Documento a cura di Simone Remoli e Xian P'eng

CAPIRE I PATTERN DI DESIGN - ATTO PRATICO

Questo documento ha l'obiettivo di fornire una sintesi chiara, lineare e dettagliata per comprendere i design pattern. In precedenza è stato prodotto un documento volto a offrire una panoramica ampia e completa dei design pattern trattati nel corso di Ingegneria del Software. In questo testo cercherò di renderli ancora più comprensibili e di semplificarne l'apprendimento; inoltre per leggere questo documento dovete cliccare il link al mio **profilo GitHub**, così da poter seguire i pattern che cito uno per uno.

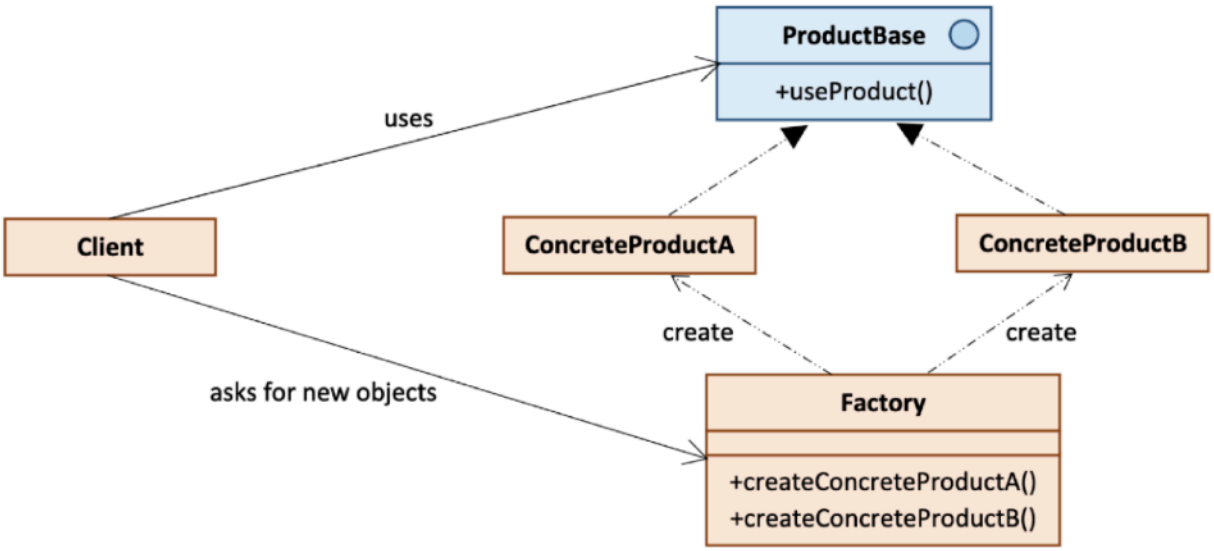
Il modo migliore per apprendere un design pattern è interiorizzarlo, farlo realmente proprio. Un buon metodo consiste nel creare una cartella su GitHub contenente tutti e sette i design pattern, così da poterli consultare e riutilizzare ogni volta che se ne presenti la necessità.

Naturalmente i pattern affrontati in questo corso sono solo 7, cioè 7 su 23 design pattern presentati nel libro della GoF, pari a circa il 30,4% del totale. Tuttavia, questi sono quelli richiesti per l'esame e per la realizzazione del progetto.

Iniziamo.

Factory Method

Molto brevemente, il factory method è un modello di progettazione creazionale che serve a spostare la logica di creazione degli oggetti fuori dalle classi di dominio.



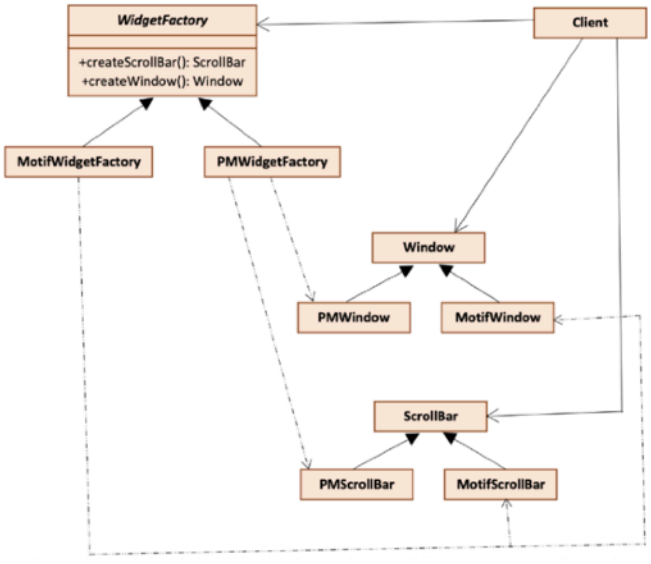
Elementi di questo pattern:

Nome Elemento	Tipologia	Spiegazione
ProductBase	Interfaccia o Classe Astratta	Dichiara le operazioni comuni a tutti i suoi prodotti.
ConcreteProductA	Classe Concreta	Implementazione concreta dell'interfaccia o classe astratta.
ConcreteProductN	Classe Concreta	Implementazione concreta dell'interfaccia o classe astratta.
Factory	Classe Concreta	È la fabbrica che produce le istanze e contiene i metodi che si occupano di creare oggetti concreti.

Nell'esempio di codice mostrato sono state create quattro classi concrete, una classe astratta e l'interfaccia base.
Data la semplicità del programma, ometterò la spiegazione dettagliata.
In sintesi, le classi concrete implementano i metodi comuni definiti nell'interfaccia, e quest'ultima viene utilizzata come tipo di riferimento per instanziare un prodotto concreto.

Abstract Factory

Molto brevemente, l'abstract factory fornisce un'interfaccia per la creazione di **famiglie** di oggetti.



Elementi di questo pattern:

Nome Elemento	Tipologia	Spiegazione
Abstract Product	Interfaccia o Classe Astratta	Definisce le operazioni e i metodi che possiede un prodotto.
Concrete Product	Classe Concreta	Implementa (o estende) il prodotto astratto.
Abstract Factory	Classe Astratta	Definisce un insieme di metodi di creazione astratti (da implementare nei concrete factory) che tornano gli abstract product e dei metodi che tornano le factory concrete in base ad una condizione. La funzione è statica.
Concrete Factory	Classe Concreta	Ritorna più prodotti concreti dello stesso tipo ed estende abstract factory.

Nel mio caso esistono due abstract product: **Human** e **FourleggedAnimal**.

Esiste una classe abstract factory generica che crea questi abstract product, non prima di aver creato le factory concrete.

Le factory concrete sono: **NoSeniorFactory** e **SeniorFactory**, entrambe estendono da abstract factory.

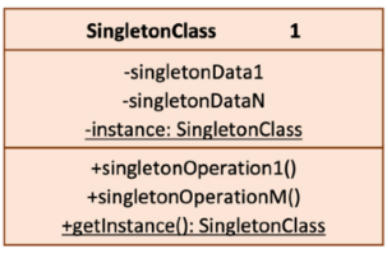
Queste factory concrete ritornano (tramite metodi override) più prodotti concreti dello stesso tipo.

I prodotti concreti sono: **NoSeniorFourLeggedAnimal**, **NoSeniorHuman**, **SeniorFourLeggedAnimal**, **SeniorHuman**.

Le **concrete factory** creano **prodotti concreti dello stesso tipo**.

Singleton

Molto brevemente, lo scopo di questo pattern è quello di assicurare che una classe abbia una sola istanza nell'applicazione e fornire un punto d'accesso globale a tale istanza.

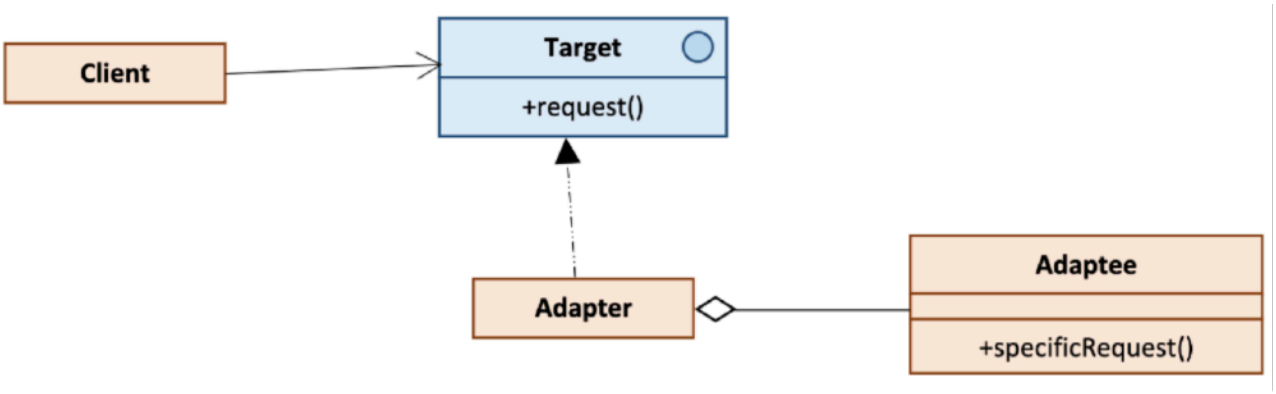


Il Singleton è come una ***mucca da mungere***: nella fattoria esiste un'unica mucca, e tutti i contadini devono utilizzare sempre quella. Allo stesso modo, nel pattern Singleton esiste **una sola istanza della classe**, alla quale tutte le altre classi accedono per ottenere informazioni o servizi condivisi, senza poter creare ulteriori istanze.

Il codice rappresenta un'implementazione del *Singleton* tramite il meccanismo dell'**inner static class**, una delle versioni più sicure ed efficienti del pattern. La classe Singleton ha un costruttore protetto e non può quindi essere istanziata dall'esterno. L'unica istanza disponibile viene creata all'interno della classe annidata Container, nel campo instance, inizializzato con due parametri (**10** e "Mum").
Il metodo statico getSingletonInstance() restituisce sempre questa stessa istanza, garantendo l'unicità dell'oggetto durante tutta l'esecuzione del programma. La classe espone inoltre alcuni metodi setter che modificano il valore dei campi interni (singletonData1 e singletonDataN) e i relativi getter per la loro lettura.
In sintesi, questo codice garantisce l'esistenza di **una sola istanza condivisa**, inizializzata una sola volta e accessibile tramite un metodo statico, seguendo il principio fondamentale del pattern Singleton.

Adapter

Molto brevemente, il pattern adapter permette al client di usare una classe esistente adaptee, che non voglio modificare, come se implementasse un'altra interfaccia.



Elementi di questo pattern:

Nome Elemento	Tipologia	Spiegazione
Target	Interfaccia o Classe Astratta	Definisce le operazioni da implementare.
Adaptee	Classe Concreta	È la classe da adattare.
Adapter	Classe Concreta	Ponte tra target e adaptee.

Lo scopo dell'Adapter è permettere al client di utilizzare l'Adaptee attraverso l'interfaccia Target, anche se le due interfacce sono incompatibili. L'Adapter fa da intermediario e traduce le chiamate dei metodi del Target nei corrispondenti metodi dell'Adaptee.

Ma quando serve questo pattern?

L'Adapter serve **quando il client NON PUÒ** chiamare direttamente l'Adaptee, quindi si usa l'adattatore che traduce le chiamate nell'adaptee.
L'Adapter non serve per comodità, ma per consentire compatibilità senza toccare il resto del sistema.

Immagina un programma vasto dove varie parti devono usare **la stessa interfaccia Target.**

Ora, se un giorno arriva una classe **incompatibile** (Adaptee) che ha metodi diversi, il client NON può adattarsi, non può cambiare il codice, non può essere modificato (magari perché è già in produzione), non può conoscere nuove classi.
L'unico modo per rendere compatibile l'Adaptee è un **Adapter.**

Decorator

Il Decorator è uno dei pattern **più importanti** della GoF; non conoscerlo è come cercare di leggere un libro al buio: puoi anche provarci, ma non capirai molto. Il package **java.io** è costruito interamente con il pattern Decorator.

Il Decorator è un pattern che permette di **aggiungere nuove funzionalità o comportamenti a un oggetto in modo dinamico**, senza modificare la sua struttura originaria.
Per comprenderlo intuitivamente, immaginiamo di avere un oggetto *Persona* inizialmente "nudo", privo di qualsiasi accessorio. Attraverso il Decorator, possiamo aggiungere progressivamente elementi come vestiti, cappelli, occhiali o accessori vari, ciascuno incapsulato in un nuovo decorator.
Ogni "strato" aggiunto arricchisce l'oggetto di una nuova responsabilità, proprio come un capo d'abbigliamento aggiunge un nuovo aspetto o una nuova funzione alla persona.

Ogni capo è un **layer**.

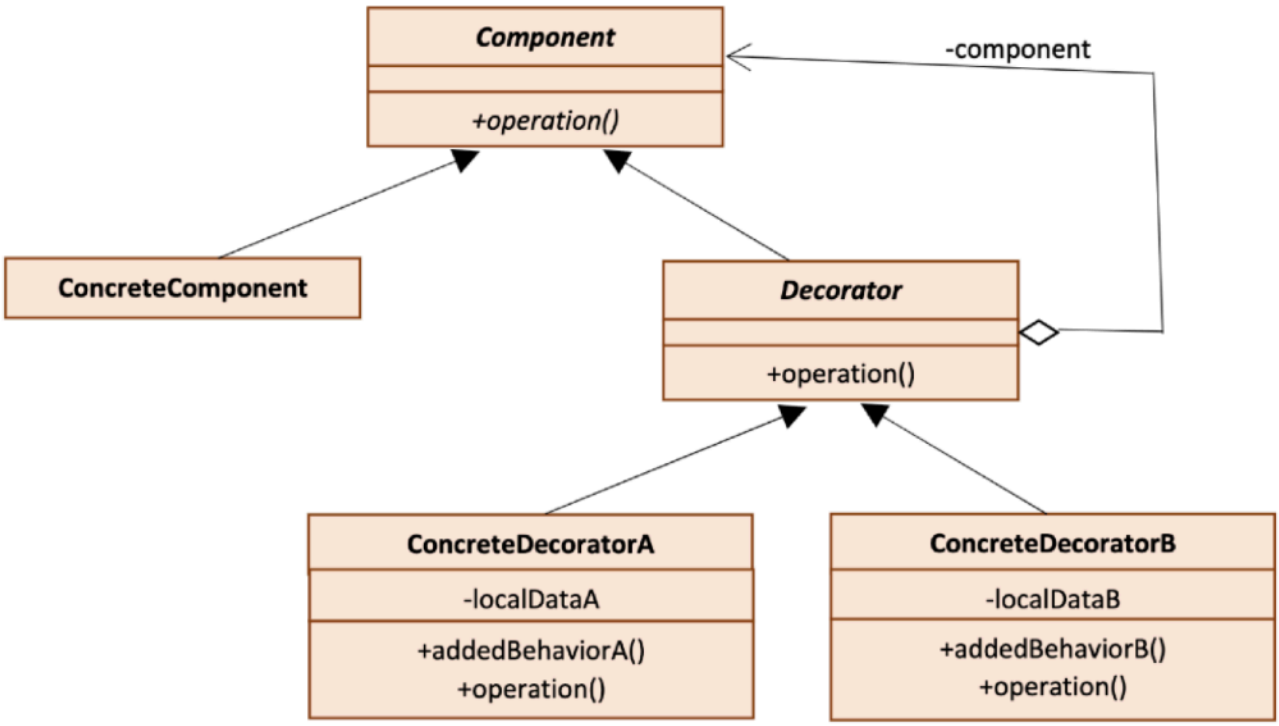
La persona rimane sempre la stessa. Il vestito non modifica la classe Persona, si limita a "decorarla": si aggiungono solo **strati** che ne **estendono** le capacità.
Puoi combinare i vestiti come preferisci: Persona + maglietta, Persona + maglietta + giacca, Persona + maglietta + giacca + cappello, etc…!

Elementi di questo pattern:

Nome Elemento	Tipologia	Spiegazione
Component	Interfaccia o Classe Astratta	Definisce l'operazione base che sia l'oggetto originale sia i decorator devono implementare.

Nome Elemento	Tipologia	Spiegazione
Concrete Component	Classe Concreta	Implementa component, è l'oggetto reale di base su cui si deve lavorare, ossia l'oggetto da "decorare".
Abstract Decorator	Classe Astratta	Implementa e aggrega internamente component.
Concrete Decorator	Classe Concreta	Estende decorator e aggiunge nuove funzionalità all'oggetto decorato.

Di seguito lo schema riassuntivo del pattern decorator:



Il punto più difficile da capire del Decorator è che **NON aggiunge nuovi metodi**, ma **aggiunge NUOVE RESPONSABILITÀ al metodo che già esiste**. Il beneficio NON è "più metodi", ma "più comportamento".

Esempio:

```
OutputStream os =
    new BufferedOutputStream(
        new GZIPOutputStream(
            new CipherOutputStream(
                new FileOutputStream("dati.bin"), cipher
            )
        )
    );
```

Alla fine ho sempre un OutputStream con un solo metodo principale:

```
os.write(...)
```

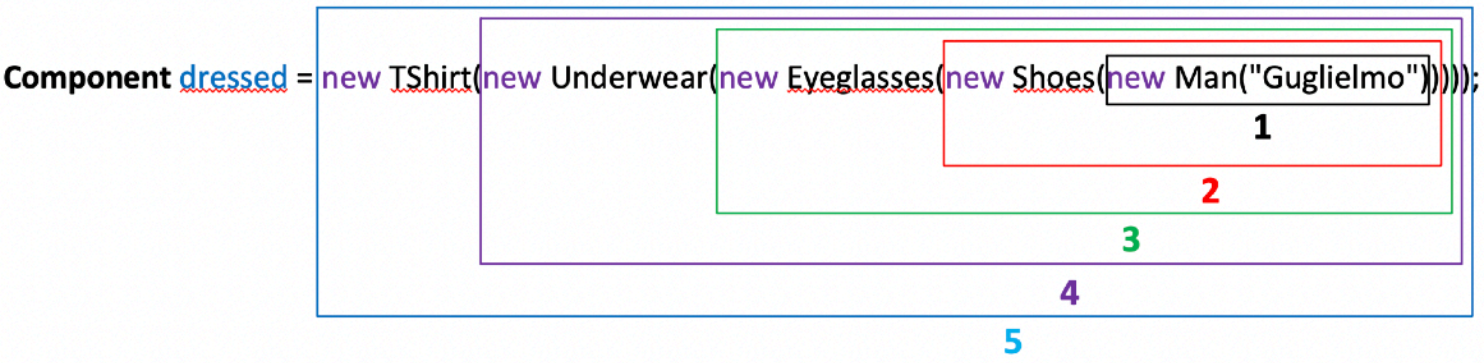
Ma il comportamento di `write()` è diventato **COMPLETAMENTE DIVERSO**!

Il codice che ho caricato su GitHub mostra esattamente questo.

Il decorator funziona tramite una **ricorsione**.

De Angelis definisce questo meccanismo ricorsivo come "**Ricorsione Strutturale**".

Analizziamo il codice e la struttura ricorsiva.



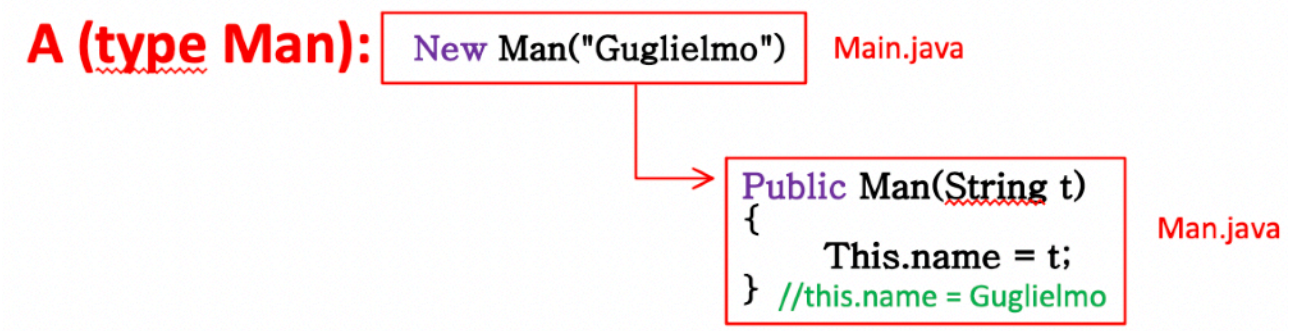
dressed punta SOLO a TShirt.

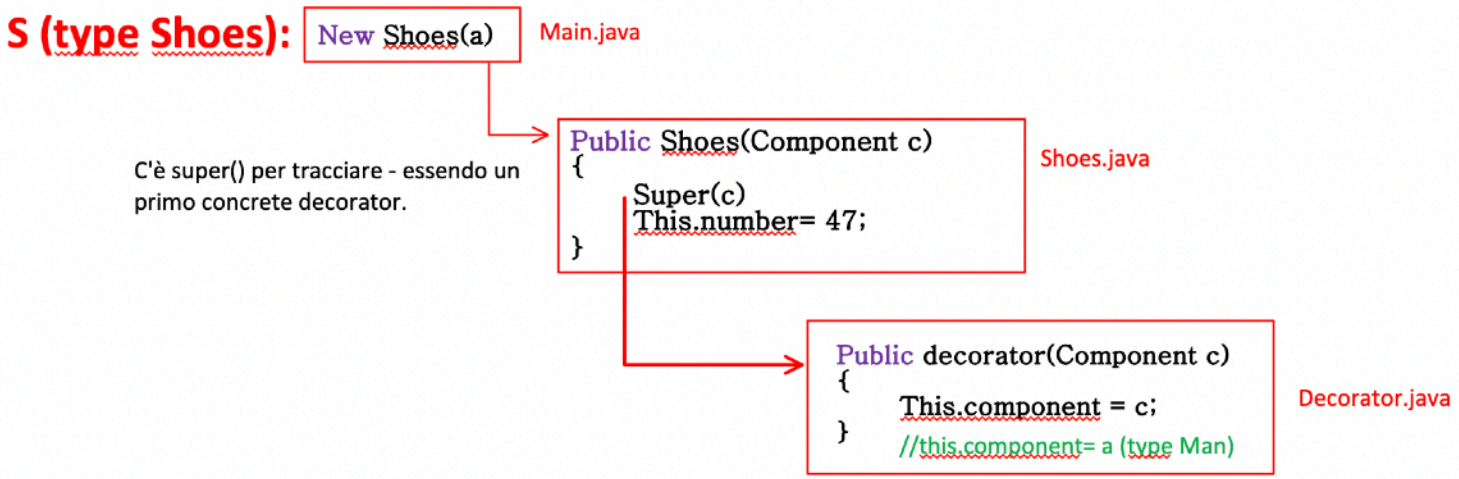
Ma dentro TShirt c'è un Underwear,
dentro Underwear c'è un Eyeglasses,
dentro Eyeglasses c'è un Shoes,
dentro Shoes c'è il Man.

È una **matrioska** perfetta.

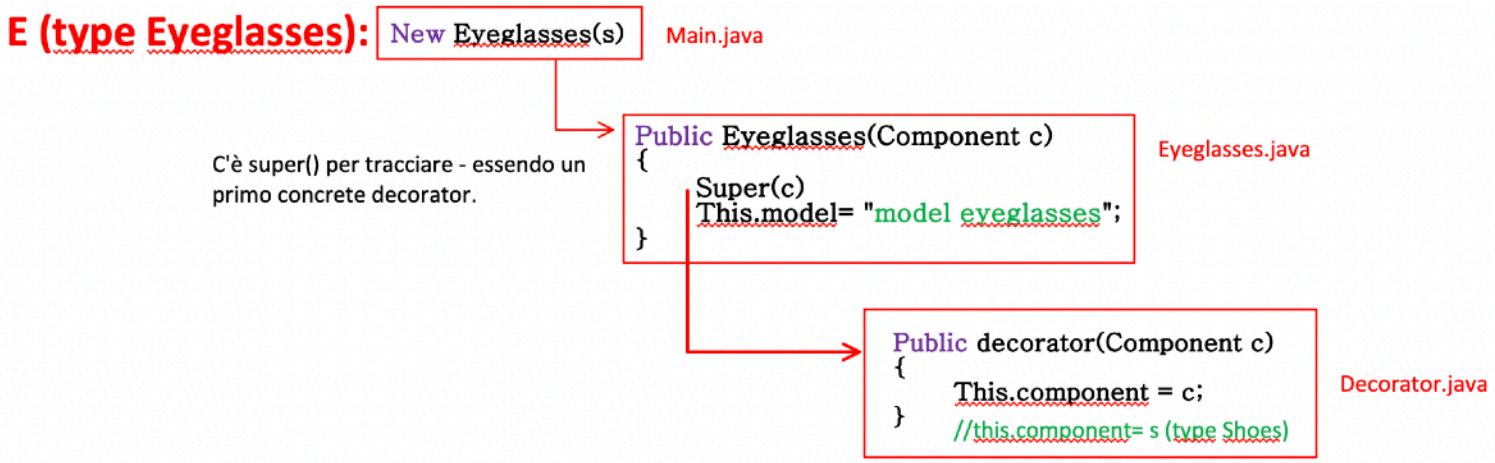
Le parentesi colorate rappresentano **ogni costruttore** chiamato.
Ogni oggetto ingloba quello precedente.

Man.java rappresenta il **ConcreteComponent**.
Gli altri sono tutti **decoratori concreti**, che hanno lo scopo di aggiungere funzionalità.

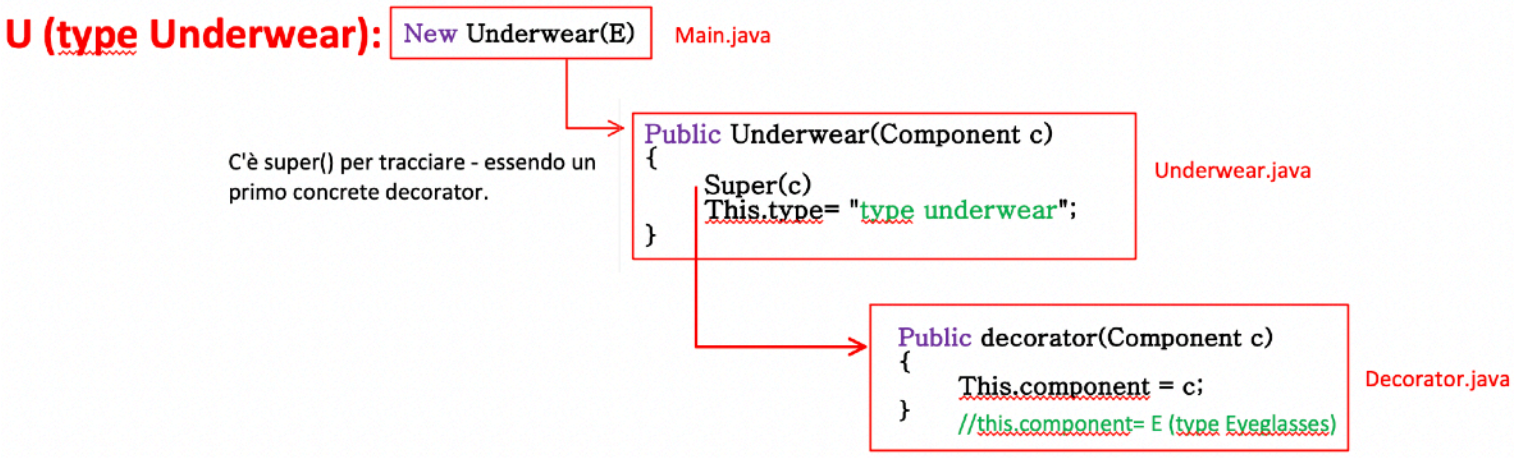




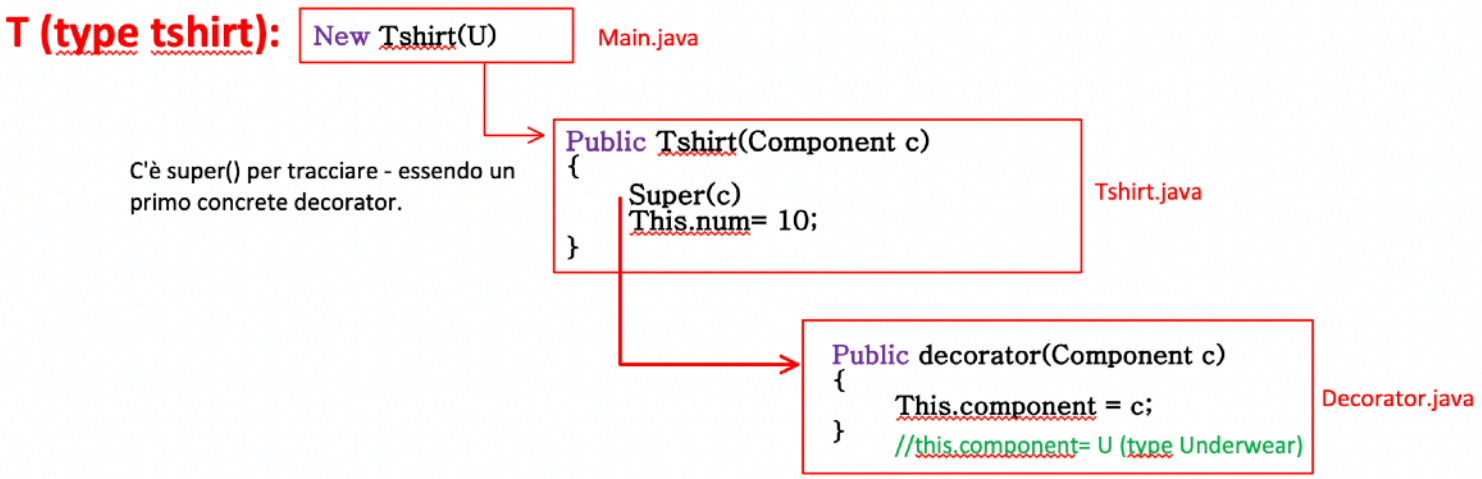
S->A. (S punta ad A).



E->S->A (E punta ad S che punta ad A).



U->E->S->A (U punta ad E che punta ad S che punta ad A).



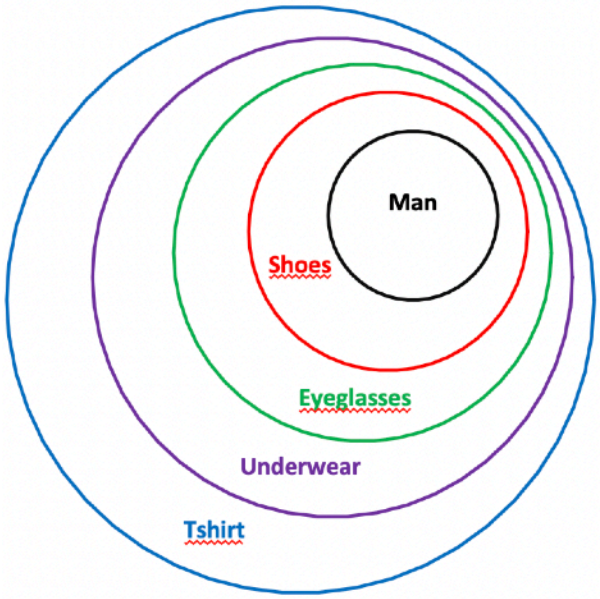
T->U->E->S->A.

A questo punto "Component = T".

E si chiama la funzione di component.

```
Component c = T;
System.out.println(c.dress_code());
```

Si è creata questa struttura, ogni livello ha contribuito nella creazione di un layer decorativo.



Che non è il logo di "disco Lazio" :D

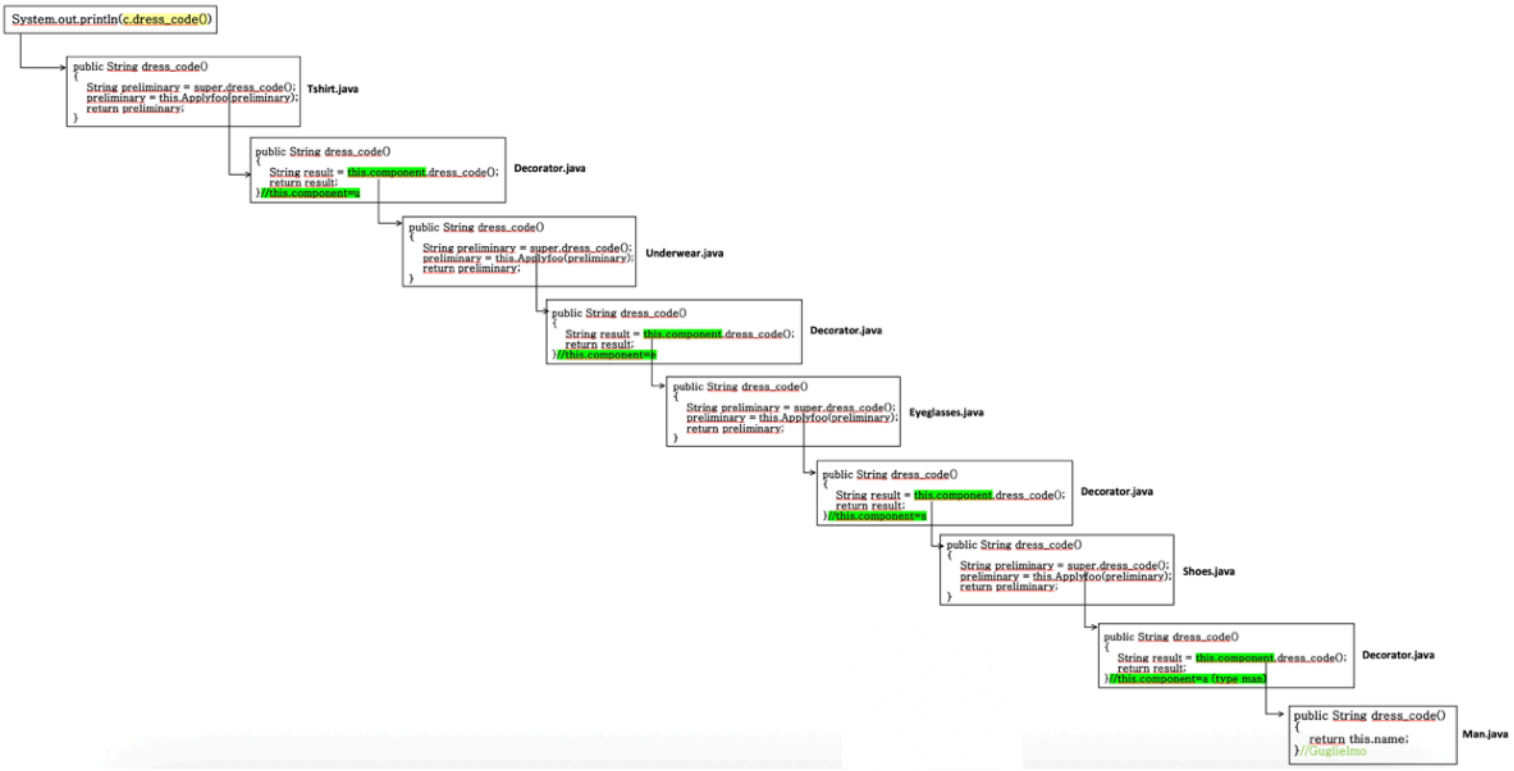
L'immagine mostra **l'essenza del Decorator Pattern:**
un oggetto di base (Man) viene avvolto da una serie di decoratori, ognuno dei quali aggiunge una funzionalità o un'informazione.
Ogni cerchio rappresenta un ***layer***, cioè un oggetto decoratore che contiene quello più interno.

Okay, fine della premessa.

Adesso non abbiamo fatto ancora nulla, abbiamo solamente creato la catena di chiamate, ora il pattern procederà in maniera ricorsiva automaticamente.
Vediamolo nel dettaglio.

RICORSIONE STRUTTURALE

La costruzione degli oggetti decoratori non attiva ancora alcuna logica: si limita a creare una catena di contenitori annidati. La decorazione vera e propria avviene solo quando viene invocato il metodo `dress_code()`, che inizia un processo ricorsivo dal decoratore più esterno verso il componente più interno.
La stampa è quindi necessaria per osservare l'effetto finale della concatenazione.



L'immagine mostra la sequenza completa delle chiamate al metodo `dress_code()` nella catena del Decorator Pattern.
Parte un vero e proprio **processo ricorsivo discendente**, che attraversa tutti i decoratori fino ad arrivare al componente concreto (`Man`).

Nota: ChatGPT spesso sbaglia la logica del Decorator.

La risalita inizia nel momento in cui il componente concreto restituisce il valore base.
Da quel punto in avanti, ciascun decoratore riceve il valore del decoratore interno, lo arricchisce con `Applyfoo()`, e lo restituisce al decoratore più esterno successivo. La logica di decorazione emerge quindi in fase di risalita, proprio come avviene nelle funzioni ricorsive.

Il flusso del Decorator funziona esattamente come una ricorsione.
E non è una metafora: **il Decorator è proprio una ricorsione strutturale**, identica al meccanismo di una funzione ricorsiva pura.

Considerazioni finali sul decorator.

1. Il componente concreto rappresenta il caso base del processo ricorsivo; una volta restituito il valore originario, ogni decoratore aggiunge il proprio contributo semantico durante la risalita.

2. Il risultato finale è dato da una costruzione cumulativa: ogni decoratore accede alla risposta dell'elemento più interno, la modifica secondo la propria logica e la restituisce allo strato superiore.

3. L'approccio decorativo favorisce un alto grado di personalizzazione del comportamento, con la possibilità di costruire oggetti complessi mediante combinazioni dinamiche di decoratori.

4. Il pattern migliora la coesione dei singoli moduli e riduce il coupling tra le classi, mantenendo un'architettura pulita e facilmente estensibile.

5. Il Decorator permette di applicare estensioni a runtime, definendo la configurazione comportamentale dell'oggetto solo al momento dell'istanziazione della catena di decoratori.

6. L'ordine con cui i decoratori vengono applicati incide direttamente sul risultato finale, conferendo al pattern una natura profondamente flessibile e modulare.

Un po' di storia.

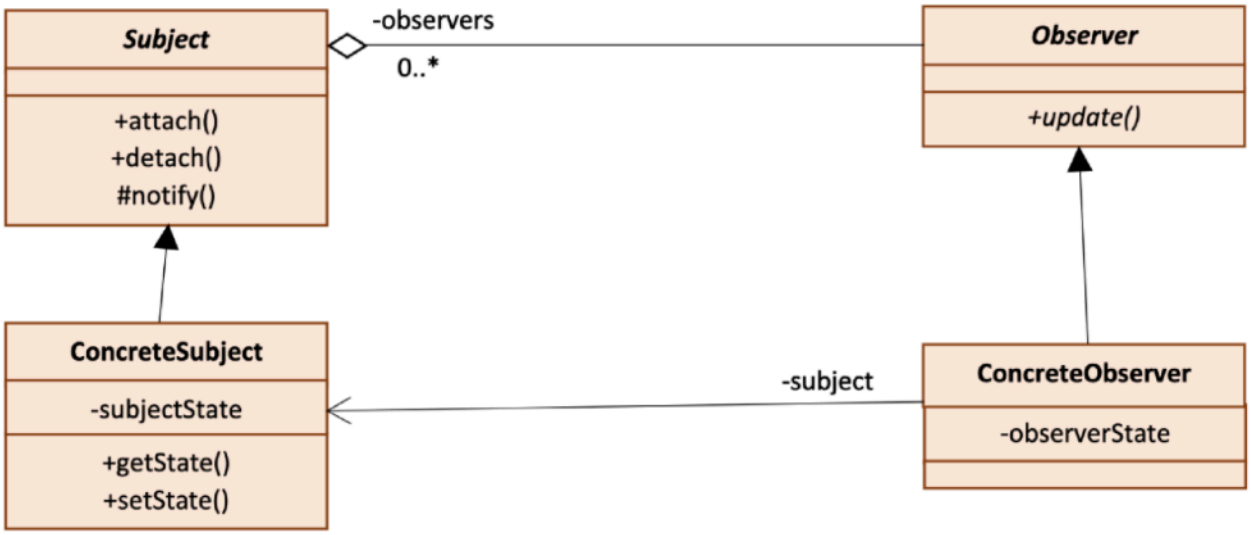
Il Decorator Pattern è stato formalizzato nel 1994 dai Gang of Four (Gamma, Helm, Johnson, Vlissides) nel libro *Design Patterns: Elements of Reusable Object-Oriented Software*, che ha sistematizzato concetti già presenti nella programmazione a oggetti e nelle architetture a componenti degli anni '80. Il suo scopo è consentire l'estensione dinamica del comportamento degli oggetti senza ricorrere all'ereditarietà, evitando quindi un'esplosione di sottoclassi e mantenendo il codice aperto all'estensione ma chiuso alla modifica.

Ad oggi, il decorator pattern è diventato uno standard formale dell'ingegneria del software, insegnato in qualunque corso universitario di progettazione del software.

Observer

Molto brevemente, questo pattern definisce una dipendenza: se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo vengono notificati e aggiornati automaticamente.

Regola pratica: *il subject mantiene una lista di observers*.



Elementi di questo pattern:

Nome Elemento	Tipologia	Spiegazione
Subject	Classe Astratta	Oggetto base da osservare.
Observers	Classe Astratta o Interfaccia	Classe base osservatrice.
Concrete Subject	Classe Concreta	Oggetto reale da osservare che estende subject.
Concrete Observers	Classe Concreta	Classe concreta osservatrice del concrete subject.

Il subject non conosce minimamente cosa fanno gli observer.
Sa solo che **deve avvisare quando cambia stato**.

Il codice che trovate è di seguito rappresentato con le seguenti classi:

- **Subject astratto** → Subject.java
- **Concrete Subject** → Notifier.java

- **Interfaccia Observer** → Observer.java
- **Concrete Observers** → EmailNotification.java, SmsNotification.java
- **EventType enum** per distinguere gli eventi
- **Main** che costruisce e collega tutto

La classe **Notifier** funge da Subject e mantiene lo stato condiviso del sistema (pagamenti e numero di ordini). Gli oggetti **EmailNotification** e **SmsNotification** agiscono come osservatori concreti che reagiscono in modo differenziato ai cambiamenti del Subject. La propagazione degli aggiornamenti avviene tramite un sistema di notifiche tipizzato, basato sull'enum **EventType**, che consente un modello di notifica selettiva.

PUSH MODEL E PULL MODEL

Pull model: l'observer concreto tira lo stato dal subject usando un getter. L'osservatore richiede esplicitamente l'aggiornamento al subject.

Push model: il subject invia direttamente ai suoi osservatori il nuovo valore durante la notifica.

Il codice implementa un Observer di tipo *pull* (con un piccolo parametro "evento" passato in push all'update(), ma i dati veri vengono recuperati in pull, attraverso un get()).

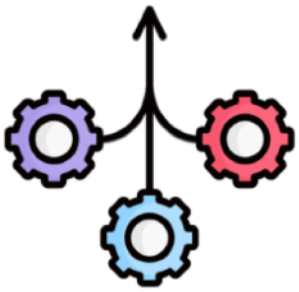
Se fosse **push**, il metodo **update** sarebbe più o meno così:

```
void update(String nuovoStatoPagamento);

// oppure

void update(EventType eventType, String statoPagamento, int numeroOrdini);
```

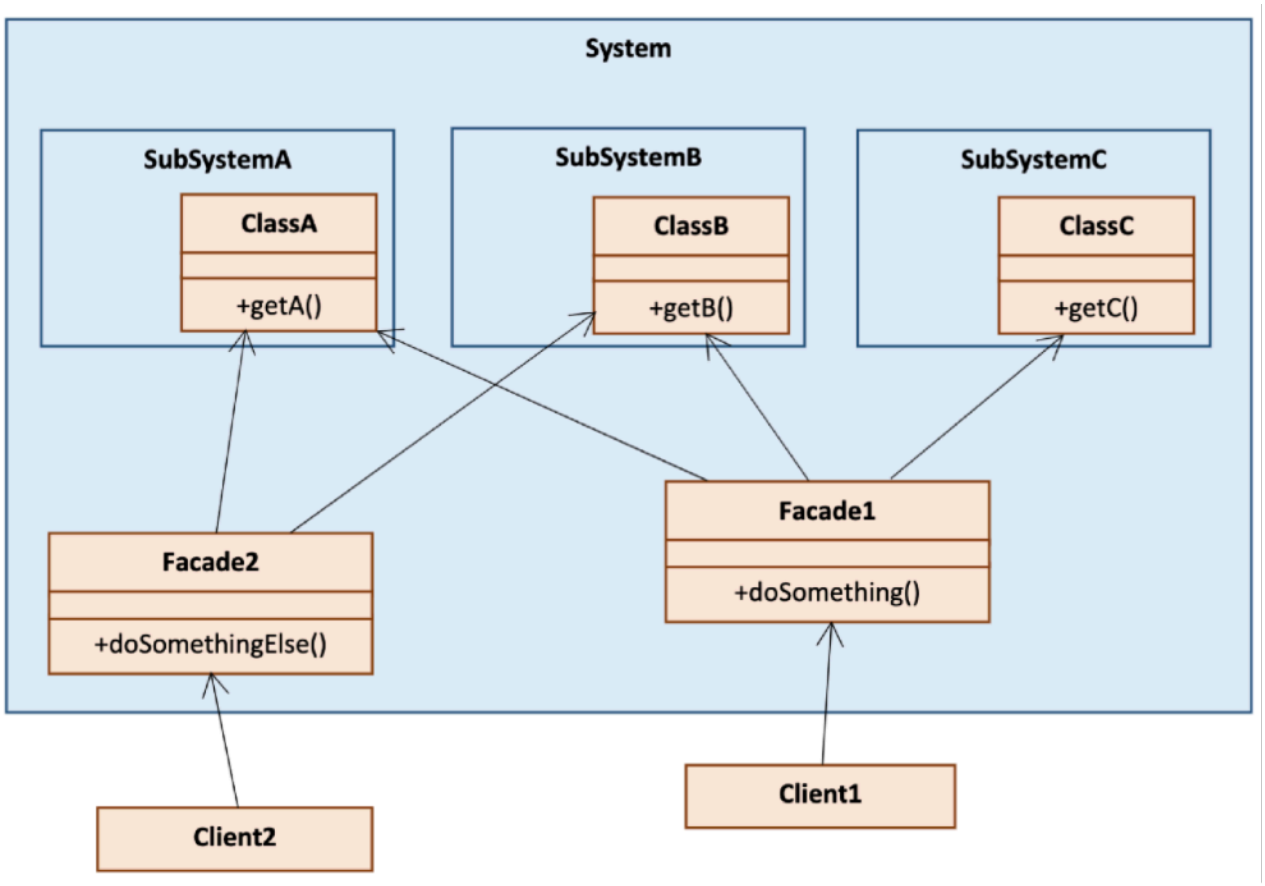
Cioè il Subject **spingerebbe (push)** i dati nuovi dentro gli Observer.



Facade

Molto brevemente, il pattern façade serve a **semplificare l'interazione con un sistema complesso, fornendo un'unica interfaccia di alto livello che nasconde i dettagli interni**.

Il client **non parla direttamente con tante classi diverse**: parla solo con una **facciata**, il facade.



Elementi di questo pattern:

Nome Elemento	Tipologia	Spiegazione
Façade	Classe Concreta	Classe che incapsula la complessità interna ed espone pochi metodi pubblici.
Subsystems	Classi Concrete	Offrono una complessità interna, ma è il façade che le coordina, e queste non sanno che il façade esiste.

Nome Elemento	Tipologia	Spiegazione
Client	Classe Concreta	Usa a facciata.

Nel codice abbiamo:

Sottosistemi (classi complicate che fanno il lavoro)

- ClsProduct : recupera i dettagli del prodotto
- ClsPayment : gestisce il pagamento
- ClsInvoice : stampa la fattura

Queste classi rappresentano il **sottosistema interno**: ognuna esegue un'operazione indipendente, e il client dovrebbe normalmente conoscerle tutte.

La Facade ClsOrder **instanzia** le classi interne, le coordina e nasconde al client la sequenza di passi necessaria.

Il main non conosce né ClsProduct, né ClsPayment, né ClsInvoice. Chiama solo il metodo della Facade e il codice resta **semplice, pulito e leggibile.**

CONCLUSIONI FINALI.

Questo documento didattico dimostra come i design pattern possano essere compresi con grande chiarezza anche tramite esempi minimi, privi di implementazioni complete.

L'uso di metodi stub, enfatizzati molto dal Prof.Falessi, rende i pattern più "leggibili", liberandoli da dettagli superflui e mettendo in evidenza solo la struttura, i ruoli e le interazioni fondamentali.

Si auspica che questa sintesi possa diventare un punto di riferimento rapido e pratico per lo studio, il ripasso e l'applicazione consapevole dei principali pattern GoF.

Questo documento è stato redatto con la collaborazione di un collega che ha contribuito alla stesura del codice e alla semplificazione degli stub.