. No allora
2. Li si inserisce nel composite tramite il costruttore (lista ...).
    a. Usiamo il costruttore di copia per evitare la perdita di un riferimento alla struttura della collezione privata.
    b. O utilizzare il meccanismo **varargs** di Java per elencare singolarmente ogni sorgente di carta.

Si preferisce il metodo add perché può essere necessario modificare lo stato del composite in fase di esecuzione.
Questo aggiunge complessità al codice perché è necessario gestire un ciclo di vita più complesso per l'oggetto composite e la differenza tra l'interfaccia del componente (che non ha il metodo add) e quella del composito (che ce l'ha).
Se la modifica del composito in fase di esecuzione non è necessaria, è probabilmente meglio inizializzarlo una volta e lasciarlo così com'è.
Alcuni aspetti pratici relativi all'utilizzo del pattern sono indipendenti dalla struttura del pattern stesso, la soluzione dipenderà dal problema di progettazione specifico in questione. (sequence diagram).

DECORATOR PATTER

In certe situazioni è necessario avere oggetti che svolto una stessa azione, ma in modo diverso o che hanno particolari caratteristiche. Quindi possiamo creare una classe specializzato per poter fare questo. Questo lo si può fare tramite ereditarietà estendo la classe, ma non darebbe flessibilità al codice perché crea un legame statico fra le classi, ed è difficile poi cambiare comportamento a runtime. (Strategy)
Una soluzione può essere quella di avere una classe multimodale, che sulla base del "enum" assegnato svolge l'azione in un determinato modo Anche questa soluzione non ci piace perché potremmo non prevedere tutti i comportamenti ed andremmo ad accrescere lo spazio degli stati. Inoltre cadremmo un'altra volta nella **GOD CLASS**, Ah shit, here we go again, e anche nell'antipatten **SWITCH STATEMENT**.

Allora ci soccorre, per fortuna, il **DECORATOR PATTERN** che aggiungere dinamicamente funzionalità o comportamenti a un oggetto senza modificarne la struttura o creare tante sottoclassi.



Questo si basa sul **COMPOSITE**, tranne che per il fatte che il composite è un decorator base. Ogni decoratore **delegata l'esecuzione** al componente interno e può **combinarsi con altri decoratori**, rendendo semplice la composizione di funzionalità.

- Le decorazioni devono essere indipendenti e additive, non devono rimuovere comportamenti per rispettare i principi della progettazione orientata agli oggetti.
- In Java, è buona pratica rendere il riferimento all'oggetto decorato final e inizializzarlo nel costruttore, poiché si assume che non cambi durante la vita del decoratore.
- Un oggetto decorato non è uguale al componente originale: ha una nuova identità dovuta alla decorazione.

COPIA POLIMORFICA

Un problema di lavorare con grafi di oggetti, oltre all'identità, è la copia. Potremmo usare il costruttore della classe, ma questo è statico e non va bene avendo una struttura che sfrutta il polimorfismo tramite le interfaccie, quindi non possiamo sapere che tipo abbiamo a runtime. E mo? La **copia polimorfica** risolve tutti nostri guai, perché permette di fare copie senza saper il tipo concreto dell'oggetto. Questo concetto è detto anche **clonazione**. Implementare il medo di copy() in strutture ricorsive è complesso.

Java dà la possibilità di utilizzare il **tipo di ritorno covariante** cioè permette a un metodo sovrascritto in una sottoclasse di restituire un tipo più specifico rispetto

a quello dichiarato nel metodo della superclasse o dell'interfaccia. Si tratta di una funzionalità molto utile, perché consente di evitare il ricorso a **downcast espliciti**.

```java
// Interfaccia base con metodo copy che restituisce un tipo generico
interface Animal {
    Animal copy();
}

// Classe concreta che implementa Animal e usa tipo di ritorno covariante
class Dog implements Animal {
    @Override
    public Dog copy() {
        return new Dog();
    }
}
```

```java
// Uso del tipo di ritorno covariante
public class Main {
    public static void main(String[] args) {
        Dog originalDog = new Dog();

        // Grazie al ritorno covariante, non serve il cast
        Dog copiedDog = originalDog.copy();

        System.out.println("Copia di Dog creata: " + copiedDog);
    }
}
```

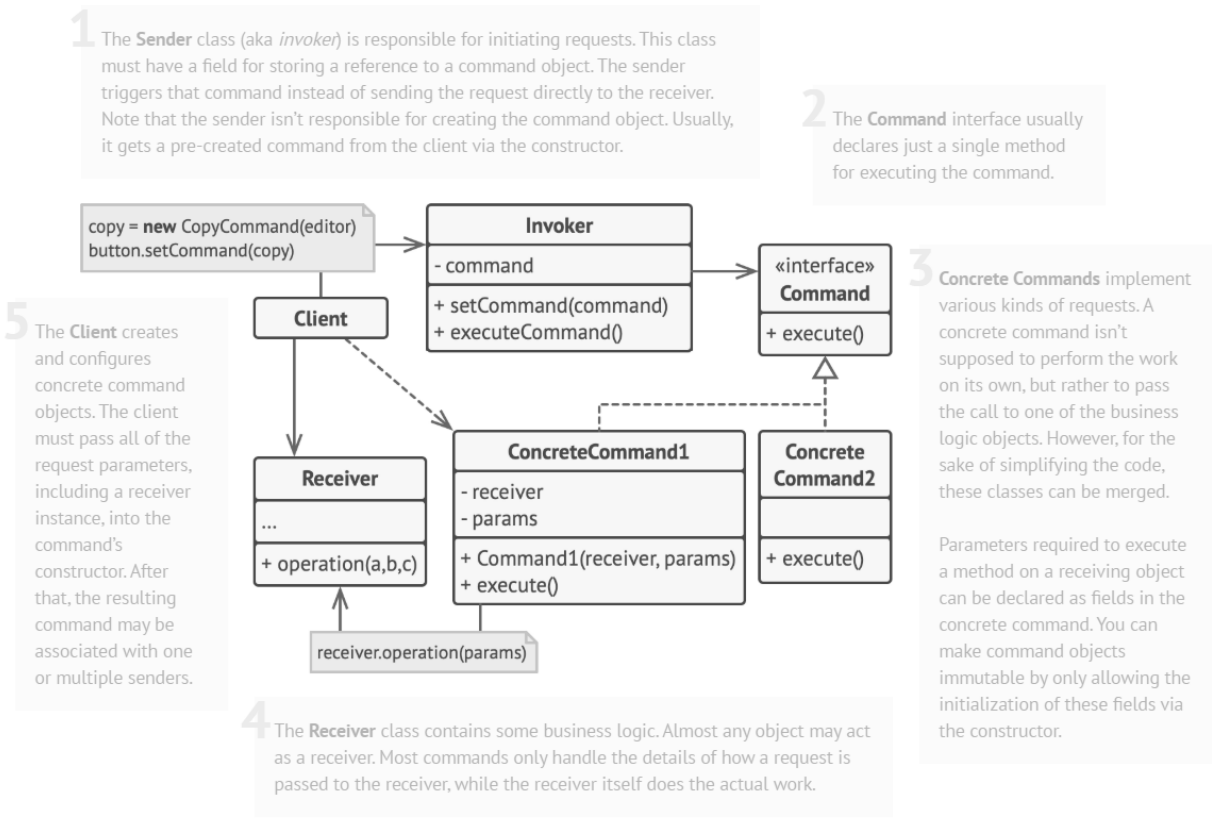
Questo meccanismo si rivela ancora più importante nel contesto di strutture **ricorsive**, come ad esempio collezioni o oggetti che contengono al loro interno riferimenti ad altri oggetti dello stesso tipo. In questi casi, è fondamentale che la copia effettuata sia profonda (deep copy), cioè che non si limiti a copiare i riferimenti, ma replichi realmente anche le componenti interne.

PROTOTYPE PATTERN

COMMAND PATTERN

In certe situazioni vogliamo svolgere delle azioni che possono essere svolte in modi diversi, o invocate in modo diverso. Potremmo usar l'ereditarietà, ma per i classici motivi non ci va bene. Allora una soluzione, basata sul principio di separazione delle responsabilità è di trasformare l'azione, o meglio il metodo che in generale è una proprietà di una classe in un oggetto, in modo di applicargli delle proprietà diverse. Per farlo esiste il **COMMAND PATTERN** che sfrutta sempre il polimorfismo tramite le interfacce, che in questo caso contengono il metodo per svolgere l'azione.

Il **COMMAND** separa l'oggetto che invoca l'operazione (**Invoker**) da quello che la esegue (**Receiver**), permettendo di trattare le richieste come oggetti riutilizzabili, componibili e serializzabili.



LAW OF DEMETER

Nella progettazione di tutto ciò, possiamo andare a creare una lunga catena di delegazione fra gli oggetti. Questo ci va bene, ma fino ad un certo punto perché secondo la **legge di Demeter** si devono **l**imitare le dipendenze tra le classi per rendere il codice più modulare, facile da mantenere e meno fragile di fronte ai cambiamenti. Ciò vuol direche un oggetto dovrebbe interagire solo con pochi amici stretti . Cadremmo nll'antipattern del **MESSAGE CHAIN** che **espone** dettagli interni degli oggetti che dovrebbero invece rimanere nascosti, violando l'information hiding.

```
double prezzo = cliente.getOrdine().getProdotto().getPrezzo();
```

Questa "legge" stabilisce che il codice di un metodo dovrebbe accedere solo a:

- le variabili di istanza del suo parametro implicito;
- gli argomenti passati al metodo;
- qualsiasi nuovo oggetto creato all'interno del metodo;
- oggetti disponibili globalmente.

Per rispettare questa linea guida, diventa necessario fornire servizi aggiuntivi nelle classi che occupano una posizione intermedia in una catena di aggregazione/delega, in modo che i client non debbano manipolare gli oggetti interni incapsulati da questi oggetti.

INVERSION OF DEPENDACY

Il principio di inversione delle dipendenze afferma che i moduli di alto livello non dovrebbero dipendere da moduli di basso livello. Invece, entrambi dovrebbero dipendere dalle astrazioni, e le astrazioni non dovrebbero dipendere dai dettagli; I dettagli dovrebbero dipendere dalle astrazioni.

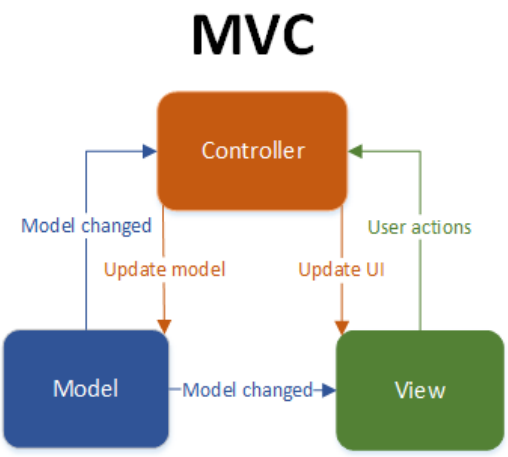
Questo principio ha lo scopo di **ridurre l'accoppiamento tra moduli** e aumentare la flessibilità del sistema, facilitando l'estensione e la manutenzione. Un antipattern in cui si cade è **PAIRWISE DEPENDECIES** cioè si creano relazioni dirette fra due componenti o moduli in un sistema. Ogni dipendenza rappresenta un legame che può introdurre accoppiamento e potenziali problemi di manutenzione.
Quando molte dipendenze si accumulano fra moduli, si crea una rete complessa che rende il sistema rigido e difficile da modificare. Si decve cercare di ridurre il numero e la forza di queste dipendenze.

Introducendo un livello di astrazione, invece di dipendere direttamente da una classe concreta (**dipendenza forte**), si dipende da un'interfaccia, riducendo così il coupling. In questo modo:

- Le modifiche in una classe concreta hanno meno impatto sugli altri moduli.
- Il sistema diventa più modulare, flessibile e testabile.

Un pattern per risolvere questo è il **MVC MODEL-VIEW-CONTROLL** un pattern architetturale che separa le responsabilità in tre componenti :

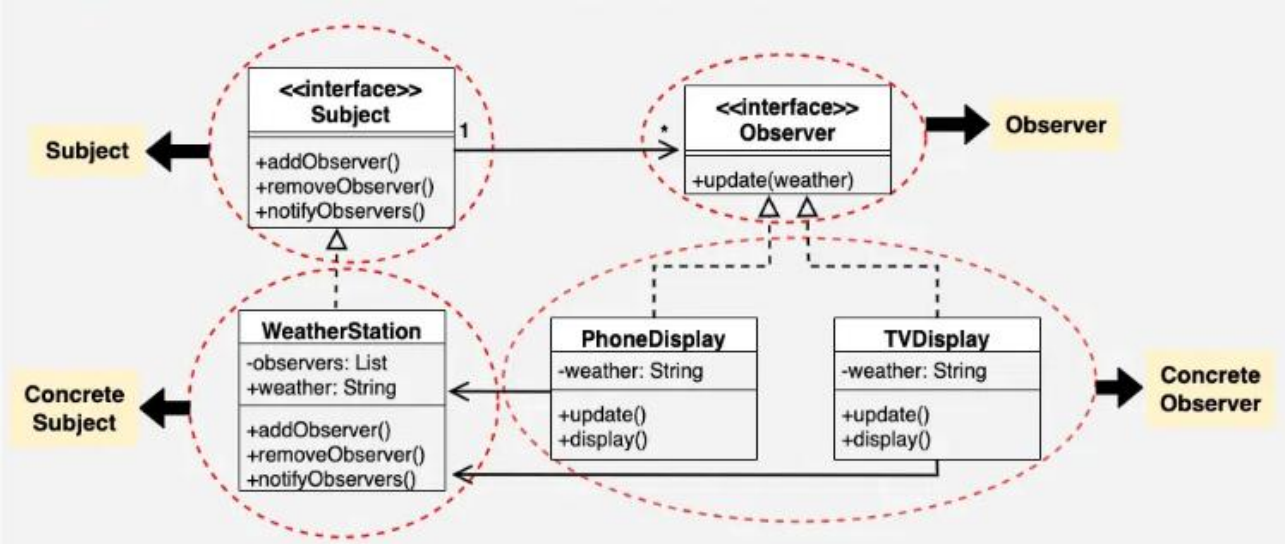
1. **Model:** Rappresenta la parte che gestisce i dati, la logica di business e lo stato dell'applicazione. Il modello non si preoccupa di come i dati vengono mostrati all'utente, ma solo di gestirli, aggiornarli e fornirli.
2. **View:** È responsabile della presentazione delle informazioni all'utente. La vista prende i dati dal modello e li mostra in modo appropriato, ad esempio sotto forma di pagine web, finestre grafiche, o altro. Non contiene logica di business, solo logica di visualizzazione.
3. **Controller:** Fa da intermediario tra l'utente e il sistema. Riceve gli input dell'utente (clic, comandi, dati inseriti), li interpreta e decide come modificare il modello o quale vista mostrare. In pratica coordina il flusso dell'applicazione.



Questo è il concetto generale, ma ci sono molti modi di implementarlo, un esempio è

OBSERVER PATTERN

Il pattern OBSERVER consiste nell'archiviare i dati in un oggetto specializzato e consentire ad altri oggetti di osservarli. Questo oggetto, detto **subject** (MODEL), mantiene una lista di altri oggetti, detti **observers**, che vogliono essere informati quando qualcosa cambia nello stato del soggetto. Quindi deve avere i metodi per la registrazione e deregistrazione degli observer.
Il soggetto **notifica automaticamente** tutti gli osservatori ogni volta che il suo stato cambia, senza che gli osservatori debbano controllare continuamente (polling).



DATA FLOW

Come fanno gli osservatori a sapere che c'è un cambiamento nello stato del modello di cui hanno bisogno? Sono spioni...
Ogni volta che si verifica un cambiamento nello stato del modello il modello lo comunica agli osservatori scorrendo l'elenco degli osservatori e chiamando il metodo notify(). Questo metodo è una callback per il DIP perché per ottenere informazioni dal modello, gli osservatori non chiamano un metodo sul modello, ma aspettano che il modello li richiami. Poi ovviamente l'observer ha bisogno di un metodo per poter aggiornare lo stato delle sue variabili.
Per garantire che il modello notifichi gli osservatori ogni volta che si verifica un cambiamento di stato, e che ci sia consistenza fra i dati, sono possibili due strategie:

1. una chiamata al metodo di notifica deve essere inserita in ogni metodo che modifica lo stato; in questo caso il metodo può essere dichiarato privato;

2. È necessario fornire una documentazione chiara che indichi agli utenti della classe modello di chiamare il metodo di notifica ogni volta che il modello deve informare gli osservatori. In questo caso, i metodi di notifica devono essere non privati.

CONTROLL FLOW

Come accedono gli osservatori alle informazioni che devono conoscere dal modello?

Si possono rendere disponibili le informazioni di interesse tramite uno o più parametri del callback. Questa strategia è anche nota come **strategia push data flow**, poiché il modello invia esplicitamente i dati di una struttura agli osservatori. Applicando questa strategia, potremmo definire il metodo di callback in modo che includa un parametro che rappresenti l'informazione memorizzata più di recente nel modello.

Questa strategia parte dal presupposto che sappiamo in anticipo quale tipo di dati del modello richiederanno gli osservatori.

Tuttavia, questo non è il caso generale. Una strategia più flessibile consiste nel consentire agli osservatori di estrarre i dati desiderati dal modello utilizzando i metodi di query definiti nel modello. Questo approccio è noto come **strategia pull data-flow**. Per implementare la strategia di flusso di dati pull, gli osservatori devono avere un riferimento al modello, ma questo riferimento non deve essere necessariamente fornito come argomento al metodo di callback. Un'altra opzione è inizializzare gli oggetti osservatori con un riferimento al modello (memorizzato come campo) e fare riferimento a tale campo se necessario.

Può sembrare che questa strategia introduca una **dipendenza circolare** tra un modello e i suoi osservatori, dato che entrambi dipendono l'uno dall'altro, ma non è così perché il modello non conosce il tipo concreto dei suoi osservatori. Attraverso la segregazione delle interfacce, l'unica porzione di comportamento che il modello richiede dagli osservatori viene specificata tramite il loro metodo di callback. Anche se però, aumenta l'accoppiamento tra osservatori e modello.

ADAPTER PATTERN

L'**ADAPTER PATTER** è un modello di progettazione strutturale che consente la collaborazione di oggetti con interfacce incompatibili. Si tratta di un oggetto speciale che converte l'interfaccia di un oggetto in modo che un altro oggetto possa comprenderlo.

Un adattatore esegue il wrapping di uno degli oggetti per nascondere la complessità della conversione che avviene dietro le quinte. L'oggetto avvolto non è nemmeno a conoscenza dell'adattatore.

Gli adattatori non solo possono convertire i dati in vari formati, ma possono anche aiutare gli oggetti con interfacce diverse a collaborare. Ecco come funziona:

1. L'adattatore ottiene un'interfaccia compatibile con uno degli oggetti esistenti.
2. Utilizzando questa interfaccia, l'oggetto esistente può chiamare in modo sicuro i metodi dell'adapter.
3. Alla ricezione di una chiamata, l'adapter passa la richiesta al secondo oggetto, ma nel formato e nell'ordine previsti dal secondo oggetto.

A volte è anche possibile creare un adattatore bidirezionale in grado di convertire le chiamate in entrambe le direzioni.