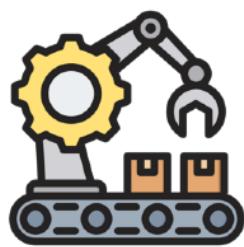




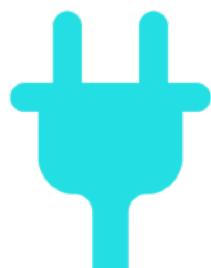
Factory Method



Abstract Factory



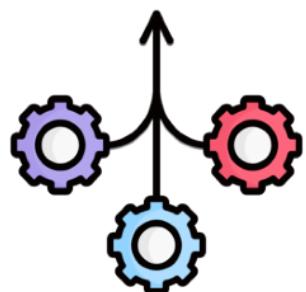
Singleton



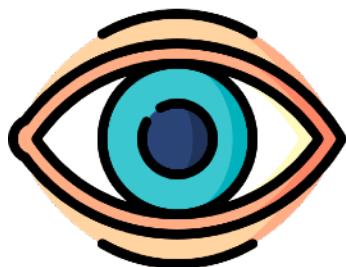
Adapter



Decorator



Facade



Observer

"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems."

(Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides — GoF, 1994)



"Un design pattern nomina, motiva e spiega in modo sistematico una soluzione generale che affronta un problema di progettazione ricorrente nei sistemi orientati agli oggetti."

(Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides — GoF, 1994)



INTRODUZIONE AI PATTERN

Attualmente abbiamo applicato il modello BCE e disponiamo di un primo abbozzo delle classi di analisi nel VOPC.

Adesso il passo successivo è raffinare.

Sarebbe utile disporre di un metodo che consenta di individuare con precisione un **problema ricorrente** e, per questo, avere a disposizione uno **schema di soluzione**. Sarebbe inoltre opportuno che tale supplemento fosse identificato univocamente da una coppia (*problema ricorrente, schema di soluzione ricorrente*).

Questo supplemento prende il nome di **pattern: soluzione consolidata ad un problema ricorrente** (tipica domanda d'esame)•

Tutte le volte che parliamo di “pattern” identifichiamo un caso a cui applichiamo una soluzione.



LA BANDA DEI QUATTRO (GoF)

Il 1994 e il 1995 furono anni fondamentali per la storia dei pattern, per la progettazione O.O e per la progettazione del software, poiché Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides pubblicarono *“Design Patterns: Elements of Reusable Object-Oriented Software”*, diventato il riferimento principale sui **design pattern** nell'ingegneria del software.

Questi quattro (da qui “**Gang of Four - GoF**” o “**Banda dei Quattro**”) hanno scritto il libro che ha avuto un impatto enorme sul modo di progettare software.

Il libro descrive **23 design pattern** ben definiti, che sono diventati un punto di riferimento nella progettazione software.



Perché “Banda dei quattro”?

Il termine “**Gang of Four**” (**Banda dei Quattro**) nasce prima in politica e solo più tardi è stato preso in prestito dall'informatica.

La **Banda dei Quattro** era un gruppo politico cinese molto influente durante la **Rivoluzione culturale** (1966-1976).

I quattro autori del libro sui design pattern non hanno nulla a che vedere con la politica, ma erano “un gruppo compatto di quattro persone” come quel famoso caso storico.

DESIGN PATTERN - ANALISI IN DETTAGLIO

Un **design pattern** è una descrizione di un problema ricorrente nella progettazione. In particolare, a ogni problema vengono associati:

- Un nome.
- Una soluzione che può essere applicata in differenti circostanze anche eterogenee tra loro.
- Una discussione sugli effetti e sulle variazioni che conseguono l'applicazione alla soluzione.

Nella pratica, l'uso e la composizione dei design pattern è quello di **formalizzare e strutturare problemi ricorrenti** e il loro scopo non è fornire nuovi spunti alla progettazione, bensì supportare l'applicazione di **tecniche consolidate**.

Propongo ora un'analogia visiva: per chi, a questo punto della lettura, non avesse ancora compreso, proviamo a rappresentare graficamente il problema.

Immagina di dover realizzare un grande **mosaico**. Hai tanti pezzetti di ceramica (le *classi* e gli *oggetti*). Potresti metterli insieme come ti pare, ma rischieresti che il mosaico venga fragile, disordinato e difficile da modificare.



Qui entrano in gioco i **design pattern**: sono schemi di disposizione ricorrenti che ti dicono:

“Se vuoi fare un bordo resistente, usa questa disposizione di pezzi”

“Se vuoi riempire un'area grande con pochi pezzi, usa quest'altro schema”

“Se vuoi fare una curva elegante, disponi i pezzi così”.

Non ti dicono **che immagine disegnare** (quello è il tuo progetto, cioè l'applicazione specifica), ma ti offrono delle **soluzioni già testate** per i problemi ricorrenti di costruzione.

In sostanza, i **design pattern** sono come **schemi di costruzione ricorrenti** che non risolvono l'intero progetto, ma ti danno **blocchi affidabili e già collaudati** per affrontare problemi comuni nella progettazione software.

I pattern GoF sono organizzati in un catalogo secondo due criteri di classificazione:

- **Scopo**: criterio di classificazione che definisce il dominio di applicazione dei pattern.

1. **Creational**: pattern che riguardano il processo di creazione di oggetti e rendono il sistema indipendente dalle modalità con cui l'istanziazione avviene.

2. **Structural**: pattern che riguardano aspetti di composizione di classi e oggetti per formare strutture complesse.

3. **Behavioral**: pattern che riguardano come classi/oggetti interagiscono per raggiungere determinati obiettivi assegnati.

- **Contesto**: criterio di classificazione che definisce la tipologia di elementi cui il pattern può essere applicato.

1. **Class**: pattern che considerano le relazioni tra classi e loro sottoclassi (struttura statica).
2. **Object**: pattern che considerano le relazioni tra oggetti modificabili a runtime (struttura dinamica).

Nella tabella seguente viene riportata una tabella riassuntiva che rappresenta il catalogo GoF con la classificazione di tutti e 23 i design pattern.

DESIGN PATTERN		SCOPO		
	CLASS	CREATIONAL	STRUCTURAL	BEHAVIORAL
CONTESTO	OBJECT	Factory Method Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Interpreter Template Method Chain of Responsibility Command Flyweight Iterator Mediator Memento Observer State Strategy Visitor

- **Pattern Creazionali & Classi**: delegano parte del processo di creazione di un oggetto alle sottoclassi.
- **Pattern Creazionali & Oggetti**: delegano parte del processo di creazione di un oggetto ad altri oggetti.
- **Pattern Strutturali & Classi**: sfruttano l'ereditarietà per comporre classi o implementazioni.
- **Pattern Strutturali & Oggetti**: descrivono modalità per raggruppare oggetti e ottenere nuove funzionalità.

- **Pattern Comportamentali & Classi:** utilizzano l'ereditarietà per definire algoritmi e flussi di controllo.
- **Pattern Comportamentali & Oggetti:** descrivono come gruppi di oggetti cooperano per svolgere compiti che un singolo oggetto non potrebbe portare a termine da solo.

Si consideri che quanto esposto finora costituisce soltanto la premessa; passiamo ora a una trattazione più analitica e dettagliata.

1. FACTORY METHOD

Il **Factory Method** è un modello di progettazione creazionale che fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma consente alle sottoclassi di alterare il tipo di oggetti che verranno creati.

Serve per spostare la logica di creazione degli oggetti **fuori** dalle classi di dominio.

Lo scopo di questo pattern è cercare di limitare l'utilizzo delle “**new**” dentro al sistema software.

Il sistema deve essere **indipendente** dalle modalità di creazione, composizione e rappresentazione dei suoi prodotti.

Si vuole una libreria (i.e. insieme di classi) che esponga soltanto l'interfaccia e non la sua implementazione.

PROBLEMA

Immagina di creare un'applicazione per la gestione della logistica. La prima versione della tua app può gestire solo il trasporto tramite camion, quindi la maggior parte del tuo codice vive all'interno della classe “Truck”.



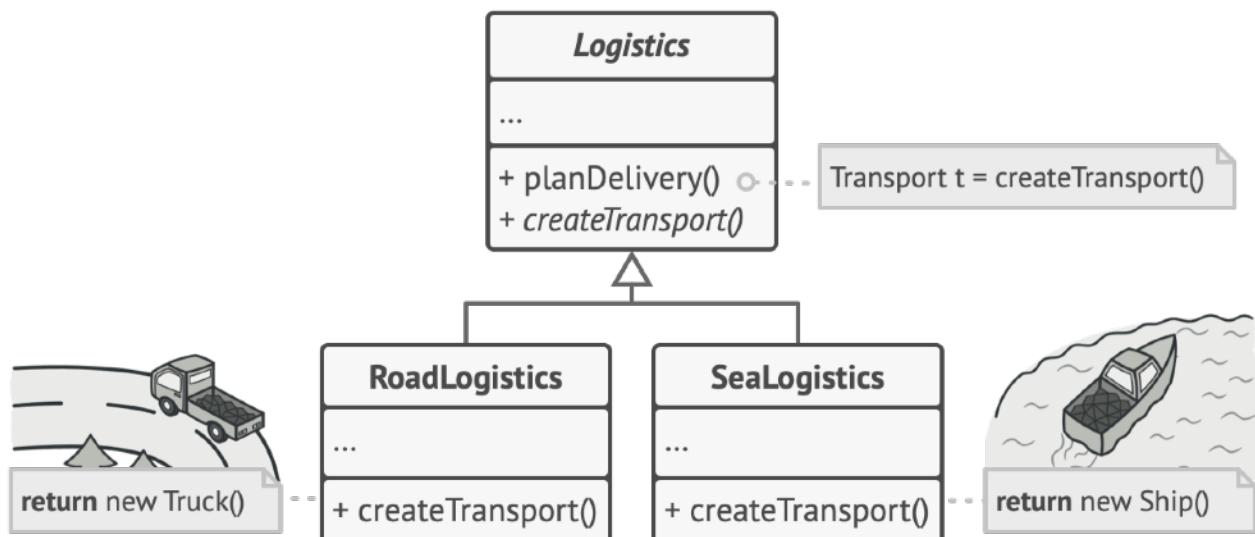
Dopo un po', la tua app diventa piuttosto popolare. Ogni giorno ricevi decine di richieste da parte delle società di trasporto marittimo per incorporare la logistica marittima nell'app.

Aggiungere una nuova classe al programma non è così semplice se il resto del codice è già accoppiato a classi esistenti. Inoltre, se in seguito decidi di aggiungere un altro tipo di trasporto all'app, probabilmente dovrà apportare nuovamente tutte queste modifiche.

SOLUZIONE

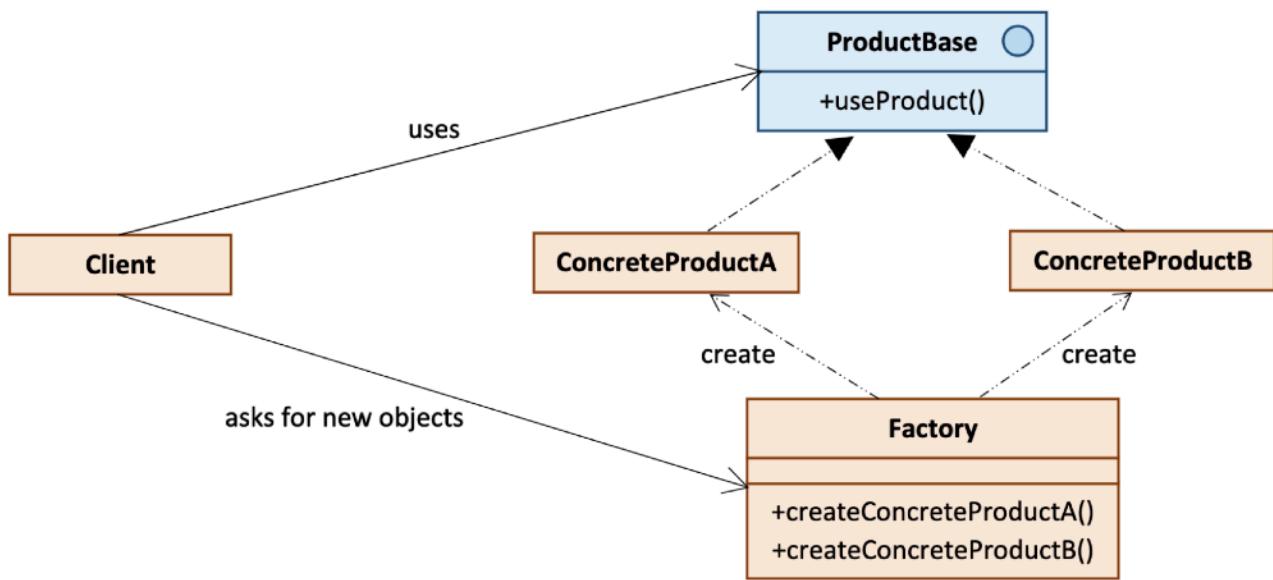
Il **Factory Method** suggerisce di sostituire le chiamate di costruzione a oggetti diretti (utilizzando il new) con chiamate ad uno speciale **Factory Method**. Non preoccuparti: gli oggetti vengono comunque creati tramite il new, ma viene chiamato dall'interno del **Factory Method**.

Gli oggetti restituiti vengono spesso definiti come *prodotti*.



A prima vista, questo cambiamento potrebbe sembrare inutile: abbiamo appena spostato la chiamata del costruttore da una parte all'altra del programma. Tuttavia, considera questo: ora puoi sovrascrivere il Factory Method in una sottoclasse e modificare la classe di prodotti creati dal metodo.

STRUTTURA - G. DE ANGELIS



- **Factory:** Contiene i metodi che si occupano di creare oggetti concreti (`createConcreteProductA()`, `createConcreteProductB()`). È la "fabbrica" che sa *come* produrre le istanze.
- **ProductBase:** È un'interfaccia o una classe astratta che dichiara le operazioni comuni a tutti i prodotti (`useProduct()` nell'esempio). Definisce cosa possono fare i prodotti, senza sapere quale tipo concreto verrà usato.
- **ConcreteProductA / ConcreteProductB:** Sono le implementazioni concrete di ProductBase. Ognuna rappresenta un tipo specifico di prodotto che la factory può restituire.
- **Client:** Non crea i prodotti direttamente con `new`, ma chiede alla factory un oggetto. Usa solo l'interfaccia `ProductBase`, senza preoccuparsi del tipo concreto.

VANTAGGI

- Disaccoppiamento: il Client non conosce i dettagli di creazione dei prodotti.
- Estendibilità: se aggiungo `ConcreteProductC`, basta modificare/estendere la factory, senza toccare il Client.
- Cohesion & Low Coupling: logica di creazione isolata, codice più ordinato.

FUNZIONAMENTO

Il Client chiede un oggetto alla Factory.

La Factory decide quale ConcreteProduct creare (es. ConcreteProductA).

Il Client riceve il prodotto come ProductBase e lo usa tramite i metodi comuni
(useProduct()).

Il libro della banda dei quattro riporta una frase che è fondamentale per questo pattern:

Nel Factory Method, il punto centrale è che **il Client non deve conoscere i dettagli delle classi concrete**, ma deve poter lavorare su un'interfaccia comune.

Nell'esempio del De Angelis, il Client usa l'interfaccia ProductBase come tipo di riferimento per lavorare con i prodotti, senza sapere se dietro c'è ConcreteProductA o ConcreteProductB.

Esempio di codice:

```
// Interfaccia comune
interface ProductBase {
    void useProduct();
}

// Implementazioni concrete
class ConcreteProductA implements ProductBase {
    public void useProduct() {
        System.out.println("Uso il prodotto A");
    }
}

class ConcreteProductB implements ProductBase {
    public void useProduct() {
        System.out.println("Uso il prodotto B");
    }
}
```

```

// Factory
class Factory {
    public static ProductBase createProduct(String type) {
        switch (type) {
            case "A": return new ConcreteProductA();
            case "B": return new ConcreteProductB();
            default: throw new IllegalArgumentException("Tipo non valido");
        }
    }
}

// Client
public class Main {
    public static void main(String[] args) {
        ProductBase p1 = Factory.createProduct("A");
        p1.useProduct(); // Output: Uso il prodotto A

        ProductBase p2 = Factory.createProduct("B");
        p2.useProduct(); // Output: Uso il prodotto B
    }
}

```

Nel **Factory Method** al posto di un'interfaccia può benissimo esserci una classe astratta.

Esempio con classe astratta al posto dell'interfaccia.

```

// Classe astratta comune
abstract class ProductBase {
    // metodo astratto da implementare nelle sottoclassi
    abstract void useProduct();

    // metodo già implementato e riutilizzabile
    public void info() {
        System.out.println("Sono un prodotto generico.");
    }
}

// Implementazioni concrete
class ConcreteProductA extends ProductBase {

```

```

@Override
void useProduct() {
    System.out.println("Uso il prodotto A");
}

}

class ConcreteProductB extends ProductBase {
@Override
void useProduct() {
    System.out.println("Uso il prodotto B");
}
}

// Factory
class Factory {
    public static ProductBase createProduct(String type) {
        switch (type) {
            case "A": return new ConcreteProductA();
            case "B": return new ConcreteProductB();
            default: throw new IllegalArgumentException("Tipo
non valido");
        }
    }
}

// Client
public class Main {
    public static void main(String[] args) {
        ProductBase p1 = Factory.createProduct("A");
        p1.useProduct(); // Uso il prodotto A
        p1.info();      // Sono un prodotto generico.

        ProductBase p2 = Factory.createProduct("B");
        p2.useProduct(); // Uso il prodotto B
        p2.info();      // Sono un prodotto generico.
    }
}

```

È VIETATO FARE “NEW” DI PRODUCT BASE.



Nella formulazione originale del pattern Factory Method si prevede un unico metodo all'interno della Factory che restituisce un solo tipo di prodotto concreto, determinato tramite un parametro passato in ingresso. Tuttavia, questa soluzione è considerata poco sicura e viene quindi sconsigliata. Per questo motivo, si preferisce la variante in cui la Factory fornisce un metodo distinto per ciascun tipo di prodotto che può essere creato.

```
// Classe astratta comune (ProductBase)
abstract class ProductBase {
    abstract void useProduct();
}

// Implementazioni concrete
class ConcreteProductA extends ProductBase {
    @Override
    void useProduct() {
        System.out.println("Uso il prodotto A");
    }
}

class ConcreteProductB extends ProductBase {
    @Override
    void useProduct() {
        System.out.println("Uso il prodotto B");
    }
}

// Factory con un metodo per ogni prodotto
class Factory {
    public ProductBase createConcreteProductA() {
        return new ConcreteProductA();
    }

    public ProductBase createConcreteProductB() {
        return new ConcreteProductB();
    }
}

// Client
public class Main {
```

```

public static void main(String[] args) {
    Factory factory = new Factory();

    ProductBase p1 = factory.createConcreteProductA();
    p1.useProduct(); // Output: Uso il prodotto A

    ProductBase p2 = factory.createConcreteProductB();
    p2.useProduct(); // Output: Uso il prodotto B
}
}

```

Di seguito vi fornisco il codice del Ricercatore Guglielmo De Angelis:

```

import it.uniroma2.dicci.ispw.factoryMethodExample.products.ProductBase;

public class Client {

    private ProductBase product1;
    private ProductBase product2;

    public Client() {
        Factory factory = new Factory();

        try{
            this.product1 = factory.createProductBase(1);
            this.product2 = factory.createProductBase(2);

        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main (String args[]){
        Client c = new Client();

        System.out.println("--- Using this.product1 ---");
        c.product1.useTheProduct();

        System.out.println("--- Using this.product2 ---");
        c.product2.useTheProduct();

    }
}

import
it.uniroma2.dicci.ispw.factoryMethodExample.products.ConcreteProductA;
import
it.uniroma2.dicci.ispw.factoryMethodExample.products.ConcreteProductA1;
import
it.uniroma2.dicci.ispw.factoryMethodExample.products.ConcreteProductB;

```

```

import
it.uniroma2.dicci.ispw.factoryMethodExample.products.ConcreteProductC;
import it.uniroma2.dicci.ispw.factoryMethodExample.products.ProductBase;

public class Factory {

    public ProductBase createProductBase(int type) throws Exception{
        switch (type)
        {
            case 1: return new ConcreteProductA();
//                case 1: return new ConcreteProductA1();
//                case 1: return new ConcreteProductC();
            case 2: return new ConcreteProductB();
            case 3: return new ConcreteProductC();
            default: throw new Exception("Invalid type : " + type);
        }
    }

    public ProductBase createProductBase(){
        // it returns the default one
        return new ConcreteProductA();
    }

    public ProductBase createConcreteProductA(){
        return new ConcreteProductA();
//        return new ConcreteProductA1();
//        return new ConcreteProductC();
    }

    public ProductBase createConcreteProductB(){
        return new ConcreteProductB();
    }
}

```

```

package it.uniroma2.dicci.ispw.factoryMethodExample.products;

public class ConcreteProductA implements ProductBase {

    @Override
    public void useTheProduct(){
        System.out.println("Here you are using: ConcreteProductA");
    }
}

```

```

package it.uniroma2.dicci.ispw.factoryMethodExample.products;

public class ConcreteProductA1 extends ConcreteProductA {

    @Override
    public void useTheProduct(){
        System.out.println("Here you are using: ConcreteProductA1");
    }
}

```

```

package it.uniroma2.dicci.ispw.factoryMethodExample.products;

```

```

public class ConcreteProductB implements ProductBase {

    @Override
    public void useTheProduct() {
        System.out.println("Here you are using: ConcreteProductB");
    }
}

package it.uniroma2.dicci.ispw.factoryMethodExample.products;

public class ConcreteProductC implements ProductBase {

    @Override
    public void useTheProduct() {
        System.out.println("Here you are using: ConcreteProductC");
    }
}

package it.uniroma2.dicci.ispw.factoryMethodExample.products;

public interface ProductBase {

    public void useTheProduct();
}

```

OUTPUT:

```

--- Using this.product1 ---
Here you are using: ConcreteProductA
--- Using this.product2 ---
Here you are using: ConcreteProductB

```

Metodo di misura: se il codice non è chiaro consiglio di non proseguire con la lettura.

Il Client non deve conoscere i dettagli delle classi concrete (ConcreteProductA, ConcreteProductB, ecc.), gli basta parlare con la Factory e ottenere oggetti che rispondono all'interfaccia comune (ProductBase).

Ripeto un concetto che dovrebbe essere assodato.

Quando nel Client il Prof scrive:

```

ProductBase p1 = factory.createProductBase(1);
p1.useTheProduct();

```

E ottengo in output:

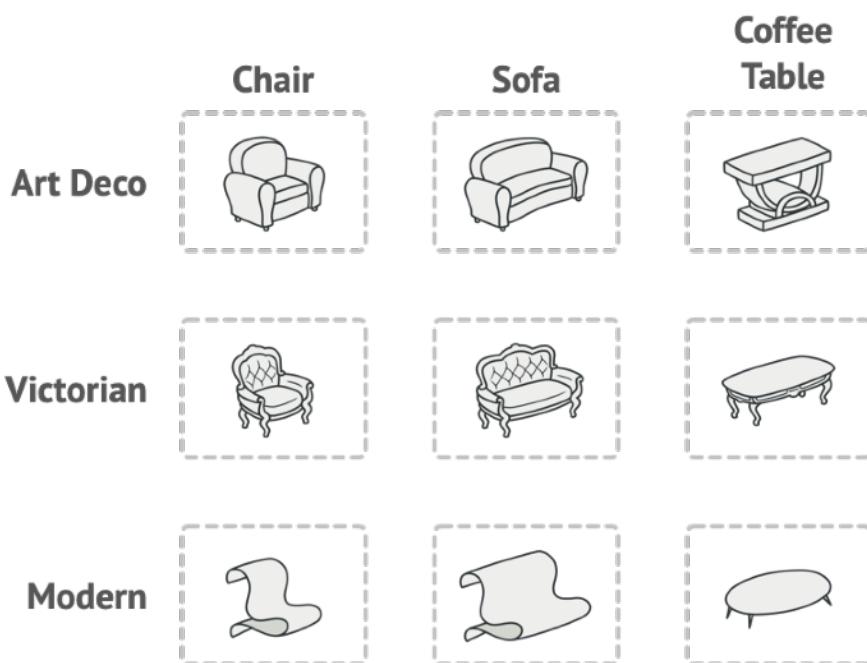
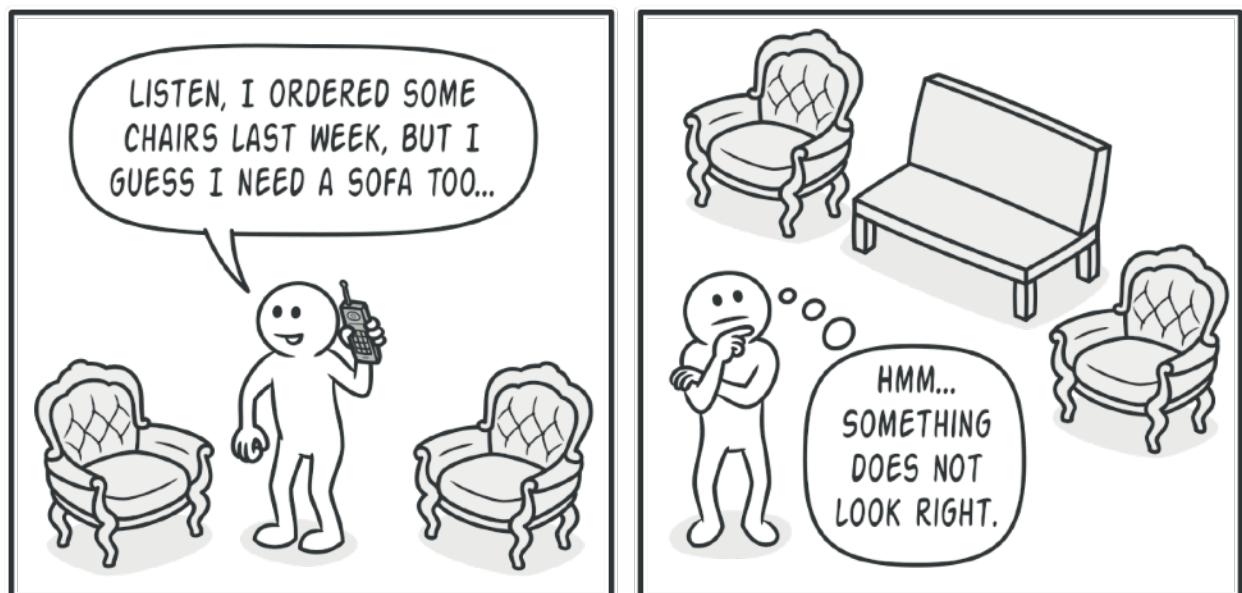
Here you are using: ConcreteProductA.

Nonostante il tipo dichiarato sia **ProductBase**, in realtà il metodo che viene eseguito è quello della **classe concreta** (ConcreteProductA).

2 . ABSTRACT FACTORY

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

Il client conosce solo l'interfaccia astratta del prodotto, non le classi concrete.



PROBLEMA

Immagina di creare un simulatore di negozio di mobili. Il tuo codice è costituito da classi che rappresentano:

1. Una famiglia di prodotti correlati: chair + sofa + CoffeeTable.
2. Diverse varianti di questa famiglia. Ad esempio, i prodotti chair + sofa + coffeetable sono disponibili in queste varianti: **Modern**, **Victorian**, **ArtDeco**.

Hai bisogno di un modo per creare singoli oggetti di arredamento in modo che corrispondano ad altri oggetti della stessa famiglia. I clienti si arrabbiano abbastanza quando ricevono mobili non corrispondenti.

SOLUZIONE

Tutte le varianti **dello stesso oggetto** devono essere spostate in un'unica gerarchia di classi.

Ad esempio, tutte le varianti di sedia possono implementare l'interfaccia "chair"; tutte le varianti di tavolino possono implementare l'interfaccia "CoffeeTable", e così via.

```
// ===== Abstract Products =====
abstract class Chair {
    abstract void sitOn();
}

abstract class Sofa {
    abstract void relaxOn();
}

abstract class CoffeeTable {
    abstract void putCoffeeOn();
}

// ===== Concrete Products: Modern =====
class ModernChair extends Chair {
    @Override
    void sitOn() {
        System.out.println("Siedo su una sedia moderna.");
    }
}

class ModernSofa extends Sofa {
    @Override
    void relaxOn() {
        System.out.println("Mi rilasso su un divano moderno.");
}
```

```

        }
    }

class ModernCoffeeTable extends CoffeeTable {
    @Override
    void putCoffeeOn() {
        System.out.println("Appoggio il caffè su un tavolino moderno.");
    }
}

// ===== Concrete Products: Victorian =====
class VictorianChair extends Chair {
    @Override
    void sitOn() {
        System.out.println("Siedo su una sedia vittoriana.");
    }
}

class VictorianSofa extends Sofa {
    @Override
    void relaxOn() {
        System.out.println("Mi rilasso su un divano vittoriano.");
    }
}

class VictorianCoffeeTable extends CoffeeTable {
    @Override
    void putCoffeeOn() {
        System.out.println("Appoggio il caffè su un tavolino vittoriano.");
    }
}

// ===== Concrete Products: ArtDeco =====
class ArtDecoChair extends Chair {
    @Override
    void sitOn() {
        System.out.println("Siedo su una sedia ArtDeco.");
    }
}

class ArtDecoSofa extends Sofa {
    @Override
    void relaxOn() {
        System.out.println("Mi rilasso su un divano ArtDeco.");
    }
}

class ArtDecoCoffeeTable extends CoffeeTable {
    @Override
    void putCoffeeOn() {
        System.out.println("Appoggio il caffè su un tavolino ArtDeco.");
    }
}

// ===== Abstract Factory =====

```

```
abstract class FurnitureFactory {
    abstract Chair createChair();
    abstract Sofa createSofa();
    abstract CoffeeTable createCoffeeTable();
}

// ===== Concrete Factories =====
class ModernFurnitureFactory extends FurnitureFactory {
    @Override
    Chair createChair() {
        return new ModernChair();
    }

    @Override
    Sofa createSofa() {
        return new ModernSofa();
    }

    @Override
    CoffeeTable createCoffeeTable() {
        return new ModernCoffeeTable();
    }
}

class VictorianFurnitureFactory extends FurnitureFactory {
    @Override
    Chair createChair() {
        return new VictorianChair();
    }

    @Override
    Sofa createSofa() {
        return new VictorianSofa();
    }

    @Override
    CoffeeTable createCoffeeTable() {
        return new VictorianCoffeeTable();
    }
}

class ArtDecoFurnitureFactory extends FurnitureFactory {
    @Override
    Chair createChair() {
        return new ArtDecoChair();
    }

    @Override
    Sofa createSofa() {
        return new ArtDecoSofa();
    }

    @Override
    CoffeeTable createCoffeeTable() {
        return new ArtDecoCoffeeTable();
    }
}
```

```

}

// ===== Client =====
public class Main {
    public static void main(String[] args) {
        // Scelgo la famiglia di mobili "Modern"
        FurnitureFactory factory = new ModernFurnitureFactory();

        Chair chair = factory.createChair();
        Sofa sofa = factory.createSofa();
        CoffeeTable table = factory.createCoffeeTable();

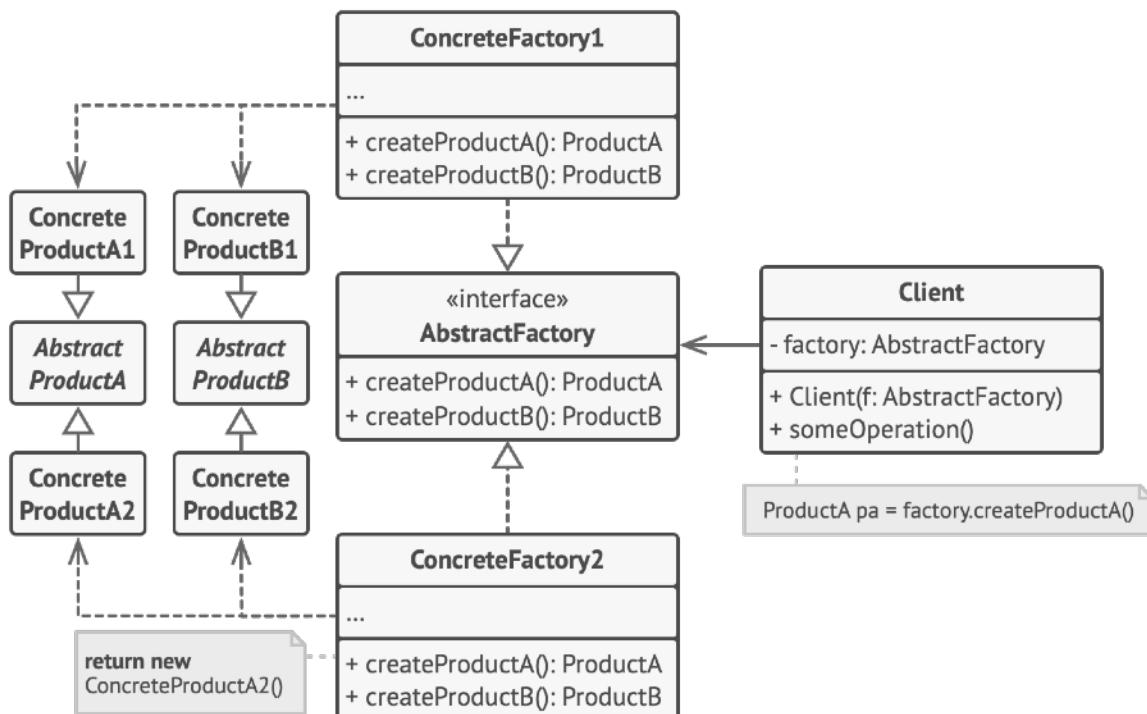
        chair.sitOn();           // Output: Siedo su una sedia moderna.
        sofa.relaxOn();         // Output: Mi rilasso su un divano moderno.
        table.putCoffeeOn();    // Output: Appoggio il caffè su un tavolino
        moderno.

        System.out.println("\n--- Cambio stile: Victorian ---");
        factory = new VictorianFurnitureFactory();
        chair = factory.createChair();
        sofa = factory.createSofa();
        table = factory.createCoffeeTable();

        chair.sitOn();           // Output: Siedo su una sedia vittoriana.
        sofa.relaxOn();          // Output: Mi rilasso su un divano
        vittoriano.
        table.putCoffeeOn();    // Output: Appoggio il caffè su un tavolino
        vittoriano.
    }
}

```

Client: chiede una FurnitureFactory e ottiene i mobili senza sapere *quale classe concreta* è stata usata.



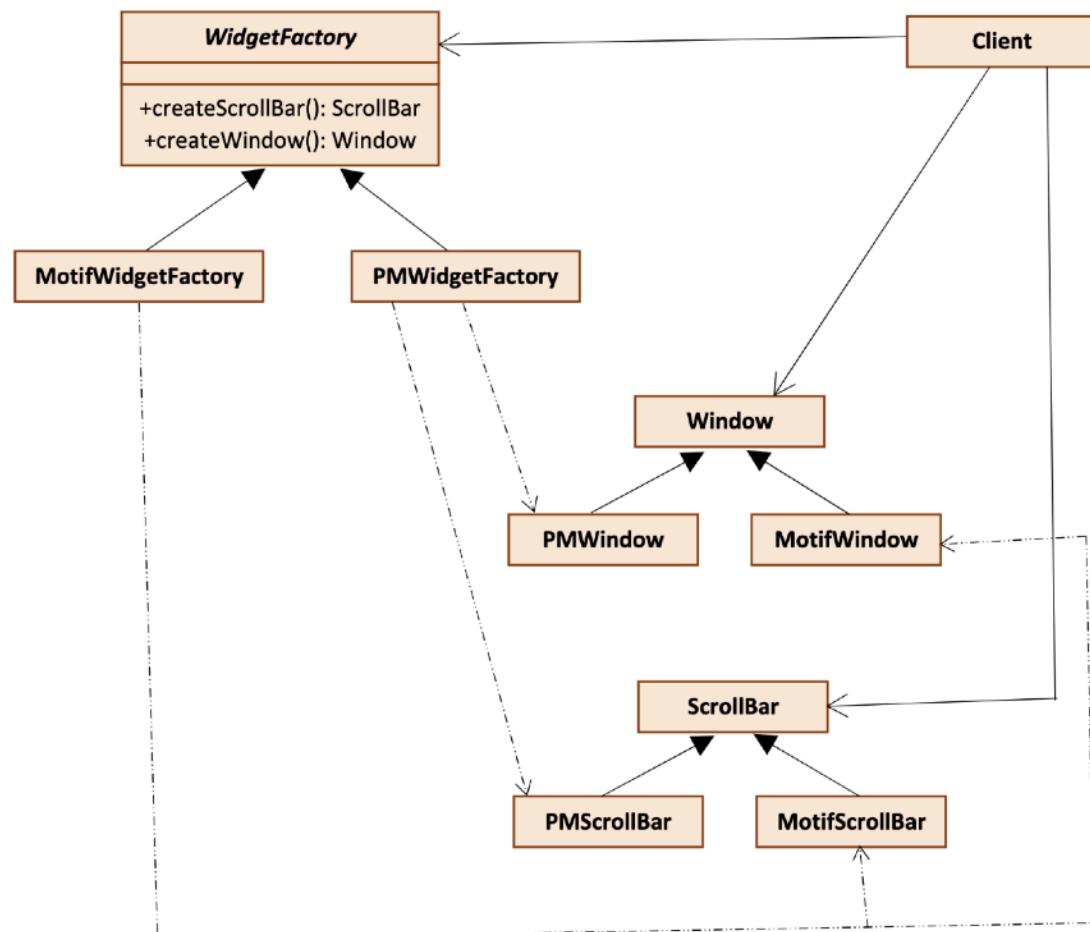
Sostanzialmente dichiaro le tre tipologie di classi (Chair, Sofa e CoffeeTable) come abstract class, questi sono **Abstract Products**.

Creo delle **sottoclassi**, **Concrete Products** che rappresentano i prodotti concreti e che estendono le classi di cui sopra. E saranno tante sottoclassi concrete per ogni prodotto astratto. Poi dichiaro la factory, **Abstract Factory**:

```
abstract class FurnitureFactory {
    abstract Chair createChair();
    abstract Sofa createSofa();
    abstract CoffeeTable createCoffeeTable();
}
```

E questa factory verrà estesa da delle **factory concrete(Concrete Factories)** per categorie, di uno stesso prodotto: **ModernFurnitureFactory**, **VictorianFurnitureFactory** e **ArtDecoFurnitureFactory**, e ciascuna contiene l'implementazione di createChair(), createSofa() e createCoffeeTable(), dove all'interno di ciascuna funzione ci sarà una new delle **sottoclassi** create prima.

STRUTTURA - G.DE ANGELIS



WidgetFactory (Abstract Factory). È la factory astratta.

Non implementa direttamente nulla: lascia alle **concrete factory** la scelta del prodotto concreto da restituire.

MotifWidgetFactory e PMWidgetFactory (Concrete Factories).

Implementano i metodi di WidgetFactory.

Ciascuna factory sa come creare **oggetti coordinati** della sua “famiglia”:

- MotifWidgetFactory crea MotifWindow e MotifScrollBar.
- PMWidgetFactory crea PMWindow e PMScrollBar.

Window e ScrollBar (Abstract Products)

- Sono le **interfacce astratte** dei prodotti.
- Definiscono i comportamenti generali che tutte le finestre o scrollbar devono avere.
- Il Client lavora solo con questi tipi astratti, non con le classi concrete.

MotifWindow / PMWindow e MotifScrollBar / PMScrollBar (Concrete Products)

- Sono le **implementazioni concrete** delle interfacce Window e ScrollBar.

Il client non conosce mai le classi concrete (MotifWindow, PMScrollBar ecc.).

- Sa solo che esiste una **WidgetFactory** da cui può chiedere una finestra e una scrollbar.
- Riceverà sempre oggetti coerenti, perché è la factory concreta che decide *quale variante* istanziare.

Esempio di codice del Prof. De Angelis:

```
package ispw.gulyx;

import ispw.gulyx.*;
import ispw.gulyx.factories.WidgetFactory;
import ispw.gulyx.scrollBar.ScrollBar;
import ispw.gulyx.window.Window;

public class Client {

    private Window w;
    private ScrollBar sb;

    public Client( WidgetFactory f ) {
        this.w = f.createWindow();
        this.sb = f.createScrollBar();
    }

    public void checkMyWindow(){
        this.w.printWhoIAm();
    }

    public void checkMyScrollBar(){
        this.sb.printWhoIAm();
    }

    public static void main (String args[]){
        Client me;
        WidgetFactory factoryInTheExample;

        //      The Motif Case
        System.out.println(" --- The Motif Case --- ");
        factoryInTheExample =
WidgetFactory.getFactory(WidgetFactory.MOTIF_LOOK_AND_FEEL);

        me = new Client(factoryInTheExample);
        me.w.printWhoIAm();
        me.sb.printWhoIAm();

        //      The Presentation Manager Case
        System.out.println(" --- The Presentation Manager Case
--- ");
    }
}
```

```
    factoryInTheExample =
WidgetFactory.getFactory(WidgetFactory.PM_LOOK_AND_FEEL);

    me = new Client(factoryInTheExample);
    me.w.printWhoIAm();
    me.sb.printWhoIAm();
}
}
```

```
package ispw.gulyx.factories;

import ispw.gulyx.*;
import ispw.gulyx.scrollBar.ScrollBar;
import ispw.gulyx.window.Window;

public abstract class WidgetFactory {

    public static final String MOTIF_LOOK_AND_FEEL = "MOTIF";
    public static final String PM_LOOK_AND_FEEL = "PM";

    public static WidgetFactory getFactory(String s) {
        if (s.equals(MOTIF_LOOK_AND_FEEL)) {
            return new MotifWidgetFactory();
        }
        return new PMWidgetFactory();
    }

    public abstract Window createWindow();
    public abstract ScrollBar createScrollBar();
}

}
```

```
package ispw.gulyx.factories;

import ispw.gulyx.scrollBar.PMScrollBar;
import ispw.gulyx.scrollBar.ScrollBar;
import ispw.gulyx.window.PMWindow;
import ispw.gulyx.window.Window;

public class PMWidgetFactory extends WidgetFactory {
```

```
@Override
public Window createWindow() {
    return new PMWindow();
}

@Override
public ScrollBar createScrollBar() {
    return new PMScrollBar();
}

}



---

package ispw.gulyx.factories;

import ispw.gulyx.*;
import ispw.gulyx.scrollBar.MotifScrollBar;
import ispw.gulyx.scrollBar.ScrollBar;
import ispw.gulyx.window.MotifWindow;
import ispw.gulyx.window.Window;

public class MotifWidgetFactory extends WidgetFactory {

    @Override
    public Window createWindow() {
        return new MotifWindow() ;
    }

    @Override
    public ScrollBar createScrollBar() {
        return new MotifScrollBar();
    }

}



---

package ispw.gulyx.scrollBar;

public abstract class ScrollBar {

    public abstract void printWhoIAm();
}



---

package ispw.gulyx.scrollBar;

public class PMScrollBar extends ScrollBar {
```

```
public void printWhoIAm(){
    System.out.println("Presentation Manager Scroll
Bar");
}

```

```
package ispw.gulyx.scrollBar;

public class MotifScrollBar extends ScrollBar {

    public void printWhoIAm(){
        System.out.println("Motif Scroll Bar");
    }
}
```

```
package ispw.gulyx.window;

public class MotifWindow extends Window {

    public void printWhoIAm(){
        System.out.println("Motif Window");
    }
}
```

```
package ispw.gulyx.window;

public class PMWindow extends Window {

    public void printWhoIAm(){
        System.out.println("Presentation Manager Window");
    }
}
```

```
package ispw.gulyx.window;

public abstract class Window {

    public abstract void printWhoIAm();
}
```

OUTPUT:

--- The Motif Case ---

Motif Window

Motif Scroll Bar

--- The Presentation Manager Case ---

Presentation Manager Window

Presentation Manager Scroll Bar

Analizzando il main notiamo subito una sottigliezza di fondamentale importanza.

Il Client riceve **una factory concreta** (MotifWidgetFactory o PMWidgetFactory).

Usa i metodi astratti createWindow() e createScrollBar() per ottenere gli oggetti.

In questo modo:

- se la factory è “Motif”, riceverà MotifWindow e MotifScrollBar;
- se la factory è “PM”, riceverà PMWindow e PMScrollBar.

Qui si vede la forza dell'**Abstract Factory**: il Client non fa mai new, delega tutto alla factory.

In pratica, il Client non ha mai bisogno di modificare il suo codice: è indipendente dalle implementazioni concrete.

3 . SINGLETION

Lo scopo di questo design pattern è quello di assicurare che **una classe abbia una sola istanza** nell'applicazione e fornire un **punto d'accesso globale a tale istanza**.

Tale istanza deve essere accessibile ai client attraverso un **punto di accesso noto a tutti gli utilizzatori**.

L'unica istanza deve poter essere estesa attraverso la definizione di sottoclassi e i client devono essere in grado di utilizzare le istanze estese senza dover modificare il proprio codice.

Tutte le implementazioni del Singleton hanno in comune questi due passaggi:

- **Rendere privato il costruttore** predefinito, per impedire ad altri oggetti di utilizzare il **new** operatore con la classe Singleton.
- **Creare un metodo di creazione statica che funga da costruttore.** Sotto il cofano, questo metodo chiama il costruttore privato per creare un oggetto e lo salva in un campo statico. Tutte le seguenti chiamate a questo metodo restituiscono l'oggetto memorizzato nella cache.

Se il tuo codice ha accesso alla classe Singleton, è in grado di chiamare il metodo statico di Singleton. Quindi, ogni volta che viene chiamato quel metodo, viene sempre restituito lo stesso oggetto.

Tieni presente che puoi sempre regolare questa limitazione e consentire la creazione di un numero qualsiasi di istanze Singleton. L'unico pezzo di codice che deve essere modificato è il corpo del metodo `GetInstance()`.

SingletonClass	1
-singletonData1 -singletonDataN <u>-instance: SingletonClass</u>	
+singletonOperation1() +singletonOperationM() <u>+getInstance(): SingletonClass</u>	

È molto utile per **servizi globali** (*logger, configurazioni, connessione DB, gestore cache...*).

Di seguito, fornisco un codice di esempio scritto da me, non dal De Angelis.

```

// Esempio di Singleton in Java: Logger (che poi vabbè è popo n'classico)
public class Logger {

    // 1. Attributo statico che contiene l'unica istanza della classe
    private static Logger instance;

    // 2. Costruttore privato: impedisce l'uso di "new Logger()"
    // dall'esterno
    private Logger() {}

    // 3. Metodo statico di accesso (lazy initialization)
    public static Logger getInstance() {
        if (instance == null) {                                // se non è ancora stato
            created                                           // lo creo la prima volta
            instance = new Logger();                          // restituisco sempre la
        }                                                       // stessa istanza
    }

    // 4. Metodo di utilità così per stampare qualcosa a caso
    public void log(String message) {
        System.out.println("[LOG] " + message);
    }

    // 5. Metodo main per testare il Singleton
    public static void main(String[] args) {
        // Ottengo due riferimenti alla stessa istanza
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("Avvio applicazione...");
        logger2.log("Operazione completata.");

        // Verifico che siano lo stesso oggetto
        System.out.println("logger1 e logger2 sono la stessa istanza? " +
                           (logger1 == logger2));
    }
}

```

Output:

```

[LOG] Avvio applicazione...
[LOG] Operazione completata.
logger1 e logger2 sono la stessa istanza? true

```

Il Metodo statico getInstance() controlla se instance è null.

Se sì, crea per la prima volta il Logger.

Se è già stato creato, restituisce sempre la stessa istanza. Questo meccanismo si chiama **lazy initialization** (l'oggetto viene creato solo al primo utilizzo).

Chiaramente, se invece togliessi **private** e mettessi il costruttore **pubblico**, allora sì che potrei fare `new Logger()`, ma a quel punto il pattern **Singleton** verrebbe **rotto**, perché potrei creare più istanze diverse.

Nel Singleton che ho mostrato il costruttore è dichiarato così:

```
private Logger() {}
```

- **private** significa che è visibile solo dentro la classe stessa.
- Dall'esterno (ad esempio dal `main`) non puoi fare `new Logger()`.

Un secondo esempio di codice fornito da me:

```
public class SingletonClass {  
  
    // Attributo statico privato che contiene l'unica istanza  
    private static SingletonClass instance;  
  
    // Attributi di esempio (singletonData1 ...  
    singletonDataN)  
    private String singletonData1;  
    private int singletonDataN;  
  
    // Costruttore privato: impedisce di creare istanze con  
    "new"  
    private SingletonClass() {  
        this.singletonData1 = "Dato iniziale";  
        this.singletonDataN = 42;  
    }  
  
    // Metodo statico che restituisce l'unica istanza  
    public static SingletonClass getInstance() {  
        if (instance == null) {  
            instance = new SingletonClass();  
        }  
        return instance;  
    }  
  
    // Operazioni pubbliche del Singleton  
    public void singletonOperation1() {  
        System.out.println("Operazione 1: " +  
    singletonData1);  
    }
```

```

    }

    public void singletonOperationM() {
        System.out.println("Operazione M: " +
singletonDataN);
    }
}

public class Main {
    public static void main(String[] args) {
        // Ottengo l'unica istanza del Singleton
        SingletonClass s1 = SingletonClass.getInstance();
        SingletonClass s2 = SingletonClass.getInstance();

        // Uso i metodi pubblici
        s1.singletonOperation1(); // Operazione 1: Dato
iniziale
        s2.singletonOperationM(); // Operazione M: 42

        // Verifica che sia la stessa istanza
        System.out.println(s1 == s2); // true
    }
}

```

È possibile specializzare il Singleton attraverso sottoclassi. Una sottoclasse può costituire la “vera” istanza singleton, ridefinendo metodi o dati.

È possibile specializzare il Singleton attraverso sottoclassi. Una sottoclasse può costituire la “vera” istanza singleton, ridefinendo metodi o dati.

Così ottieni un Singleton polimorfico: il client usa SingletonBase, ma in realtà l’istanza è una ConcreteSingletonA **o** ConcreteSingletonB.

```

// Classe base Singleton
abstract class SingletonBase {

    // unica istanza (può essere una sottoclasse)
    private static SingletonBase instance;

    // costruttore privato: impedisce "new"
    protected SingletonBase() {}

    // factory method statico
    public static SingletonBase getInstance(String type) {
        if (instance == null) {
            switch (type) {
                case "A":

```

```

        instance = new ConcreteSingletonA();
        break;
    case "B":
        instance = new ConcreteSingletonB();
        break;
    default:
        throw new IllegalArgumentException("Tipo non
supportato");
    }
}
return instance;
}

// metodo astratto da implementare nelle sottoclassi
public abstract void doSomething();
}

// Sottoclasse A del Singleton
class ConcreteSingletonA extends SingletonBase {
    @Override
    public void doSomething() {
        System.out.println("Sono il Singleton specializzato A");
    }
}

// Sottoclasse B del Singleton
class ConcreteSingletonB extends SingletonBase {
    @Override
    public void doSomething() {
        System.out.println("Sono il Singleton specializzato B");
    }
}

// Client
public class Main {
    public static void main(String[] args) {
        // La prima volta scelgo A
        SingletonBase s1 = SingletonBase.getInstance("A");
        s1.doSomething(); // Output: Sono il Singleton specializzato A

        // Occhio eh, anche se provo a chiedere "B", il Singleton è già
        stato istanziato come A. Quindi te ridà sempre A capito?
        SingletonBase s2 = SingletonBase.getInstance("B");
        s2.doSomething(); // Output: Sono il Singleton specializzato A

        // Verifica: s1 e s2 sono la stessa istanza
        System.out.println(s1 == s2); // true
    }
}

```

SingletonBase è la classe astratta che definisce il meccanismo singleton.

ConcreteSingletonA e ConcreteSingletonB sono due possibili specializzazioni.

Il metodo statico getInstance(String type) decide quale sottoclasse istanziare la prima volta.

- Dopo la prima creazione, l'istanza non cambia più (rimane coerente con la definizione di Singleton).

Il client non conosce la sottoclasse concreta: lavora sempre tramite SingletonBase.

Ma se tanto ne scelgo una e poi resta sempre quella, perché complicarmi la vita con sottoclassi del Singleton?



Il senso non è **cambiare variante a runtime** (cosa che non avrebbe senso in un Singleton).

Il senso è che puoi scegliere la variante giusta al bootstrap, e da quel momento tutta l'app usa quella.

Esempio.

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

// ===== Classe base astratta Singleton =====
abstract class Logger {

    // Istanza unica del Singleton
    private static Logger instance;

    // Costruttore protetto (evita new dall'esterno)
    protected Logger() {}

    // Metodo statico per ottenere l'istanza
    public static Logger getInstance(String type) {
        if (instance == null) {    // Lazy initialization
            switch (type) {
                case "console":
                    instance = new ConsoleLogger();
                    break;
                case "file":
                    instance = new FileLogger("log.txt");
                    break;
                default:
                    throw new IllegalArgumentException("Tipo di logger non supportato: " + type);
            }
        }
        return instance;
    }
}
```

```

        }
    }
    return instance;
}

// Metodo astratto che le sottoclassi devono implementare
public abstract void log(String message);
}

// ===== Variante Console =====
class ConsoleLogger extends Logger {
    @Override
    public void log(String message) {
        System.out.println("[ConsoleLogger] " + message);
    }
}

// ===== Variante File =====
class FileLogger extends Logger {
    private PrintWriter writer;

    public FileLogger(String filename) {
        try {
            writer = new PrintWriter(new FileWriter(filename, true)); // append
        } catch (IOException e) {
            throw new RuntimeException("Errore apertura file di log", e);
        }
    }

    @Override
    public void log(String message) {
        writer.println("[FileLogger] " + message);
        writer.flush();
    }
}

// ===== Client =====
public class Main {
    public static void main(String[] args) {
        // Scelgo la variante al bootstrap
        Logger logger1 = Logger.getInstance("console");
        logger1.log("Avvio applicazione...");

        // Provo a chiedere ancora logger (rimane la stessa istanza, console)
        Logger logger2 = Logger.getInstance("file");
        logger2.log("Questo va comunque sulla console!");

        // Verifica che sia sempre la stessa istanza
        System.out.println("logger1 == logger2 ? " + (logger1 == logger2));
    }
}

```

Output (se avvio con console)

```
[ConsoleLogger] Avvio applicazione...
[ConsoleLogger] Questo va comunque sulla console!
    logger1 == logger2 ? true
```

Output (se avvio con file)

```
[FileLogger] Avvio applicazione...
[FileLogger] Questo va comunque sul file!
    logger1 == logger2 ? true
```

Di seguito l'esempio di codice fornito dal Ricercatore Guglielmo De Angelis:

```
public class BetterSingletonClass {

    private static BetterSingletonClass instance = null;

    private int singletonData1;
    private String singletonDataN;

    protected BetterSingletonClass(int init) {
        this.singletonData1 = init;
        this.singletonDataN = "The string : " + init;
    }

    public void singletonOperation1() {
        instance.singletonData1 += 10;
        instance.singletonDataN = "Updated now to : " +
instance.singletonData1;
    }

    public String singletonOperationM() {
        return "This is " + instance.singletonDataN;
    }
}
```

```
    public synchronized static BetterSingletonClass  
getSingletonInstance() {  
    if (BetterSingletonClass.instance == null)  
        BetterSingletonClass.instance = new  
BetterSingletonClass(10);  
    return instance;  
}  
}
```

```
public class LazySingletonClass {  
  
    private int singletonData1;  
    private String singletonDataN;  
  
    /**  
     *  
     * The inner-class LazyCointainer is loaded only at the  
     * first invocation of getInstance.  
     * This activity results "thread-safe".  
     * @author gulyx  
     *  
     */  
    private static class LazyCointainer{  
        public final static LazySingletonClass  
singletonInstance = new LazySingletonClass(10);  
    }  
  
    protected LazySingletonClass(int init) {  
        this.singletonData1 = init;  
        this.singletonDataN = "The string : " + init;  
    }  
  
    public void singletonOperation1() {  
        LazyCointainer.singletonInstance.singletonData1 += 10;  
        LazyCointainer.singletonInstance.singletonDataN =  
"Updated now to : " +  
LazyCointainer.singletonInstance.singletonData1;  
    }  
  
    public String singletonOperationM() {  
        return "This is " +  
LazyCointainer.singletonInstance.singletonDataN;  
    }  
}
```

```
    public static final LazySingletonClass
getSingletonInstance() {
        return LazyCointainer.singletonInstance;
    }

}



---



```
public class SingletonClass {

 private static SingletonClass instance = null;

 private int singletonData1;
 private String singletonDataN;

 protected SingletonClass(int init) {
 this.singletonData1 = init;
 this.singletonDataN = "The string : " + init;
 }

 public void singletonOperation1() {
 instance.singletonData1 += 10;
 instance.singletonDataN = "Updeted now to : " +
instance.singletonData1;
 }

 public String singletonOperationM() {
 return "This is " + instance.singletonDataN;
 }

 public synchronized static SingletonClass
getSingletonInstance() {
 if (SingletonClass.instance == null)
 SingletonClass.instance = new SingletonClass(10);
 return instance;
 }

}

```
public class TestTheSingletonClass {

    public void testSingletonClass() {
        System.out.println("Testing the SigletonClass");

        SingletonClass s =
SingletonClass.getSingletonInstance();
```


```


```

```

        s.singletonOperation1();
        System.out.println("First reference: " +
s.singletonOperationM()));

        SingletonClass s1 =
SingletonClass.getSingletonInstance();
        s1.singletonOperation1();
        System.out.println("Second reference: " +
s1.singletonOperationM());
    }

    public void testLazySingletonClass() {
        System.out.println("Testing the LazySingletonClass");

        LazySingletonClass s =
LazySingletonClass.getSingletonInstance();
        s.singletonOperation1();
        System.out.println("First reference: " +
s.singletonOperationM());

        SingletonClass s1 =
SingletonClass.getSingletonInstance();
        s1.singletonOperation1();
        System.out.println("Second reference: " +
s1.singletonOperationM());
    }

    public static void main(String[] args) {
        TestTheSingletonClass test = new
TestTheSingletonClass();

        test.testSingletonClass();
//        test.testLazySingletonClass();

    }
}

```

Output:

```

Testing the SingletonClass
First reference: This is Updated now to : 20
Second reference: This is Updated now to : 30

```

Occhio, se sul main levo il commento :

```
// test.testLazySingletonClass();
```

E la inserisco come istruzione, l'output diventa:

```
Testing the SingletonClass
First reference: This is Updeted now to : 20
Second reference: This is Updeted now to : 30
Testing the LazySingletonClass
First reference: This is Updeted now to : 20
Second reference: This is Updeted now to : 40
```

Tutte e tre le classi fanno la stessa cosa: garantiscono un'istanza unica. La differenza sta nel *come* lo fanno: più semplice ma meno efficiente (con synchronized), oppure più elegante (con inner class).

SingletonClass : È la versione più semplice.

- ha una variabile instance che contiene l'oggetto unico,
- il costruttore è protetto, quindi non puoi fare new da fuori,
- un metodo getSingletonInstance() controlla se l'istanza esiste: se no la crea, se sì restituisce sempre quella.

In più è synchronized, cioè sicuro anche se più thread provano a creare l'istanza contemporaneamente.

È la forma “classica”, funziona sempre, ma ogni chiamata ha un piccolo costo dovuto alla sincronizzazione.

LazySingletonClass : Questa è la versione più elegante e moderna.

L'idea è usare una **inner class statica** (LazyContainer) che contiene l'istanza unica.

In Java, le inner class statiche vengono caricate solo quando servono, e il caricamento è garantito thread-safe dalla JVM.

Risultato: l'istanza viene creata solo la prima volta che qualcuno chiama getSingletonInstance(), non serve synchronized, è efficiente e pulito.

È considerata la versione migliore quando lavori in Java.

Dove sta l'unicità?

- L'unicità sta nel fatto che singletonInstance è static final.

- In Java, un campo static final viene inizializzato **una sola volta** al caricamento della classe, e non può mai cambiare.
- Quindi, anche se chiami `getInstance()` 1000 volte, ricevi sempre lo stesso riferimento.

Quindi:

```
LazySingletonClass s1 = LazySingletonClass.getInstance();
```

```
LazySingletonClass s2 = LazySingletonClass.getInstance();
```

```
System.out.println(s1 == s2); // true
```

Stessa istanza. Singleton approva.

Il Singleton è come una relazione esclusiva: una sola istanza valida, accessibile sempre nello stesso modo, e **qualsiasi cambiamento riguarda tutti i riferimenti a quella stessa persona.**

Problema del threading nel Singleton

Se hai un'implementazione “pigra/non sincronizzata” come:

```
public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
```

In un ambiente con **più thread**, succede che:

1. **Thread A** entra in `getInstance()`, vede che `instance == null`, e sta per creare l'istanza.
2. Prima che completi `new Singleton()`, viene sospeso.

3. **Thread B** entra nello stesso metodo, vede ancora `instance == null` (perché A non ha ancora finito l'inizializzazione), quindi crea un'altra istanza.
4. Ora hai **due oggetti diversi** creati, violando la proprietà del Singleton.

Soluzione: Metodo `synchronized` su `getInstance()`, garantisce che solo un thread alla volta entri nella creazione, oppure il De Angelis usa una soluzione lazy.

La soluzione “lazy” con inner class statica

(LazySingletonClass) è uno dei modi più eleganti e sicuri per evitare i problemi di thread.

In Java, il caricamento di una classe **interna statica** (`LazyHolder`) avviene **solo la prima volta che qualcuno la usa** e, cosa importantissima, il caricamento delle classi è **thread-safe** per definizione.

ABSTRACT FACTORY E SINGLETON - AMICI PER LA PELLE



Il problema è: quante volte devo creare la factory?

```
// ===== Prodotti astratti =====
abstract class Sofa {
    public abstract void sitOn();
}

abstract class Chair {
    public abstract void use();
}

// ===== Prodotti concreti: Modern =====
class ModernSofa extends Sofa {
    @Override
    public void sitOn() {
        System.out.println("Siedo su un divano moderno");
    }
}
class ModernChair extends Chair {
    @Override
    public void use() {
        System.out.println("Uso una sedia moderna");
    }
}
```

```

// ===== Prodotti concreti: Victorian =====
class VictorianSofa extends Sofa {
    @Override
    public void sitOn() {
        System.out.println("Siedo su un divano vittoriano");
    }
}

class VictorianChair extends Chair {
    @Override
    public void use() {
        System.out.println("Uso una sedia vittoriana");
    }
}

// ===== Abstract Factory =====
abstract class FurnitureFactory {
    public abstract Sofa createSofa();
    public abstract Chair createChair();

    // ---- Parte Singleton ----
    private static FurnitureFactory instance;

    // Metodo statico per ottenere l'unica factory globale
    public static FurnitureFactory getFactory(String style) {
        if (instance == null) {
            switch (style) {
                case "modern":
                    instance = new ModernFurnitureFactory();
                    break;
                case "victorian":
                    instance = new VictorianFurnitureFactory();
                    break;
                default:
                    throw new IllegalArgumentException("Stile non
supportato");
            }
        }
        return instance; // sempre la stessa istanza
    }
}

// ===== Concrete Factories =====
class ModernFurnitureFactory extends FurnitureFactory {
    @Override
    public Sofa createSofa() { return new ModernSofa(); }
    @Override
    public Chair createChair() { return new ModernChair(); }
}

class VictorianFurnitureFactory extends FurnitureFactory {
    @Override
    public Sofa createSofa() { return new VictorianSofa(); }
    @Override
    public Chair createChair() { return new VictorianChair(); }
}

```

```

// ===== Client =====
public class Main {
    public static void main(String[] args) {
        // Scelgo lo stile una volta sola (es. "modern")
        FurnitureFactory factory = FurnitureFactory.getFactory("modern");

        // Creo mobili coordinati
        Sofa sofa = factory.createSofa();
        Chair chair = factory.createChair();

        sofa.sitOn();    // Output: Siedo su un divano moderno
        chair.use();    // Output: Uso una sedia moderna

        // Se chiedo ancora la factory, è sempre la stessa istanza
        FurnitureFactory sameFactory = FurnitureFactory.getFactory("victorian");
        System.out.println(factory == sameFactory); // true
    }
}

```

FurnitureFactory è l'**Abstract Factory** : ha i metodi per creare sofa e sedie.

In Main, se scegli "modern", tutta l'applicazione userà mobili moderni.

Anche se più avanti chiedi "victorian", ormai la factory è stata scelta e resta sempre la stessa.

VERSIONE CON LA INNER CLASS

```

// ===== Prodotti astratti =====
abstract class Sofa {
    public abstract void sitOn();
}

abstract class Chair {
    public abstract void use();
}

// ===== Prodotti concreti: Modern =====
class ModernSofa extends Sofa {
    @Override
    public void sitOn() {
        System.out.println("Siedo su un divano moderno");
    }
}
class ModernChair extends Chair {
    @Override
    public void use() {

```

```

        System.out.println("Uso una sedia moderna");
    }
}

// ===== Prodotti concreti: Victorian =====
class VictorianSofa extends Sofa {
    @Override
    public void sitOn() {
        System.out.println("Siedo su un divano vittoriano");
    }
}
class VictorianChair extends Chair {
    @Override
    public void use() {
        System.out.println("Uso una sedia vittoriana");
    }
}

// ===== Abstract Factory =====
abstract class FurnitureFactory {
    public abstract Sofa createSofa();
    public abstract Chair createChair();

    // Factory concreta scelta una volta sola (Singleton)
    private static FurnitureFactory instance;

    // Lazy inner class: garantisce inizializzazione thread-
    // safe e on-demand
    private static class FactoryHolder {
        private static FurnitureFactory INSTANCE;
    }

    // Metodo statico per ottenere la factory
    public static FurnitureFactory getFactory(String style) {
        if (FactoryHolder.INSTANCE == null) {
            switch (style) {
                case "modern":
                    FactoryHolder.INSTANCE = new
ModernFurnitureFactory();
                    break;
                case "victorian":
                    FactoryHolder.INSTANCE = new
VictorianFurnitureFactory();
                    break;
                default:

```

```

        throw new IllegalArgumentException("Stile
non supportato");
    }
}
return FactoryHolder.INSTANCE;
}

// ===== Concrete Factories =====
class ModernFurnitureFactory extends FurnitureFactory {
    @Override
    public Sofa createSofa() { return new ModernSofa(); }
    @Override
    public Chair createChair() { return new ModernChair(); }
}

class VictorianFurnitureFactory extends FurnitureFactory {
    @Override
    public Sofa createSofa() { return new VictorianSofa(); }
    @Override
    public Chair createChair() { return new VictorianChair(); }
}

// ===== Client =====
public class Main {
    public static void main(String[] args) {
        // Scelgo lo stile una volta sola (es. "modern")
        FurnitureFactory factory =
FurnitureFactory.getFactory("modern");

        Sofa sofa = factory.createSofa();
        Chair chair = factory.createChair();

        sofa.sitOn();    // Output: Siedo su un divano moderno
        chair.use();     // Output: Uso una sedia moderna

        // Se provo a cambiare stile più avanti, rimane
        sempre lo stesso
        FurnitureFactory sameFactory =
FurnitureFactory.getFactory("victorian");
        System.out.println(factory == sameFactory); // true
    }
}

```

La parte Singleton è gestita da una inner class statica (FactoryHolder) che contiene l'istanza unica.

Questa inner class viene caricata solo quando chiamo `getFactory()`.

4 . ADAPTER



Spesso le classi di un sistema vengono strutturate/progettate cercando di offrire un profilo **riusabile**. Tuttavia, può capitare che queste classi **non possono essere effettivamente riusate**; per esempio perché la loro interfaccia offerta non rispecchia esattamente i requisiti (o la segnatura) richiesti di uno specifico dominio.

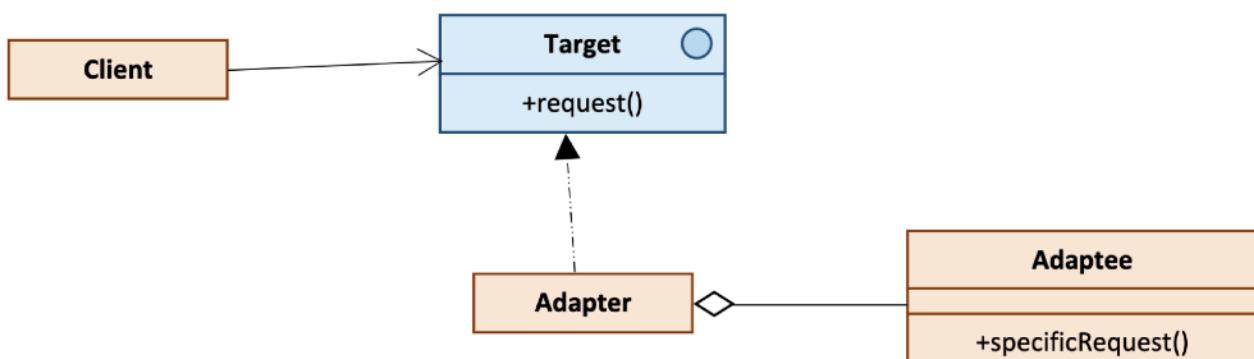
Lo scopo è gestire interfacce incompatibili, fornire un’interfaccia stabile a classi funzionalmente simili o con interfacce diverse.

Si vuole utilizzare una classe esistente ma la sua interfaccia non è compatibile con quella che serve.

Si vuole realizzare una classe **riusabile** che coopera con altre classi anche se scorrelate o impreviste e con una interfaccia eventualmente incompatibile.

Analogia con la corrente: È come una presa elettrica universale, hai un dispositivo con spina americana, la tua presa a muro è europea, metti un adattatore in mezzo, e tutto funziona senza cambiare né la spina né la presa.

In sintesi, il pattern Adapter serve ad adattare un'interfaccia incompatibile a quella che ti aspetti. In altre parole, ti permette di **usare una classe esistente** (che non puoi o non vuoi modificare) come se implementasse un'altra interfaccia.



Il Client vuole usare Target con il metodo request().

Ma la classe che già esiste (Adaptee) ha solo un metodo specificRequest(), quindi l'interfaccia non coincide.

L'Adapter fa da ponte: implementa Target e al suo interno chiama specificRequest() di Adaptee.

```
// Interfaccia Target che il Client si aspetta
interface Target {
    void request();
}

// Classe esistente con interfaccia incompatibile (Adaptee)
class Adaptee {
    public void specificRequest() {
        System.out.println("Chiamata a specificRequest() dell'Adaptee");
    }
}

// Adapter: adatta l'Adaptee al Target
class Adapter implements Target {
    private Adaptee adaptee; // composizione

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    // Traduce la chiamata del Client nella chiamata giusta
    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

// Client: usa solo l'interfaccia Target
public class Main {
    public static void main(String[] args) {
        // Il client lavora con Target, adatta adaptee al target
        Target target = new Adapter(new Adaptee());

        // Ma in realtà l'Adapter traduce la chiamata verso Adaptee
        target.request();
    }
}

Output:
Chiamata a specificRequest() dell'Adaptee
```

Il Client sa usare solo Target.request().

L'oggetto utile (Adaptee) ha però solo specificRequest().

L'Adapter implementa Target, così il Client lo accetta, ma al suo interno richiama specificRequest() dell'Adaptee.

Risultato: il Client lavora come se fosse tutto compatibile, senza sapere nulla del codice legacy.

I Clients invocano operazioni su una istanza di Adapter. A sua volta l'istanza di Adapter gestisce opportunamente l'invocazione verso le istanze di Adaptee.

ESEMPIO DEL PROF. DE ANGELIS

```
public interface Jewel {  
  
    /*  
     * It returns the carats of the jewel  
     */  
    public int weight();  
  
    /*  
     * It returns true if the Jewel have some value  
     */  
    public boolean hasAValue();  
  
    /*  
     * It returns true if the Jewel is Natural and from EU.  
     * It returns false if it is not, or not specified.  
     */  
    public boolean isEuropeanNatual();  
}
```

```
public class Adapter implements Jewel {  
  
    private Stone s;  
    private final float TINY_STONE_OVERALL_VOLUME_IN_MM3 = 0.125f;  
    public Adapter(Stone s) {  
        this.s = s;  
    }  
  
    @Override  
    public int weight() {  
        return Math.round((this.s.weightInKilos()*1000) * 5);  
    }  
  
    @Override  
    public boolean hasAValue() {  
        StoneKinds kind = this.s.getStoneKind();  
    }  
}
```

```
        return ((kind == StoneKinds.Diamond) || (kind ==  
StoneKinds.Ruby));  
    }  
  
    /**  
     * Note that in this method the Adapter has several interactions  
(i.e., 3)  
     * with the Adaptee (i.e. Stone) in order to realize a functionality  
     */  
    @Override  
    public boolean isEuropeanNatural() {  
        boolean outcome = false;  
  
        if ( (this.s.isOriginKnown()) && (!this.s.madeInLaboratory()) ) {  
            Countries c = this.s.comesFrom();  
            switch (c) {  
                case Italy:  
                case Germany:  
                case CzechRepublic:  
                    outcome = true;  
                    break;  
                default:  
                    outcome = false;  
                    break;  
            }  
        }  
  
        return outcome;  
    }  
}  
  
-----  
public enum Countries {  
    Italy, SouthAfrica, China, Russia, Germany, CzechRepublic  
}  
-----  
import java.util.Random;  
  
public class Foo {  
    private float weight;  
    private StoneKinds kind;  
  
    public Foo() {  
        Random r = new Random();  
        this.weight = r.nextInt(50);  
        int nKinds = StoneKinds.values().length;  
        if (nKinds!=0)  
            this.kind = StoneKinds.values()[r.nextInt(nKinds)];  
        else  
            this.kind = null;  
    }  
  
    public Foo(StoneKinds kind, float weightInKilos) {  
        this.weight = weightInKilos;  
        this.kind = kind;  
    }  
}
```

```
}

public float boo() {
    return this.weight;
}

public StoneKinds goo() {
    return this.kind;
}
}

public class Origin {

    private Countries country;
    private boolean synth;

    public Origin(Countries country, boolean isSynth){
        this.country = country;
        this.synth = isSynth;
    }

    public Countries originFrom() {
        return this.country;
    }

    public boolean isNatural() {
        return !this.synth;
    }
}

import java.util.Random;

public class Stone {
    private float weight;
    private StoneKinds kind;
    private Origin origin;

    public Stone() {
        Random r = new Random();

        this.weight = r.nextInt(50);

        this.kind = this.generateRandomicStoneKind(r);

        this.origin = this.generateRandomicOrigin(r);
    }

    public Stone(StoneKinds kind, float weightInKilos) {
        this(kind, weightInKilos, null);

        Random r = new Random();
        this.origin = this.generateRandomicOrigin(r);
    }

    public Stone(StoneKinds kind, float weightInKilos, Origin origin) {
```

```

        this.weight = weightInKilos;
        this.kind = kind;
        this.origin = origin;
    }

    public float weightInKilos() {
        return this.weight;
    }

    public StoneKinds getStoneKind() {
        return this.kind;
    }

    public boolean isOriginKnown() {
        return this.origin != null;
    }

    public boolean madeInLaboratory() {
        if (this.origin == null) {
            return false;
        }
        return !this.origin.isNatural();
    }

    public Countries comesFrom() {
        if (this.origin == null) {
            return null;
        }
        return this.origin.originFrom();
    }

    private StoneKinds generateRandomicStoneKind(Random r) {
        StoneKinds k = null;
        int nKinds = StoneKinds.values().length;
        if (nKinds!=0) {
            k = StoneKinds.values()[r.nextInt(nKinds)];
        }
        return k;
    }

    private Origin generateRandomicOrigin(Random r) {
        Origin o = null;
        int nCountries = Countries.values().length;
        if (nCountries!=0) {
            Countries c = Countries.values()[r.nextInt(nCountries)];
            o = new Origin(c, r.nextBoolean());
        }
        return o;
    }
}

public enum StoneKinds {
    Diamond, Marble, Ruby, Granite
}

import java.util.Vector;

```

```

public class Jeweller {

    private Vector<Jewel> luxuryJewels;
    private Vector<Jewel> natualEUJewels;
    private int minCarats;

    public Jeweller( int thresholdInCarats ) {
        this.luxuryJewels = new Vector<Jewel>();
        this.natualEUJewels = new Vector<Jewel>();
        this.minCarats = thresholdInCarats ;
    }

    private boolean evaluate (Jewel j){
        return ((j.hasAValue()) && (j.weight() >= this.minCarats));
    }

    private boolean evaluateEU (Jewel j){
        return ((j.hasAValue()) && (j.isEuropeanNatual()));
    }

    public boolean buyLuxuryJewels (Jewel j){
        if (this.evaluate(j))
            return this.luxuryJewels.add(j);
        return false;
    }

    public boolean buyEUJewels (Jewel j){
        if (this.evaluateEU(j))
            return this.natualEUJewels.add(j);
        return false;
    }

    public static void main(String args[] ) {
        Jeweller me = new Jeweller(24);

        // Create a Stone
        Stone s1 = new Stone();
        // Adapt the Stone as Jewel
        Jewel a = new Adapter(s1);

        // The Jeweller tries to by the "luxury stone" :-D
        System.out.println("Let's by a \"luxury stone\"");
        if (!me.buyLuxuryJewels(a))
            System.out.println(" - That Jewel was not good for me!!!!");
        else
            System.out.println(" - Yes!!!! I got it!");

        // The Jeweller tries to by a natural "stone" from the EU :-D
        System.out.println("Is it a natural \"stone\" from the EU");
        if (!me.buyEUJewels(a))
            System.out.println(" - That Jewel it not natural or it is not
from the EU !!!");
        else
            System.out.println(" - Yes, it is!");
    }
}

```

```
}
```

```
        Output:  
        Let's buy a "luxury stone"  
            - Yes!!!! I got it!  
        Is it a natural "stone" from the EU  
            - Yes, it is!
```

Chi fa chi!

Client = Jeweller
Target (interfaccia attesa) = Jewel
Adaptee (classe esistente) = Stone
Adapter (ponte tra i due) = Adapter

Il sistema conosce e gestisce solo l'**interfaccia Jewel** (gioiello).

L'Adapter implementa l'interfaccia.

Stone rappresenta una pietra, con metodi diversi, ma non parla il linguaggio di Jewel.
Il Jeweller può trattare una Stone come Jewel, perché l'Adapter traduce le chiamate.

L'Adapter traduce i metodi di Jewel in chiamate ai metodi di Stone.

In questo modo puoi **riutilizzare Stone senza toccarla**, e usarla in un contesto dove serve un Jewel.

Sequenza:

Avvio (metodo **main** in **Jeweller**): Crea un gioielliere.
Inizializza due liste: **luxuryJewels** e **naturalEUJewels**.
minCarats di Jeweller = 24.

Dopodiché Creazione della pietra (Adaptee).

Costruttore **random** della Stone:

- **weight**: intero casuale salvato come **float**.

- kind: uno tra Diamond, Marble, Ruby, Granite.

Per settare kind si usa:

```
private StoneKinds generateRandomicStoneKind(Random r) {
    StoneKinds k = null;
    int nKinds = StoneKinds.values().length; //4
    if (nKinds != 0) {
        k = StoneKinds.values()[r.nextInt(nKinds)];
    }
    return k;
}
```

Questa funzione sceglie casualmente un tipo di pietra dall'enum StoneKinds.

Nel nostro caso, può restituire a caso: Diamond, Marble, Ruby o Granite.

- origin: paese casuale (Countries) + booleano sintetico/naturale (tramite Origin).

Per dettare Origin si usa:

```
private Origin generateRandomicOrigin(Random r) {
    Origin o = null;
    int nCountries = Countries.values().length;
    if (nCountries != 0) {
        Countries c = Countries.values()
[r.nextInt(nCountries)];
        o = new Origin(c, r.nextBoolean());
    }
    return o;
}
```

Questa funzione seleziona casualmente Italy, SouthAfrica, China, Russia, Germany o CzechRepublic e la assegna ad un enumeratore Countries, chiamato c. E sarà uno di questi paesi.

Successivamente si crea un nuovo oggetto Origin con i dati estratti: o = new Origin(c, r.nextBoolean());

Il secondo parametro estrae un valore booleano casuale (true/false).

E quindi abbiamo dettato anche origin tramite una classe. Prima avevamo dettato StoneKinds tramite numeratore.

Ora li abbiamo tutti i dati della pietra.

Adesso c'è l'adattamento a Jewel (Adapter).

La stone creata è s1.

Jewel a = new Adapter(s1);

Ora il gioielliere non parla più con Stone, ma con l'interfaccia Jewel.

L'Adapter memorizza il riferimento a s1.

- Da ora in poi il Client userà "a" come se fosse un Jewel, mentre l'Adapter traduce tutto su Stone.

Ricorda: il parametro che va dentro l'adapter, è proprio l'adaptee. In questo modo l'adapter adatta **Stone = adaptee** a Jewel.

```
Stone s1 = new Stone();      // creo la pietra  
Jewel a = new Adapter(s1);  // adatto la pietra a Jewel
```

È qui che gli fai capire che dietro quel Jewel c'è proprio quella Stone. Quell'oggetto "a" è in realtà un Adapter che contiene un riferimento alla Stone che il prof ha passato (s1).

Adesso si fa un Tentativo di acquisto "Luxury".

```
System.out.println("Let's buy a \"luxury stone\"");
```

Flusso interno nel client:

1. buyLuxuryJewels → chiama evaluate(j), una funzione sempre nel client:

- j.hasAValue() → va su Adapter.hasAValue(), ora si può fare, su una pietra stone che non conosceva niente di Jewel → che a sua volta chiama s.getStoneKind():

- true solo se **Diamond o Ruby**; false per Marble/Granite.

Il gioielliere prova a comprarla:

```
if (!me.buyLuxuryJewels(a)) ...
```

Dentro `buyLuxuryJewels` succede questo:

- Chiama `evaluate(j)`:

```
return (j.hasAValue() && j.weight() >= this.minCarats);
```

Ed ecco che entra in gioco l'Adapter:

```
public boolean hasAValue() {  
    StoneKinds kind = this.s.getStoneKind();  
    return ((kind == StoneKinds.Diamond) || (kind ==  
StoneKinds.Ruby));  
}
```

Qui l'Adapter chiede al **Stone** che tipo è (`getStoneKind()`):

- Se è Diamond o Ruby → `hasAValue()` restituisce **true**.
- Se è Marble o Granite → restituisce **false**.

Quindi sì: in questo punto capisce che pietra ho.

L'Adapter traduce la richiesta `hasAValue()` del client in una chiamata a `s.getStoneKind()` dell'Adaptee (**Stone**).

Quindi, quando il gioielliere (**Jeweller**) chiama: `a.hasAValue();`
`a` è un **Adapter** (che implementa **Jewel**). Dentro il metodo, l'**Adapter** usa la **Stone** memorizzata nel costruttore (`this.s`), cioè proprio quella che gli hai passato (`s1`). Quindi “capisce” che pietra ho.

- `j.weight()` → va su `Adapter.weight()` → che chiama `s.weightInKilos()` e converte:
 - $\text{carati} = \text{round}(\text{kg} * 1000 * 5)$ (1 g ≈ 5 carati).

- Se **entrambi** (valore prezioso e peso ≥ 24 carati) sono veri \rightarrow l'oggetto viene aggiunto a luxuryJewels e stampa "Yes!!! I got it!".
Altrimenti stampa "not good".

MORALE: LA PIETRA (STONE) NON CONOSCEVA L'INTERFACCIA JEWEL .

È l'adapter che conosce l'interfaccia.

L'ADAPTER HA IMPLEMENTATO L'INTERFACCIA E HA FATTO ESEGUIRE I METODI DELL'INTERFACCIA SULLA PIETRA.

In questi metodi chiama, ad esempio, `this.m.getStoneKind()`, e trova la pietra di s1.

Penso sia tutto chiaro.

Ed è proprio questo il cuore del pattern Adapter: **adattare una classe esistente con interfaccia incompatibile** ad un'interfaccia che il client si aspetta, senza modificare la classe originale.

Nella pratica, questo pattern è utilissimo quando si vuole utilizzare una classe esistente ma la sua interfaccia non è compatibile con quella che serve, perché magari si vuole fruire di questa interfaccia e dei relativi servizi.

Proseguiamo con i pattern.

5. DECORATOR

È uno dei pattern *fondamentali* della GoF.

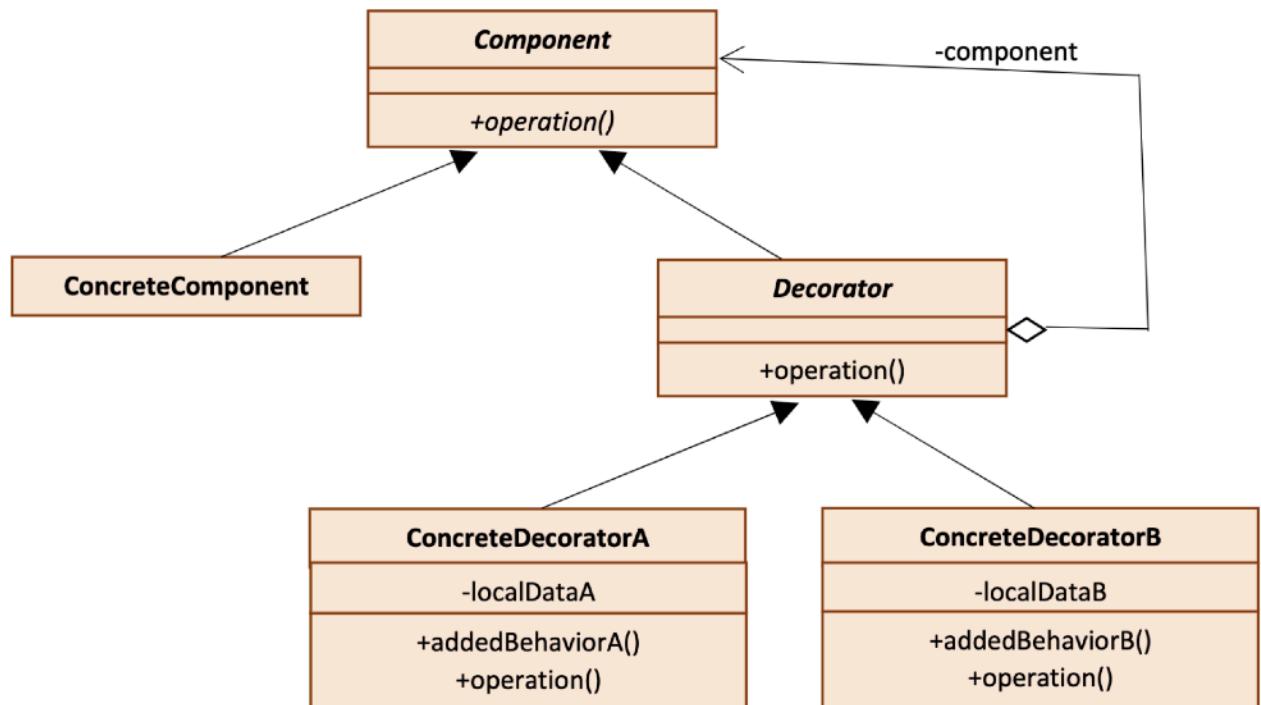
Il design pattern **decorator** permette di aggiungere nuove responsabilità o comportamenti a un oggetto in maniera dinamica. Questo viene realizzato costruendo una nuova classe decoratore che "avvolge" l'oggetto originale.

Scopo del decorator: aggiungere dinamicamente responsabilità ad un oggetto.

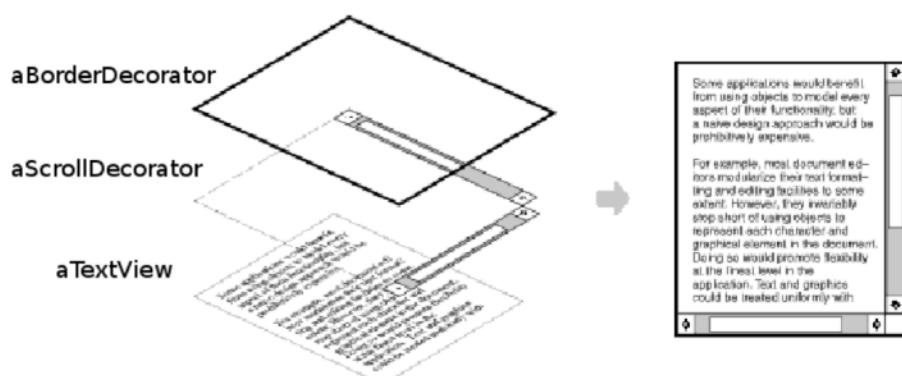
I decoratori forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità.

Per sua natura come sottolinea anche il Prof. De Angelis conviene presentare il pattern **Decorator** esattamente nella forma in cui è rappresentato nel libro originale della *Banda dei Quattro*. Tuttavia, esso può essere esteso a un'infinità di domini differenti e adattato anche a contesti che esulano dalle interfacce grafiche.

Morale: il Decorator non è necessariamente legato al solo ambito delle interfacce grafiche.



Normalmente l'interfaccia grafica può essere rappresentata come una serie di livelli sovrapposti.



Ad ogni livello possiamo assegnare una responsabilità specifica.
Ad esempio, una finestra del nostro ambiente grafico può essere interpretata come composta da tre strati: il **contenuto**, il **livello di scorrimento** e un **livello di decorazione dei bordi**.

L'interfaccia, quindi, può essere rappresentata come una sovrapposizione di strati indipendenti e intercambiabili, che possono essere applicati l'uno sopra l'altro.
Nulla vieta, inoltre, di introdurre ulteriori livelli, come ad esempio una **colorazione di sfondo**.

L'idea di fondo è quella di concepire l'ambiente come un insieme di informazioni **strutturate e componibili dinamicamente**: a runtime è possibile scegliere come comporre lo **stack dei livelli**, ottenendo così un'interfaccia flessibile e personalizzabile.

Il colore di sfondo decido di aggiungerlo a Run-Time.

Decido di comporre dinamicamente i contenuti in modo stratificato senza andare a modificare la struttura del sistema.

Elementi del Decorator:

Component (Componente)

- È l'**interfaccia** o classe astratta che definisce l'operazione base che sia l'oggetto originale sia i decorator devono implementare.

ConcreteComponent (Componente Concreto) - oggetto base da decorare

- È l'**oggetto reale di base** a cui si possono aggiungere dinamicamente responsabilità.
- Implementa Component.

Decorator (Decoratore astratto)

- Classe astratta che implementa **Component** e contiene un riferimento a un **Component**

- Ha lo scopo di delegare le chiamate all'oggetto “decorato”.

ConcreteDecorator (Decoratore Concreto)

- Estende Decorator e aggiunge nuove funzionalità all'oggetto decorato.

Il vantaggio principale è che puoi aggiungere “decorazioni” (funzionalità extra) ad un oggetto:

- **senza modificare il codice originale** dell'oggetto (rispetta il principio *Open/Closed*: aperto all'estensione, chiuso alla modifica).
- **in maniera dinamica**: puoi decidere a **runtime** quali strati applicare e in quale ordine.
- **combinando più decorazioni** come mattoncini Lego, senza dover creare nuove sottoclassi per ogni combinazione.

Esempio di codice.

```
// Component @Author Simone Remoli
interface Component {
    void operation();
}

// ConcreteComponent: implementazione base
class ConcreteComponent implements Component {
    @Override
    public void operation() {
        System.out.println("Operazione di base del ConcreteComponent");
    }
}

// Decorator: classe astratta che implementa Component
abstract class Decorator implements Component {
    protected Component component;

    public Decorator(Component component) {
        this.component = component;
    }

    @Override
    public void operation() {
        // delega all'oggetto decorato
        component.operation();
    }
}
```

```

// ConcreteDecoratorA: aggiunge un comportamento extra
class ConcreteDecoratorA extends Decorator {
    private String localDataA = "Dati extra A";

    public ConcreteDecoratorA(Component component) {
        super(component);
    }

    private void addedBehaviorA() {
        System.out.println("Comportamento aggiuntivo A: uso " +
localDataA);
    }

    @Override
    public void operation() {
        super.operation(); // chiama l'operazione di base
        addedBehaviorA(); // aggiunge nuovo comportamento
    }
}

// ConcreteDecoratorB: aggiunge un altro comportamento extra
class ConcreteDecoratorB extends Decorator {
    private int localDataB = 42;

    public ConcreteDecoratorB(Component component) {
        super(component);
    }

    private void addedBehaviorB() {
        System.out.println("Comportamento aggiuntivo B: valore = " +
localDataB);
    }

    @Override
    public void operation() {
        super.operation(); // chiama l'operazione precedente
        addedBehaviorB(); // aggiunge nuovo comportamento
    }
}

// Client
public class Main {
    public static void main(String[] args) {
        // Creo il componente base
        Component simple = new ConcreteComponent();

        // Aggiungo decoratore A
        Component decoratedA = new ConcreteDecoratorA(simple);

        // Aggiungo decoratore B sopra A
        Component decoratedAB = new ConcreteDecoratorB(decoratedA);

        System.out.println("==== Solo componente base ====");
        simple.operation();
    }
}

```

```

        System.out.println("\n==== Con Decorator A ===");
        decoratedA.operation();

        System.out.println("\n==== Con Decorator A + B ===");
        decoratedAB.operation();
    }
}

```

Output:

```

        === Solo componente base ===
        Operazione di base del ConcreteComponent

        === Con Decorator A ===
        Operazione di base del ConcreteComponent
        Comportamento aggiuntivo A: uso Dati extra A

        === Con Decorator A + B ===
        Operazione di base del ConcreteComponent
        Comportamento aggiuntivo A: uso Dati extra A
        Comportamento aggiuntivo B: valore = 42

```

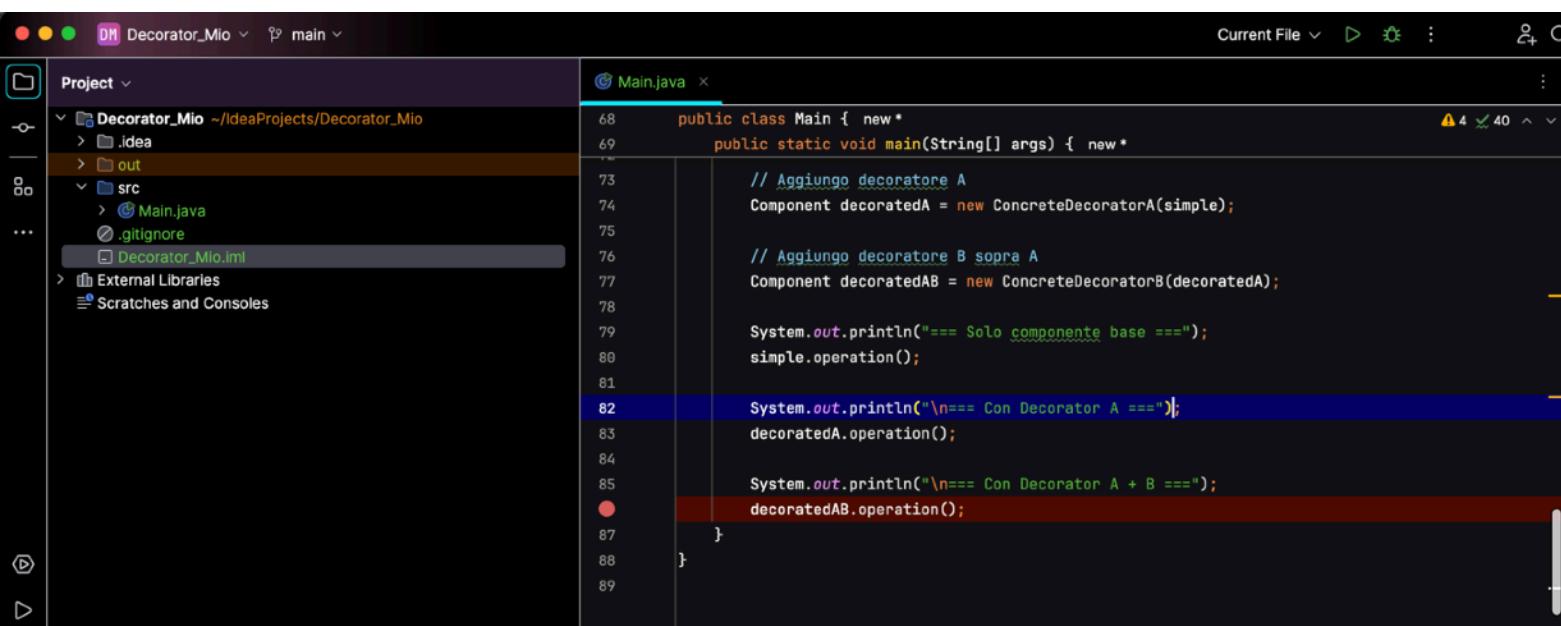
In questo modo si vede chiaramente:

- ConcreteComponent fa il lavoro di base,
- DecoratorA aggiunge un comportamento in più,
- DecoratorB aggiunge un altro,
- e il client può **combinare i decoratori come vuole**, senza dover creare mille sottoclassi diverse.

L'ordine degli strati conta: se inverti A e B, cambia l'ordine con cui compaiono i messaggi/effetti.

Il client vede sempre e solo l'interfaccia Component (sostituibilità), ma sotto c'è una **catena** di oggetti che arricchiscono il comportamento.

DEBUGGING SUL DECORATOR - BREAKPOINT



The screenshot shows an IDE interface with the project 'Decorator_Mio' open. The 'src' folder contains the 'Main.java' file. The code in 'Main.java' is as follows:

```
public class Main { new*
    public static void main(String[] args) { new*
        // Aggiungo decoratore A
        Component decoratedA = new ConcreteDecoratorA(simple);

        // Aggiungo decoratore B sopra A
        Component decoratedAB = new ConcreteDecoratorB(decoratedA);

        System.out.println("==> Solo componente base ==>");
        simple.operation();

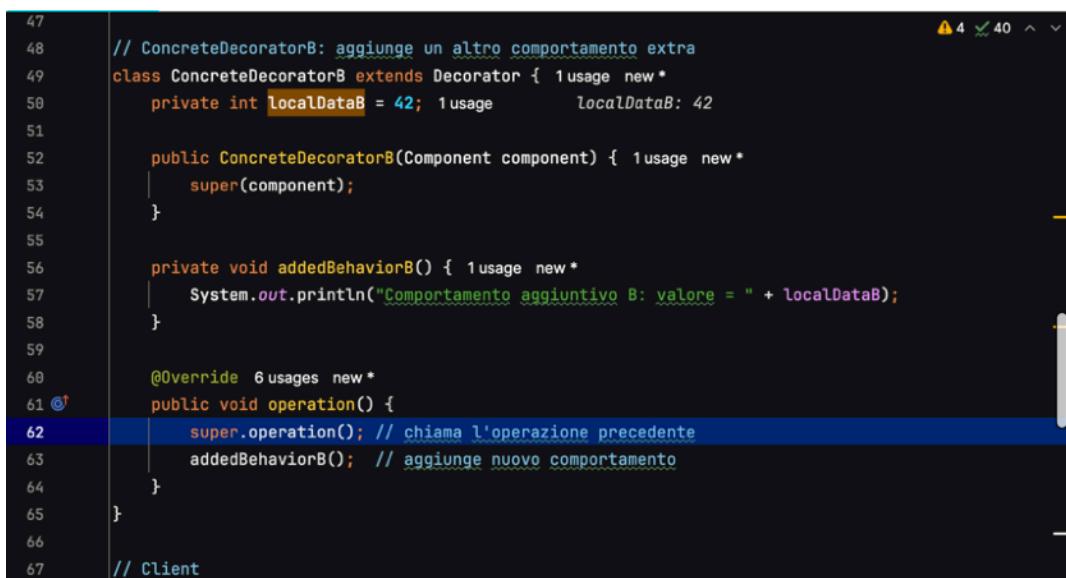
        System.out.println("\n==> Con Decorator A ==>");
        decoratedA.operation();

        System.out.println("\n==> Con Decorator A + B ==>");
        decoratedAB.operation();
    }
}
```

A red dot indicates a breakpoint is set on line 86, where 'decoratedAB.operation()' is called. The status bar at the top right shows 'Current File' and '40'.

Immagina di mettere un breakpoint alla linea 86 e procedere con il debug.

Vediamo cosa succede passo passo.



The screenshot shows the 'ConcreteDecoratorB.java' code during a debug session. The code is as follows:

```
// ConcreteDecoratorB: aggiunge un altro comportamento extra
class ConcreteDecoratorB extends Decorator {
    private int localDataB = 42; 1 usage      localDataB: 42

    public ConcreteDecoratorB(Component component) { 1 usage new*
        super(component);
    }

    private void addedBehaviorB() { 1 usage new*
        System.out.println("Comportamento aggiuntivo B: valore = " + localDataB);
    }

    @Override 6 usages new*
    public void operation() {
        super.operation(); // chiama l'operazione precedente
        addedBehaviorB(); // aggiunge nuovo comportamento
    }
}

// Client
```

The execution path is highlighted with blue bars. The cursor is at line 62, where 'super.operation()' is called. The status bar at the top right shows 'Current File' and '40'.



The screenshot shows the 'Decorator.java' code during a debug session. The code is as follows:

```
// Decorator: classe astratta che implementa Component
abstract class Decorator implements Component { 2 usages 2 inheritors new*
    protected Component component; 2 usages      component: ConcreteDecoratorA@782

    public Decorator(Component component) { 2 usages new*
        this.component = component;
    }

    @Override 6 usages 2 overrides new*
    public void operation() {
        // delega all'oggetto decorato
        component.operation(); component: ConcreteDecoratorA@782
    }
}
```

The execution path is highlighted with blue bars. The cursor is at line 25, where 'component.operation()' is called. The status bar at the top right shows 'Current File' and '40'.

Ovvio no? La classe decorator è la superclasse di concretedecoratorB.

In memoria ho già creato:

- simple → un ConcreteComponent.
- decoratedA → un ConcreteDecoratorA che contiene simple.
- decoratedAB → un ConcreteDecoratorB che contiene decoratedA.

Variabili allo stato attuale:

- simple = ConcreteComponent
- decoratedA = ConcreteDecoratorA(component = simple)
- decoratedAB = ConcreteDecoratorB(component = decoratedA).

ConcreteDecoratorB → ConcreteDecoratorA → ConcreteComponent.

Qui, component = decoratedA, perché B è stato costruito con decoratedA.

Quindi si va nella classe concretedecoratorA:

```
30     class ConcreteDecoratorA extends Decorator { 1 usage new *
31         private String localDataA = "Dati extra A"; 1 usage           localDataA: "Dati extra A"
32
33         public ConcreteDecoratorA(Component component) { 1 usage new *
34             super(component);
35         }
36
37         private void addedBehaviorA() { 1 usage new *
38             System.out.println("Comportamento aggiuntivo A: uso " + localDataA);
39         }
40
41         @Override 6 usages new *
42         public void operation() {
43             super.operation(); // chiama l'operazione di base
44             addedBehaviorA(); // aggiunge nuovo comportamento
45         }
46     }
```

```
22         @Override 6 usages 2 overrides new *
23         public void operation() {
24             // delega all'oggetto decorato
25             |     component.operation(); component: ConcreteComponent@781
26         }
27     }
```

Qui, **component** = **simple**.

Simple era di tipo **ConcreteComponent()**, per il polimorfismo la funzione chiamata è questa.

```
6  // ConcreteComponent. Implementazione base a cui si aggiungono cose
7  class ConcreteComponent implements Component { 1 usage new*
8      @Override 6 usages new*
9      public void operation() {
10          System.out.println("Operazione di base del ConcreteComponent");
11      }
12  }
```

```
30 class ConcreteDecoratorA extends Decorator { 1 usage new*
31     private String localDataA = "Dati extra A"; 1 usage           localDataA: "Dati extra A"
32
33     public ConcreteDecoratorA(Component component) { 1 usage new*
34         super(component);
35     }
36
37     private void addedBehaviorA() { 1 usage new*
38         System.out.println("Comportamento aggiuntivo A: uso " + localDataA);
39     }
40
41     @Override 6 usages new*
42     public void operation() {
43         super.operation(); // chiama l'operazione di base
44         addedBehaviorA(); // aggiunge nuovo comportamento
45     }
46 }
```

```
30 class ConcreteDecoratorA extends Decorator { 1 usage new*
31     private String localDataA = "Dati extra A"; 1 usage           localDataA: "Dati extra A"
32
33     public ConcreteDecoratorA(Component component) { 1 usage new*
34         super(component);
35     }
36
37     private void addedBehaviorA() { 1 usage new*
38         System.out.println("Comportamento aggiuntivo A: uso " + localDataA); localDataA: "Dati e
39     }
40
41     @Override 6 usages new*
42     public void operation() {
43         super.operation(); // chiama l'operazione di base
44         addedBehaviorA(); // aggiunge nuovo comportamento
45     }
46 }
```

```

48 // ConcreteDecoratorB: aggiunge un altro comportamento extra
49 class ConcreteDecoratorB extends Decorator { 1 usage new*
50     private int localDataB = 42; 1 usage           localDataB: 42
51
52     public ConcreteDecoratorB(Component component) { 1 usage new*
53         super(component);
54     }
55
56     private void addedBehaviorB() { 1 usage new*
57         System.out.println("Comportamento aggiuntivo B: valore = " + localDataB);
58     }
59
60     @Override 6 usages new*
61     public void operation() {
62         super.operation(); // chiama l'operazione precedente
63         addedBehaviorB(); // aggiunge nuovo comportamento
64     }

```

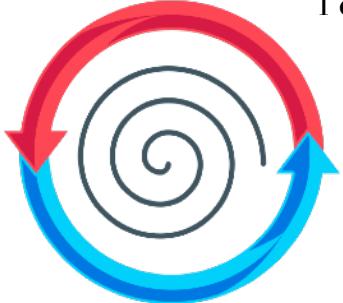
```

48 // ConcreteDecoratorB: aggiunge un altro comportamento extra
49 class ConcreteDecoratorB extends Decorator { 1 usage new*
50     private int localDataB = 42; 1 usage           localDataB: 42
51
52     public ConcreteDecoratorB(Component component) { 1 usage new*
53         super(component);
54     }
55
56     private void addedBehaviorB() { 1 usage new*
57         System.out.println("Comportamento aggiuntivo B: valore = " + localDataB); localDataB: 42
58     }
59
60     @Override 6 usages new*
61     public void operation() {
62         super.operation(); // chiama l'operazione precedente
63         addedBehaviorB(); // aggiunge nuovo comportamento
64     }

```

Fine esecuzione.

Mental model: **matrioska**. Ogni decorator apre la bambola e passa la chiamata a quella più interna; al ritorno aggiunge la propria “decorazione”.



I concetto di base è strettamente connesso all'utilizzo della logica.
Vorrei trasmettere un consiglio: immaginate che ogni volta che viene creato un nuovo decorator composto, esso sia a sua volta costituito da una catena di decorator concreti.

Quando istanzio una loro classe, utilizzo **super()** come forse avete notato e passo quindi alla classe superiore, che

assegna al componente il parametro. In questo modo, nella concreta decorazione di **B**, il parametro corrisponde alla concreta decorazione di **A**, che viene impostata tramite la **super** di **B**.

Il consiglio che voglio dare è il seguente: esiste una catena, e dovete ragionare in termini di **ricorsione**.

La catena prevede la creazione **S → A → B**; al contrario, la risalita avviene come **B super A super S**. Da questo punto, il flusso delle chiamate procede “verso il basso” nella sequenza **S → A → B**.

L'estensione diretta attraverso le definizioni di sottoclassi non è praticabile.

E questa era solo la premessa.

Adesso addentriamoci nel codice del Prof.De Angelis.

Struttura del progetto: due package.



Come potete notare non esiste un’interfaccia. Il motivo è che l’interfaccia è facilmente intercambiabile con una classe astratta.

Di seguito il codice e il relativo output:

```
package decorations;

import component.*;
public abstract class Decorator extends VisualComponent {

    private VisualComponent component;

    public Decorator( VisualComponent component){
        this.component = component;
    }

    @Override
    public String draw() {
        String resultsFromRedirection = this.component.draw();
        return resultsFromRedirection;
    }
}

package decorations;

import component.*;
public class ScrollDecorator extends Decorator {

    private int scrollPosition;

    public ScrollDecorator(VisualComponent component) {
        super(component);
        scrollPosition = 0;
    }

    public void scrollTo(int offset){
        this.scrollPosition = offset;
    }

    protected String applyScroll(String input){
        String output = "[scroll " + this.scrollPosition + "]" + input +
"[/scroll]";
        return output;
    }

    @Override
    public String draw() {
        String preliminaryResults = super.draw();
        preliminaryResults = this.applyScroll(preliminaryResults);
        return preliminaryResults;
    }
}

package decorations;
```

```
import component.*;
public class BorderDecorator extends Decorator {

    private int borederWidth;
    private int tick;

    public BorderDecorator(VisualComponent component) {
        super(component);
        this.setWidth(5);
        this.tick = 1;
    }

    public void setWidth(int width){
        if (width > 0 )
            this.borederWidth = width;
    }

    public void incWidth(){
        this.borederWidth += this.tick;
    }

    public void decWidth(){
        if ((this.borederWidth - this.tick) > 0)
            this.borederWidth -= this.tick;
        else
            this.borederWidth = 0;
    }

    protected String applyBorder(String input){
        String output = "[border " + this.borederWidth + "]" + input +
"[ /border ]";
        return output;
    }

    @Override
    public String draw() {
        String preliminaryResults = super.draw();
        preliminaryResults = this.applyBorder(preliminaryResults);
        return preliminaryResults;
    }
}

package component;

public abstract class VisualComponent {

    public abstract String draw();

}

package component;

public class TextView extends VisualComponent {

    private String text;
```

```
public TextView(String text){
    this.setText(text);
}

public void setText(String text){
    this.text = text;
}

@Override
public String draw() {
    return (this.text);
}

}



---


package component;
import decorations.*;

public class ContentContainer {

    private VisualComponent contents;

    public ContentContainer (){
        this.setContents(null);
    }

    public ContentContainer (VisualComponent contents){
        this.setContents(contents);
    }

    public void setContents(VisualComponent contents) {
        this.contents = contents;
    }

    public void display(){
        System.out.println(this.contents.draw());
    }

    public String getContents(){
        return this.contents.draw();
    }

    public static void main(String args[]){
        TextView tv = new TextView("This is foo!!!");

        ContentContainer me = new ContentContainer(tv);
        me.display();
        System.out.println();

        BorderDecorator btv = new BorderDecorator(tv);
        me.setContents(btv);
        me.display();
        btv.incWidth();
        tv.setText("This WAS foo ... ");
        me.display();
        System.out.println();
    }
}
```

```

    ScrollDecorator sbtv = new ScrollDecorator(btv);
    me.setContents(sbtv);
    me.display();
    sbtv.scrollTo(-4);
    btv.incWidth();
    tv.setText("This WAS foo ... and it keeps changing!!!");
    me.display();
}
}

```

Output:

This is foo!!!

[border 5]This is foo!!![/border]
[border 6]This WAS foo ... [/border]

[scroll 0][border 6]This WAS foo ... [/border][/scroll]
[scroll -4][border 7]This WAS foo ... and it keeps changing!!![/border][/scroll]

Innanzitutto capiamo subito i ruoli.

RUOLI

Ruolo	Classe	Descrizione
Component	VisualComponent	Interfaccia astratta comune
Concrete Component	TextView	Componente reale, disegnabile
Decorator astratto	Decorator	Classe base per i decoratori; mantiene riferimento al componente
Concrete Decorators	BorderDecorator, ScrollDecorator	Decoratori specifici che estendono il comportamento
Client	ContentContainer	Codice che usa componenti e decoratori in modo trasparente

La tabella mostra i ruoli del Pattern Decorator inerente al codice di Gulyx.

Siamo pronti? Procediamo con il debug passo passo.

Immaginiamo, al solito, di inserire un breakpoint alla linea 29.

The screenshot shows the IntelliJ IDEA interface. On the left is the Project tool window with the 'src' folder expanded, showing 'component' and 'decorations' packages. The ContentContainer.java file is open in the main editor. A red circle indicates a breakpoint is set on line 29. The code is as follows:

```
public class ContentContainer { new*
    public static void main(String args[]){ new*
        TextView tv = new TextView("This is foo!!!");

        ContentContainer me = new ContentContainer(tv);
        me.display();
        System.out.println();

        BorderDecorator btv = new BorderDecorator(tv);
        me.setContents(btv);
        me.display();
        btv.incWidth();
        tv.setText("This WAS foo ... ");
        me.display();
        System.out.println();

        ScrollDecorator sbtv = new ScrollDecorator(btv);
        me.setContents(sbtv);
        me.display();
        sbtv.scrollTo(offset -4);
        btv.incWidth();
        tv.setText("This WAS foo ... and it keeps changing!!!");
        me.display();
    }
}
```

The screenshot shows the IntelliJ IDEA interface with the TextView.java file open. The code is as follows:

```
package component;
public class TextView extends VisualComponent { //oggetto reale da disegnare 2 usages new*
    private String text; 2 usages      text: "This is foo!!!"      text: null
    public TextView(String text){ 1 usage new*
        this.setText(text);
    }
    public void setText(String text){ 3 usages new*
        this.text = text;
    }
    @Override 5 usages new*
    public String draw(){ 
        return (this.text);
    }
}
```

Below the editor, the Debug tool window is open, showing a stack trace for the ContentContainer class. It shows the current frame is at line 29, with the variable 'text' set to "This is foo!!!".

```
main:29, ContentCon
Switch frames from any... X
```

Quindi viene istanziato un oggetto di tipo TextView. Guarda dopo.

Project

```

1 package component;
2
3 public class TextView extends VisualComponent { //oggetto reale da disegnare 2 usages new*
4
5     private String text; 2 usages      text: null
6
7     public TextView(String text){ 1 usage new*
8         this.setText(text);
9     }
10
11    public void setText(String text){ 3 usages new*      text: "This is foo!!!"
12        this.text = text;  text: "This is foo!!!"  text: null
13    }
14
15    @Override 5 usages new*
16    public String draw(){ 1 usage new*
17        return (this.text);
18    }
19
20}
21

```

Debug ContentContainer

Evaluate expression (e) or add a watch (w)

```

"mai...NING" 
setText:12, TextView
<init>:8, TextView
main:29, ContentC

```

Switch frames from any...

A questo punto si torna nel client.

Project

```

public class ContentContainer { new*
    public String getContents(){ no usages new*
        return this.contents.draw();
    }
    public static void main(String args[]){ new*      args: []
        TextView tv = new TextView("This is foo!!!");  tv: TextView@777
        ContentContainer me = new ContentContainer(tv);  tv: ContentContainer@777
        me.display();
        System.out.println();
        BorderDecorator btv = new BorderDecorator(tv);
        me.setContents(btv);
        me.display();
        btv.incWidth();
        tv.setText("This WAS foo ... ");
        me.display();
        System.out.println();
        ScrollDecorator sbtv = new ScrollDecorator(btv);
    }
}
```

è una semplice istanziazione dell'oggetto ContentContainer, e serve per avvolgere (contenere) un componente visivo (tv, cioè il TextView).

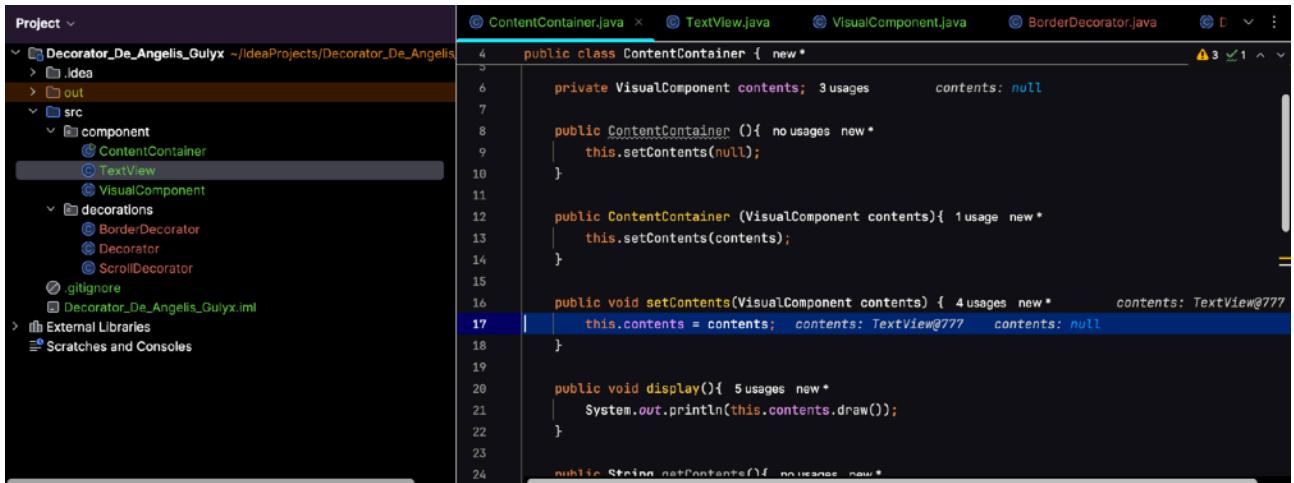
Project

```

public class ContentContainer { new*
    private VisualComponent contents; 3 usages      contents: null
    public ContentContainer (){ no usages new*
        this.setContents(null);
    }
    public ContentContainer (VisualComponent contents){ 1 usage new*      contents: TextView@777
        this.setContents(contents);  contents: TextView@777
    }
    public void setContents(VisualComponent contents){ 4 usages new*
        this.contents = contents;
    }
    public void display(){ 5 usages new*
        System.out.println(this.contents.draw());
    }
    public String getContents(){ no usages new*

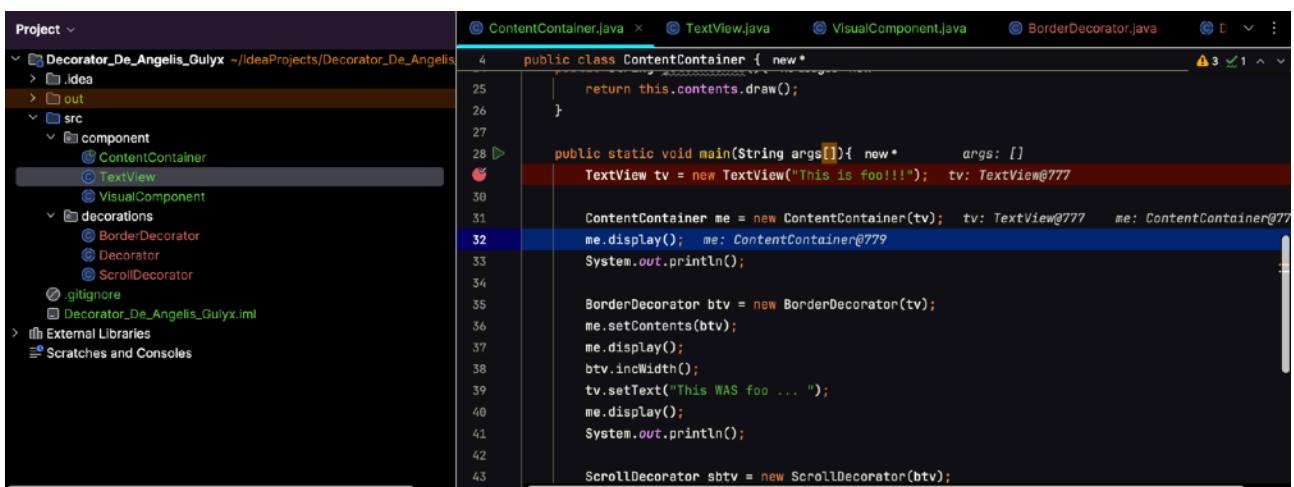
```

Nella successiva istruzione, semplicemente si assegna il riferimento al campo privato **contents** della classe **ContentContainer**.



```
public class ContentContainer { new*
    private VisualComponent contents; 3 usages      contents: null
    public ContentContainer (){ no usages new*
        this.setContents(null);
    }
    public ContentContainer (VisualComponent contents){ 1 usage new*
        this.setContents(contents);
    }
    public void setContents(VisualComponent contents) { 4 usages new*      contents: TextView@777
        this.contents = contents;  contents: TextView@777  contents: null
    }
    public void display(){ 5 usages new*
        System.out.println(this.contents.draw());
    }
    public String getContents(){ no usages new*
}
```

I container “contiene” (cioè memorizza internamente) l’oggetto **tv**.

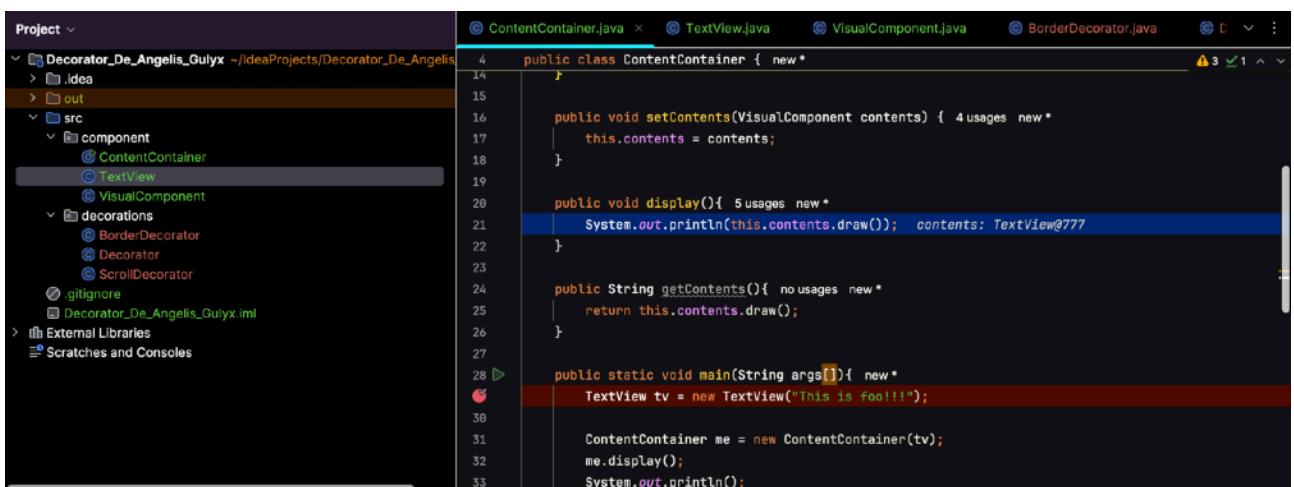


```
public class ContentContainer { new*
    return this.contents.draw();
}
public static void main(String args[]){ new*      args: []
    TextView tv = new TextView("This is foo!!!");  tv: TextView@777
    ContentContainer me = new ContentContainer(tv);  tv: TextView@777  me: ContentContainer@777
    me.display();  me: ContentContainer@777
    System.out.println();

    BorderDecorator btv = new BorderDecorator(tv);
    me.setContents(btv);
    me.display();
    btv.incWidth();
    tv.setText("This WAS foo ... ");
    me.display();
    System.out.println();

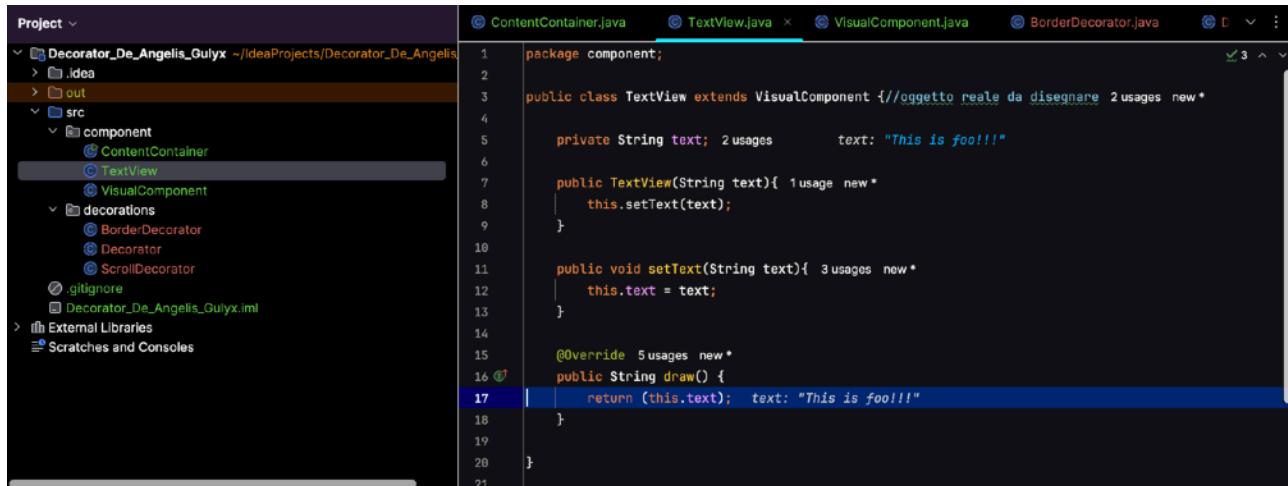
    ScrollDecorator sbtv = new ScrollDecorator(btv);
}
```

Ora attenzione, chiamiamo **display**.



```
public class ContentContainer { new*
    r
    public void setContents(VisualComponent contents) { 4 usages new*
        this.contents = contents;
    }
    public void display(){ 5 usages new*
        System.out.println(this.contents.draw());  contents: TextView@777
    }
    public String getContents(){ no usages new*
        return this.contents.draw();
    }
    public static void main(String args[]){ new*
        TextView tv = new TextView("This is foo!!!");
        ContentContainer me = new ContentContainer(tv);
        me.display();
        System.out.println();
    }
}
```

Quindi, subito dopo la costruzione, il campo interno `this.contents` del nostro oggetto `me` vale esattamente `tv`, cioè il `TextView` che il Prof ha passato.



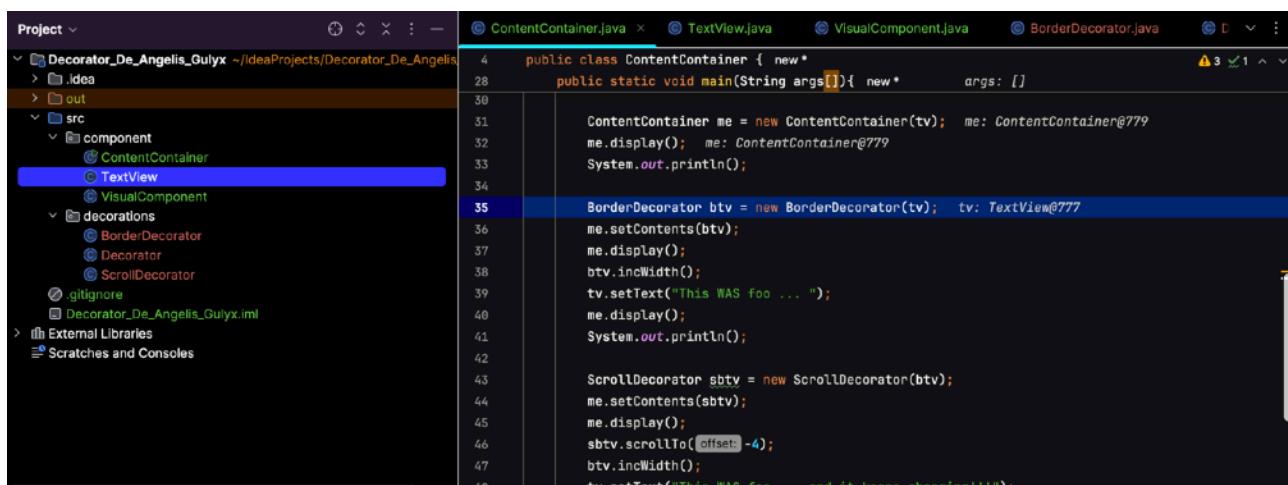
```

1 package component;
2
3 public class TextView extends VisualComponent { //oggetto reale da disegnare 2 usages new*
4
5     private String text; 2 usages      text: "This is foo!!!"
6
7     public TextView(String text){ 1 usage new*
8         this.setText(text);
9     }
10
11    public void setText(String text){ 3 usages new*
12        this.text = text;
13    }
14
15    @Override 5 usages new*
16    public String draw() {
17        return (this.text); text: "This is foo!!!"
18    }
19
20
21

```

E così abbiamo effettuato la prima stampa: This is foo!!!

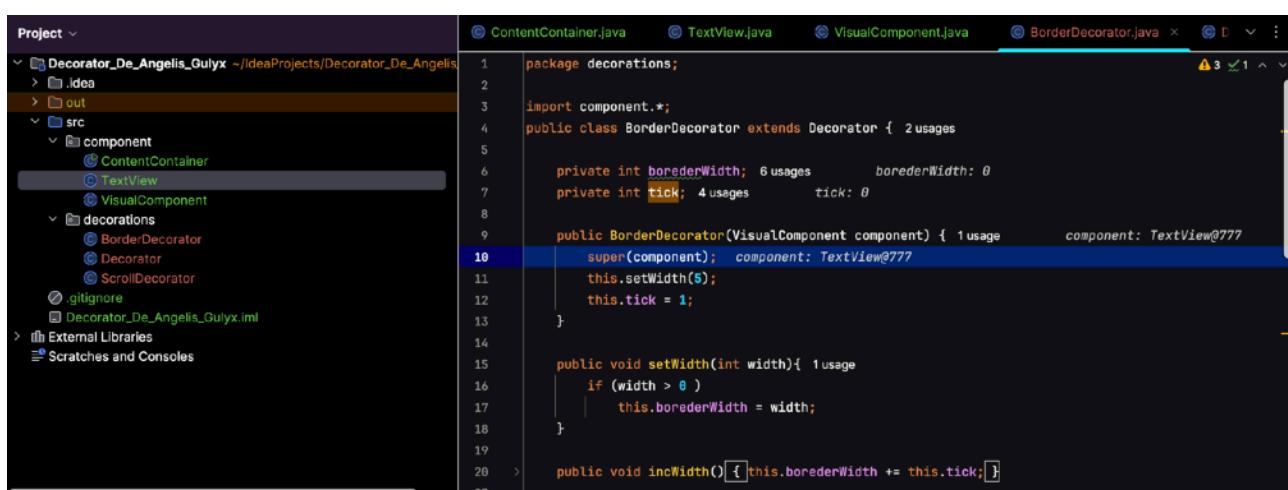
Ora si crea un decoratore che avvolge il `TextView`.



```

4     public class ContentContainer { new*
28     public static void main(String args[]){ new*      args: []
29
30         ContentContainer me = new ContentContainer(tv); me: ContentContainer@777
31         me.display(); me: ContentContainer@777
32         System.out.println();
33
34         BorderDecorator btv = new BorderDecorator(tv); tv: TextView@777
35         me.setContents(btv);
36         me.display();
37         btv.incWidth();
38         tv.setText("This WAS foo ... ");
39         me.display();
40         System.out.println();
41
42
43         ScrollDecorator sbtv = new ScrollDecorator(btv);
44         me.setContents(sbtv);
45         me.display();
46         sbtv.scrollTo(offset -4);
47         btv.incWidth();
48         tv.setText("This WAS foo ... and it keeps changing!!!");

```



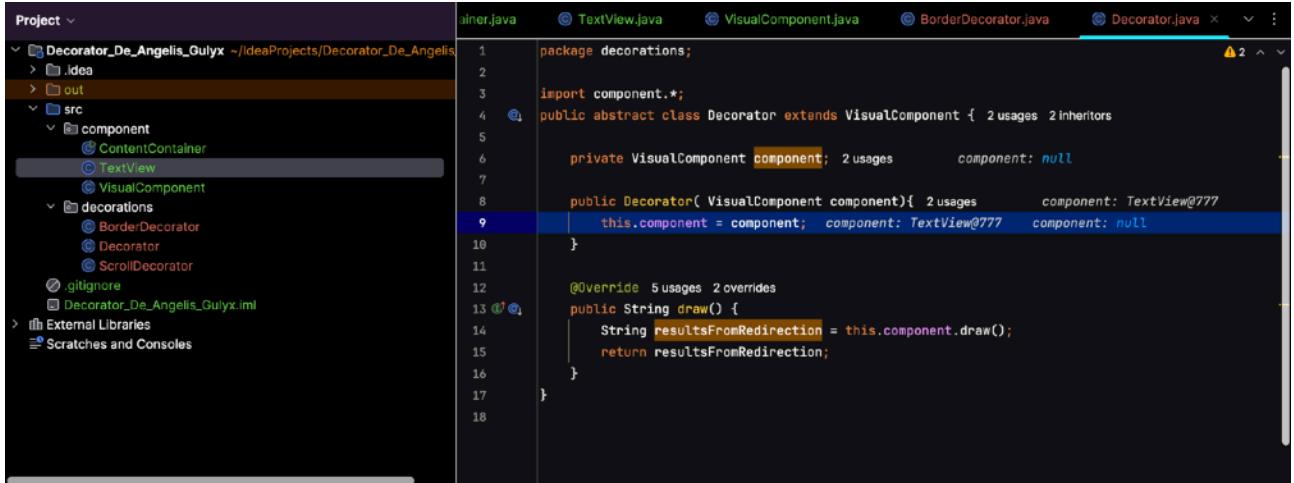
```

1 package decorations;
2
3 import component.*;
4 public class BorderDecorator extends Decorator { 2 usages
5
6     private int borederWidth; 6 usages      borederWidth: 0
7     private int tick; 4 usages      tick: 0
8
9     public BorderDecorator(VisualComponent component) { 1 usage      component: TextView@777
10        super(component); component: TextView@777
11        this.setWidth(5);
12        this.tick = 1;
13    }
14
15    public void setWidth(int width){ 1 usage
16        if (width > 0 )
17            this.borederWidth = width;
18    }
19
20    public void incWidth(){ this.borederWidth += this.tick; }
21

```

Il costruttore di BorderDecorator riceve come argomento component = tv.

Viene chiamato il costruttore della superclasse (Decorator) con
super(component).

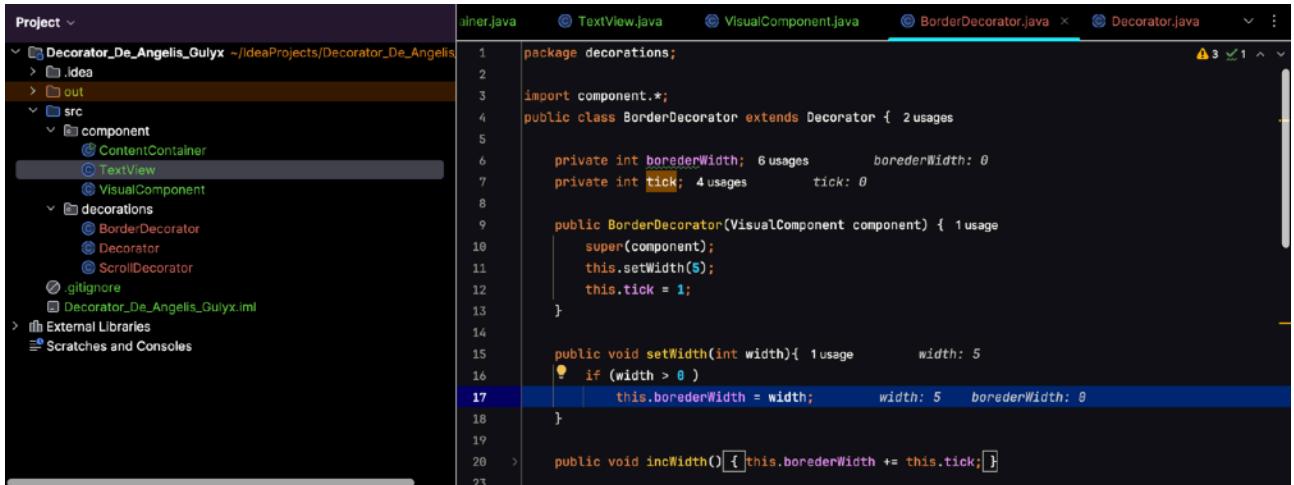


```
Project: Decorator_De_Angelis_Gulyx
File: BorderDecorator.java
1 package decorations;
2
3 import component.*;
4
5 public abstract class Decorator extends VisualComponent {
6     private VisualComponent component;
7
8     public Decorator(VisualComponent component) {
9         this.component = component;
10    }
11
12     @Override
13     public String draw() {
14         String resultsFromRedirection = this.component.draw();
15         return resultsFromRedirection;
16     }
17 }
18
```

Qui il parametro component (cioè tv) viene salvato internamente nel campo privato this.component del decoratore.

Tornati nel costruttore di BorderDecorator, vengono eseguite:

```
this.setWidth(5);
this.tick = 1;
```

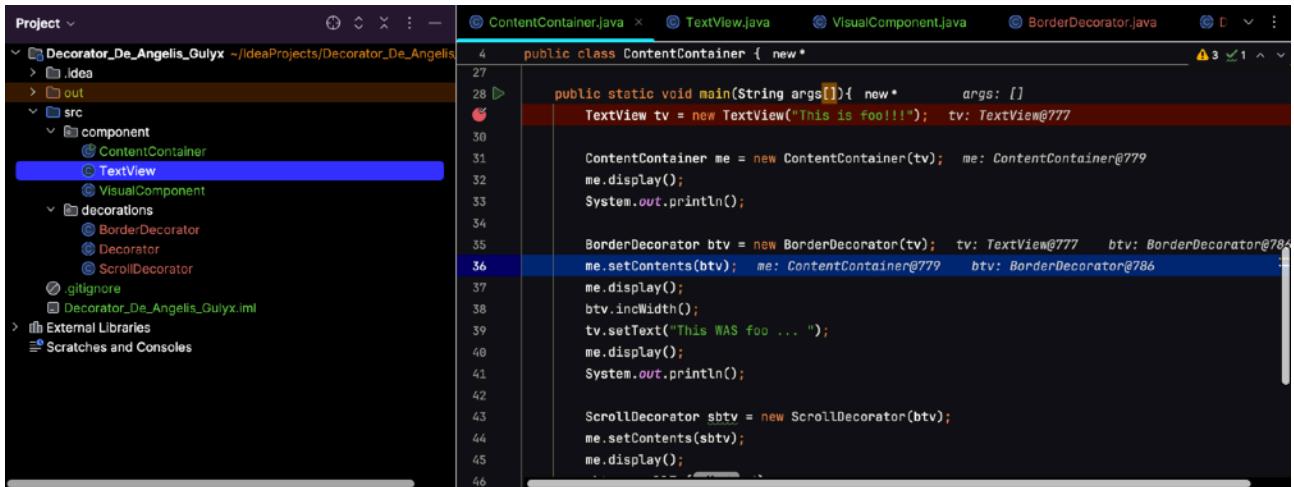


```
Project: Decorator_De_Angelis_Gulyx
File: BorderDecorator.java
1 package decorations;
2
3 import component.*;
4
5 public class BorderDecorator extends Decorator {
6     private int borederWidth;
7     private int tick;
8
9     public BorderDecorator(VisualComponent component) {
10        super(component);
11        this.setWidth(5);
12        this.tick = 1;
13    }
14
15     public void setWidth(int width) {
16         if (width > 0)
17             this.borederWidth = width;
18     }
19
20     public void incWidth() {
21         this.borederWidth += this.tick;
22     }
23 }
```

Quindi me è un oggetto della classe ContentContainer. Btv è un oggetto di tipo BorderDecorator, costruito così:

BorderDecorator btv = new BorderDecorator(tv);

Questo significa che btv avvolge già il TextView tv.



```

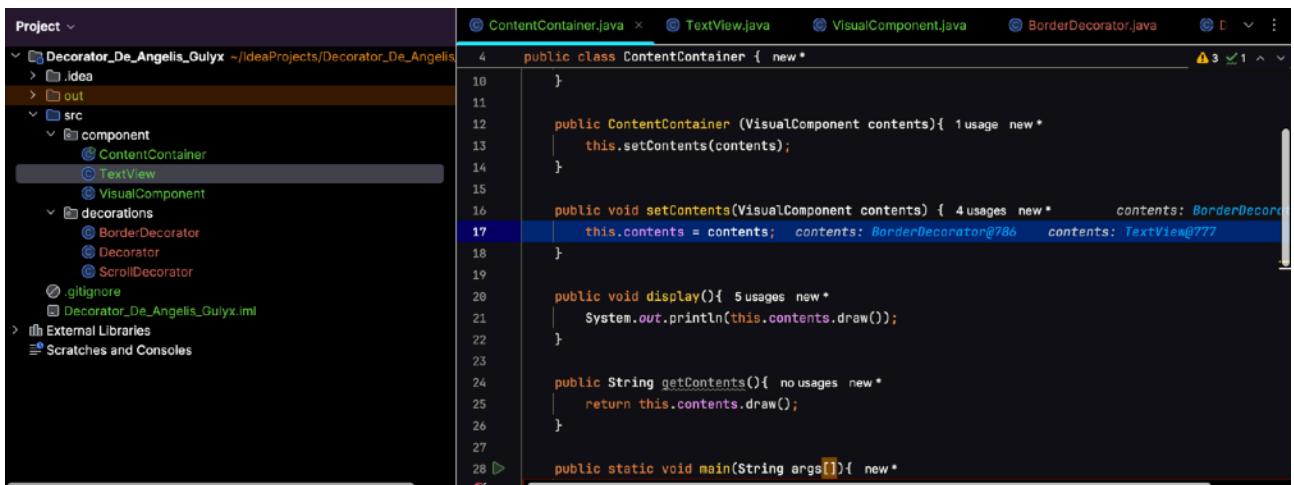
public class ContentContainer { new*
    public static void main(String args[]){ new*      args: []
        TextView tv = new TextView("This is foo!!!"); tv: TextView@777
        ContentContainer me = new ContentContainer(tv); me: ContentContainer@779
        me.display();
        System.out.println();

        BorderDecorator btv = new BorderDecorator(tv); tv: TextView@777 btv: BorderDecorator@786
        me.setContents(btv); me: ContentContainer@779 btv: BorderDecorator@786
        me.display();
        btv.incWidth();
        tv.setText("This WAS foo ... ");
        me.display();
        System.out.println();

        ScrollDecorator sbtv = new ScrollDecorator(btv);
        me.setContents(sbtv);
        me.display();
    }
}

```

Ora, il metodo che viene chiamato sostituisce il contenuto visuale interno (`this.contents`) con quello che gli passo come parametro.



```

public class ContentContainer { new*
    public ContentContainer (VisualComponent contents){ 1 usage new*
        this.setContents(contents);
    }

    public void setContents(VisualComponent contents) { 4 usages new*      contents: BorderDecorat
    this.contents = contents; contents: BorderDecorator@786 contents: TextView@777
    }

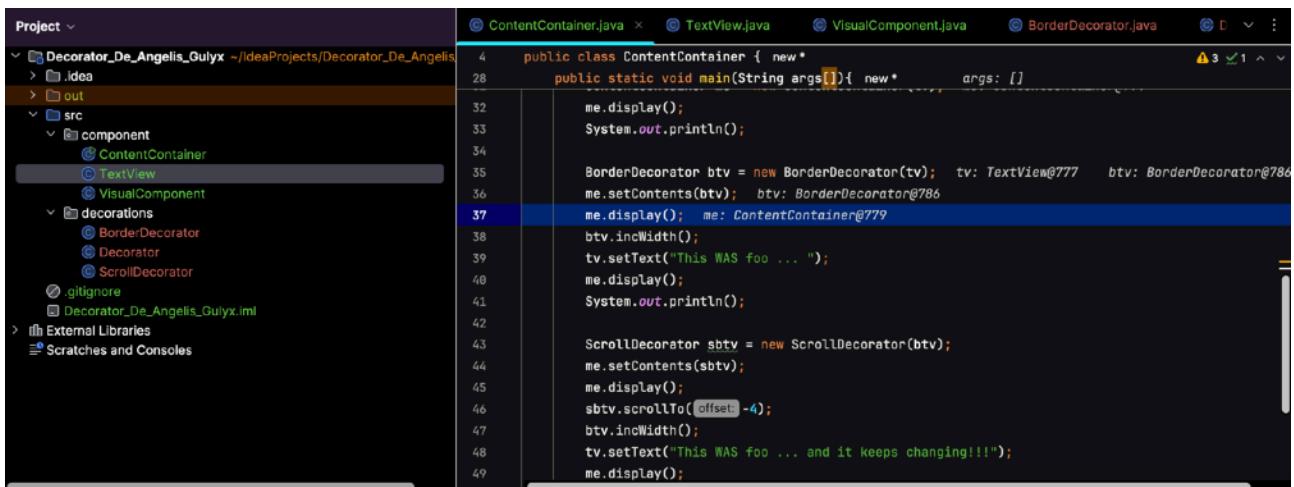
    public void display(){ 5 usages new*
        System.out.println(this.contents.draw());
    }

    public String getContents(){ no usages new*
        return this.contents.draw();
    }

    public static void main(String args[]){ new*
}

```

`this.contents` vale il riferimento all'oggetto `btv`, cioè al decoratore che avvolge il `TextView`.



```

public class ContentContainer { new*
    public static void main(String args[]){ new*      args: []
        me.display();
        System.out.println();

        BorderDecorator btv = new BorderDecorator(tv); tv: TextView@777 btv: BorderDecorator@786
        me.setContents(btv); btv: BorderDecorator@786
        me.display(); me: ContentContainer@779
        btv.incWidth();
        tv.setText("This WAS foo ... ");
        me.display();
        System.out.println();

        ScrollDecorator sbtv = new ScrollDecorator(btv);
        me.setContents(sbtv);
        me.display();
        sbtv.scrollTo( offset: -4);
        btv.incWidth();
        tv.setText("This WAS foo ... and it keeps changing!!!");
        me.display();
    }
}

```

Adesso chiamo `display`.

```

public class ContentContainer {
    public ContentContainer() {
        this.contents = new TextView();
    }

    public void setContents(VisualComponent contents) {
        this.contents = contents;
    }

    public void display() {
        System.out.println(this.contents.draw());
    }

    public String getContents() {
        return this.contents.draw();
    }

    public static void main(String args[]) {
        TextView tv = new TextView("This is foo!!!");

        ContentContainer me = new ContentContainer(tv);
        me.display();
        System.out.println();
    }
}

```

L'oggetto me (cioè il ContentContainer) **non contiene più direttamente** il TextView, ma un decoratore (btv), che a sua volta contiene il TextView originale.

this.contents → è btv.

Quindi la riga equivale a:

System.out.println(btv.draw());

Btv era di tipo BorderDecorator.

Occhio ora.

```

package decorations;

import component.*;
public abstract class Decorator extends VisualComponent {
    private VisualComponent component;

    public Decorator(VisualComponent component) {
        this.component = component;
    }

    @Override
    public String draw() {
        String resultsFromRedirection = this.component.draw();
        return resultsFromRedirection;
    }
}

```

`super.draw()` chiama il metodo `draw()` della superclasse, cioè di **Decorator**:

Qui `this.component` è il `tv` (il `TextView` passato nel costruttore del decoratore).

Quindi, `resultsFromRedirection = tv.draw();`

```
ContentContainer.java TextView.java VisualComponent.java BorderDecorator.java
4   public class BorderDecorator extends Decorator { 2 usages
24  public void decWidth(){ no usages
29  }
30
31  protected String applyBorder(String input){ 1 usage
32  |  String output = "[border " + this.borederWidth + "]" + input + "[/border]";
33  |  return output;
34  }
35
36  @Override 5 usages
37  public String draw() {
38  |  String preliminaryResults = super.draw();
39  |  preliminaryResults = this.applyBorder(preliminaryResults);
40  |  return preliminaryResults;
41
42
43
44
45 }
```

```
TextView.java VisualComponent.java BorderDecorator.java Decorator.java ScrollDecor...
3  public class TextView extends VisualComponent { //oggetto reale da disegnare 2 usages new*
6
7  public TextView(String text){ 1 usage new*
8  |  this.setText(text);
9  }
10
11 public void setText(String text){ 3 usages new*
12  |  this.text = text;
13  }
14
15 @Override 5 usages new*
16  public String draw() {
17  |  return (this.text); text: "This is foo!!!"
18  }
19
20
21 }
```

```
TextView.java VisualComponent.java BorderDecorator.java Decorator.java ScrollDecor...
4  public class BorderDecorator extends Decorator { 2 usages
24  public void decWidth(){ no usages
25  |  if ((this.borederWidth - this.tick) > 0)
26  |  |  this.borederWidth -= this.tick;
27  |  else
28  |  |  this.borederWidth = 0;
29  }
30
31  protected String applyBorder(String input){ 1 usage      input: "This is foo!!!"
32  |  String output = "[border " + this.borederWidth + "]" + input + "[/border]"; input: "This is foo!!!"
33  |  return output; output: "[border 5]This is foo!!![/border]"
34  }
35
36  @Override 5 usages
37  public String draw() {
38  |  String preliminaryResults = super.draw();
39  |  preliminaryResults = this.applyBorder(preliminaryResults);
40  |  return preliminaryResults;
41  }
42
43 }
```

Project ▾

Project ▾

 `-> Decorator_De_Angelis_Gulyx ~/IdeaProjects/Decorator_De_Angelis

 `-> .idea

 `-> out

 `-> src

 `-> component

 `-> ContentContainer

 `-> TextView

 `-> VisualComponent

 `-> decorations

 `-> BorderDecorator

 `-> Decorator

 `-> ScrollDecorator

 `-> .gitignore

 `-> Decorator_De_Angelis_Gulyx.iml

 `-> External Libraries

 `-> Scratches and Consoles

Project ▾

Project ▾

 `-> Decorator_De_Angelis_Gulyx ~/IdeaProjects/Decorator_De_Angelis

 `-> .idea

 `-> out

 `-> src

 `-> component

 `-> ContentContainer

 `-> TextView

 `-> VisualComponent

 `-> decorations

 `-> BorderDecorator

 `-> Decorator

 `-> ScrollDecorator

 `-> .gitignore

 `-> Decorator_De_Angelis_Gulyx.iml

 `-> External Libraries

 `-> Scratches and Consoles

Project ▾

Project ▾

 `-> Decorator_De_Angelis_Gulyx ~/IdeaProjects/Decorator_De_Angelis

 `-> .idea

 `-> out

 `-> src

 `-> component

 `-> ContentContainer

 `-> TextView

 `-> VisualComponent

 `-> decorations

 `-> BorderDecorator

 `-> Decorator

 `-> ScrollDecorator

 `-> .gitignore

 `-> Decorator_De_Angelis_Gulyx.iml

 `-> External Libraries

 `-> Scratches and Consoles

```

Border.java  @ TextView.java  @ VisualComponent.java  @ BorderDecorator.java  @ Decorator.java  x  v  :
@ TextView.java  @ VisualComponent.java  @ BorderDecorator.java  x  @ Decorator.java  @ ScrollDecorat  v  :
4   public class BorderDecorator extends Decorator { 2 usages
4   public void decWidth(){ no usages
25   if ((this.borderWidth - this.tick) > 0)
26   this.borderWidth -= this.tick;
27   else
28   this.borderWidth = 0;
29 }

31   protected String applyBorder(String input){ 1 usage
32   String output = "[border " + this.borderWidth + "] " + input + "[/border]";
33   return output;
34 }

36 @Override 5 usages
37   public String draw() {
38   String preliminaryResults = super.draw(); preliminaryResults: "[border 5]This is foo!!![/border]"
39   preliminaryResults = this.applyBorder(preliminaryResults);
40   return preliminaryResults; preliminaryResults: "[border 5]This is foo!!![/border]"
41 }

43 public void decWidth(){ no usages
29 }

31   protected String applyBorder(String input){ 1 usage
32   String output = "[border " + this.borderWidth + "] " + input + "[/border]";
33   return output;
34 }

36 @Override 5 usages
37   public String draw() {
38   String preliminaryResults = super.draw(); preliminaryResults: "This is foo!!!"
40 ContentContainer.java  x  @ TextView.java  @ VisualComponent.java  @ BorderDecorator.java  @ D  v  :
4   public class ContentContainer { new*
14   }

15   public void setContents(VisualComponent contents) { 4 usages new*
16   this.contents = contents;
17   }

19   public void display(){ 5 usages new*
21   System.out.println(this.contents.draw()); contents: BorderDecorator@786
22   }

24   public String getContents(){ no usages new*
25   return this.contents.draw();
26   }

27   public static void main(String args[]){ new*
28 ContentContainer.java  x  @ TextView.java  @ VisualComponent.java  @ BorderDecorator.java  @ D  v  :
4   public class ContentContainer { new*
28   public static void main(String args[]){ new* args: []
30   }

31   ContentContainer me = new ContentContainer(tv); me: ContentContainer@779
32   me.display();
33   System.out.println();

35   BorderDecorator btv = new BorderDecorator(tv); tv: TextView@777 btv: BorderDecorator@786
36   me.setContents(btv);
37   me.display(); me: ContentContainer@779
38   btv.incWidth(); btv: BorderDecorator@786
39   tv.setText("This WAS foo ... ");
40   me.display();
41   System.out.println();

43   ScrollDecorator sbtv = new ScrollDecorator(btv);
44   me.setContents(sbtv);
45   me.display();
46   sbtv.scrollTo(offset: -4);
47   btv.incWidth();
48

```

Project ▾

Project ▾

 `-> Decorator_De_Angelis_Gulyx ~/IdeaProjects/Decorator_De_Angelis

 `-> .idea

 `-> out

 `-> src

 `-> component

 `-> ContentContainer

 `-> TextView

 `-> VisualComponent

 `-> decorations

 `-> BorderDecorator

 `-> Decorator

 `-> ScrollDecorator

 `-> .gitignore

 `-> Decorator_De_Angelis_Gulyx.iml

 `-> External Libraries

 `-> Scratches and Consoles

Project ▾

ContentContainer.java @ TextView.java @ VisualComponent.java @ BorderDecorator.java x @ D v :

```

4   public class BorderDecorator extends Decorator { 2 usages
4   public void setWidth(int width){ 1 usage
16   if (width > 0 )
17   this.borderWidth = width;
18 }

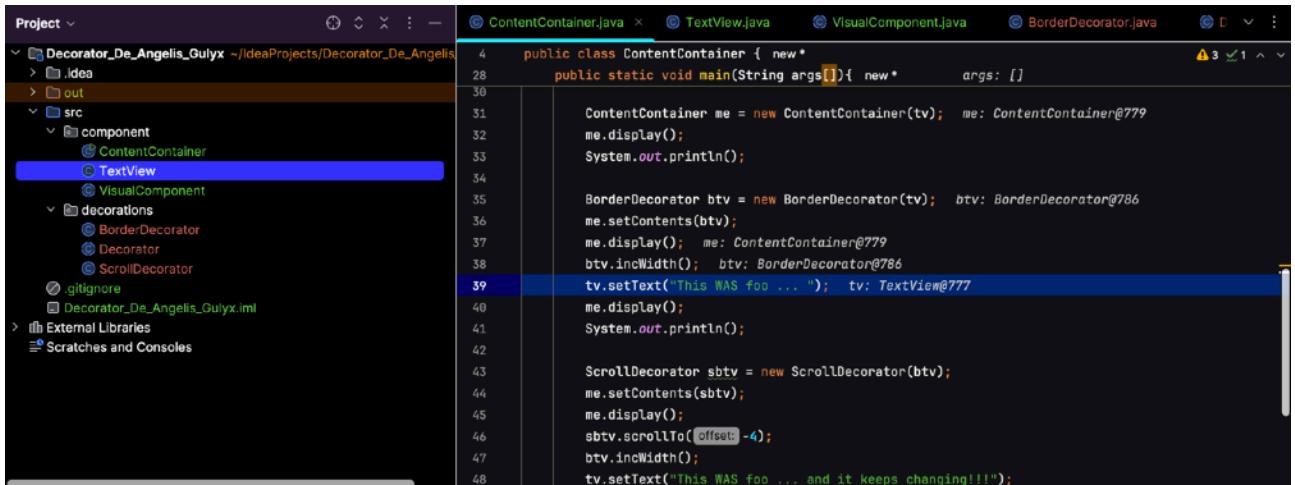
20   public void incWidth(){ 2 usages
21   this.borderWidth += this.tick; borderWidth: 5 tick: 1
22 }

24   public void decWidth(){ no usages
25   if ((this.borderWidth - this.tick) > 0)
26   this.borderWidth -= this.tick;
27   else
28   this.borderWidth = 0;
29 }

31   protected String applyBorder(String input){ 1 usage
32   String output = "[border " + this.borderWidth + "] " + input + "[/border]";
33   return output;
34 }

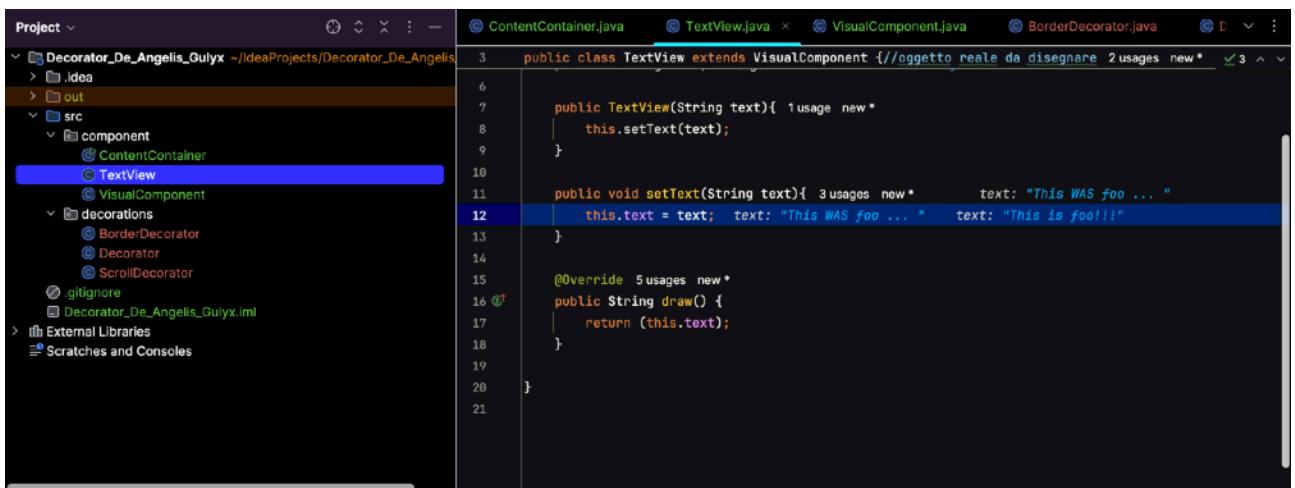
```

Ora super.draw() ha restituito "This is foo!!!", quindi:



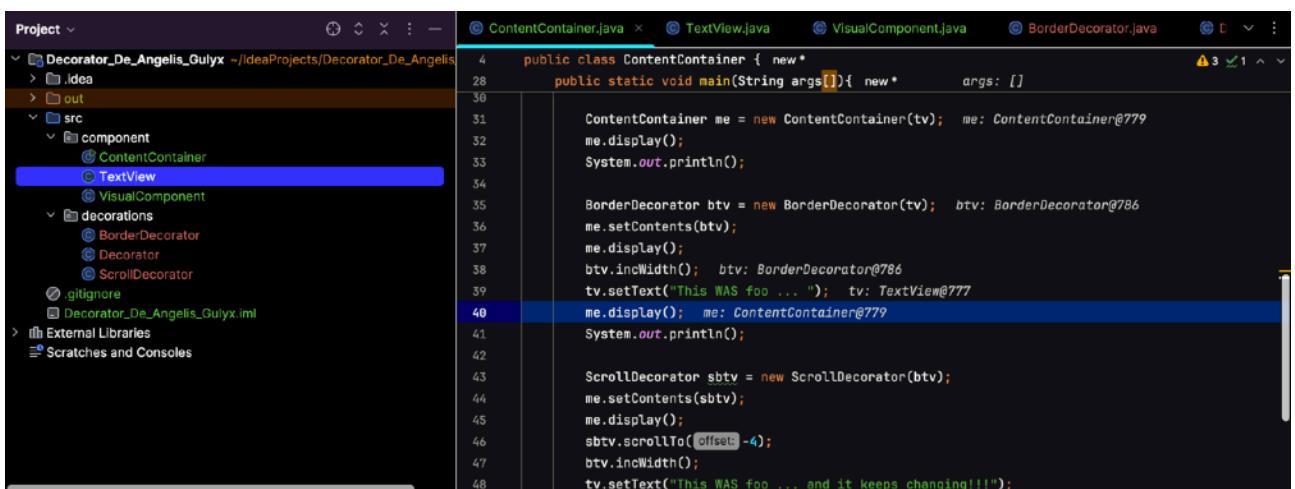
The screenshot shows the IntelliJ IDEA interface with the ContentContainer.java file open. The code is as follows:

```
public class ContentContainer { new*  
    public static void main(String args[]){ new*      args: []  
        ContentContainer me = new ContentContainer(tv); me: ContentContainer@779  
        me.display();  
        System.out.println();  
  
        BorderDecorator btv = new BorderDecorator(tv); btv: BorderDecorator@786  
        me.setContents(btv);  
        me.display(); me: ContentContainer@779  
        btv.incWidth(); btv: BorderDecorator@786  
        tv.setText("This WAS foo ... "); tv: TextView@777  
        me.display();  
        System.out.println();  
  
        ScrollDecorator sbtv = new ScrollDecorator(btv);  
        me.setContents(sbtv);  
        me.display();  
        sbtv.scrollTo(offset: -4);  
        btv.incWidth();  
        tv.setText("This WAS foo ... and it keeps changing!!!");
```



The screenshot shows the IntelliJ IDEA interface with the TextView.java file open. The code is as follows:

```
public class TextView extends VisualComponent { //oggetto reale da disegnare 2 usages new*  
    public TextView(String text){ 1 usage new*  
        this.setText(text);  
    }  
  
    public void setText(String text){ 3 usages new*      text: "This WAS foo ... "  
        this.text = text; text: "This WAS foo ... " text: "This is foo!!!"  
    }  
  
    @Override 5 usages new*  
    public String draw(){  
        return (this.text);  
    }  
}
```



The screenshot shows the IntelliJ IDEA interface with the ContentContainer.java file open. The code is identical to the first screenshot:

```
public class ContentContainer { new*  
    public static void main(String args[]){ new*      args: []  
        ContentContainer me = new ContentContainer(tv); me: ContentContainer@779  
        me.display();  
        System.out.println();  
  
        BorderDecorator btv = new BorderDecorator(tv); btv: BorderDecorator@786  
        me.setContents(btv);  
        me.display();  
        btv.incWidth(); btv: BorderDecorator@786  
        tv.setText("This WAS foo ... "); tv: TextView@777  
        me.display(); me: ContentContainer@779  
        System.out.println();  
  
        ScrollDecorator sbtv = new ScrollDecorator(btv);  
        me.setContents(sbtv);  
        me.display();  
        sbtv.scrollTo(offset: -4);  
        btv.incWidth();  
        tv.setText("This WAS foo ... and it keeps changing!!!");
```

String preliminaryResults = "This is foo!!!";

preliminaryResults = this.applyBorder(preliminaryResults);

```

4  public class ContentContainer { new *
14 }
15
16     public void setContents(VisualComponent contents) { 4 usages new *
17         this.contents = contents;
18     }
19
20     public void display(){ 5 usages new *
21         System.out.println(this.contents.draw());  contents: BorderDecorator@786
22     }
23
24     public String getContents(){ no usages new *
25         return this.contents.draw();
26     }
27
28     public static void main(String args[]){ new *
29         TextView tv = new TextView("This is foo!!!");
30
31         ContentContainer me = new ContentContainer(tv);
32         me.display();
33         System.out.println();
34     }

```

Il bordo viene incrementato di 1.

this.contents = btv (il BorderDecorator)

```

4  public class BorderDecorator extends Decorator { 2 usages
31     protected String applyBorder(String input){ 1 usage
32         String output = "[border " + this.borderWidth + "] " + input + "[/border]";
33         return output;
34     }
35
36     @Override 5 usages
37     public String draw() {
38         String preliminaryResults = super.draw();
39         preliminaryResults = this.applyBorder(preliminaryResults);
40         return preliminaryResults;
41     }
42
43
44 }
45

```

```

1  package decorations;
2
3  import component.*;
4  public abstract class Decorator extends VisualComponent { 2 usages 2 inheritors
5
6      private VisualComponent component; 2 usages  component: TextView@777
7
8      public Decorator( VisualComponent component){ 2 usages
9          this.component = component;
10     }
11
12     @Override 5 usages 2 overrides
13     public String draw() {
14         String resultsFromRedirection = this.component.draw();  component: TextView@777
15         return resultsFromRedirection;
16     }
17
18 }

```

```

Project ~
  Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis
    .idea
    out
    src
      component
        ContentContainer
        TextView
        VisualComponent
      decorations
        BorderDecorator
        Decorator
        ScrollDecorator
      .gitignore
      Decorator_De_Angelis_Gulyx.iml
  External Libraries
  Scratches and Consoles

  TextView.java
  VisualComponent.java
  BorderDecorator.java
  Decorator.java

  3 public class TextView extends VisualComponent { 1 usage new*
  6   public TextView(String text){ 1 usage new*
  7     this.setText(text);
  8   }
  10
  11   public void setText(String text){ 3 usages new*
  12     this.text = text;
  13   }
  14
  15   @Override 5 usages new*
  16   public String draw(){ 1 usage new*
  17     return this.text;  text: "This WAS foo ... "
  18   }
  19
  20 }
  21

```

Ora, this.component = tv, quindi chiama: tv.draw();

```

Project ~
  Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis
    .idea
    out
    src
      component
        ContentContainer
        TextView
        VisualComponent
      decorations
        BorderDecorator
        Decorator
        ScrollDecorator
      .gitignore
      Decorator_De_Angelis_Gulyx.iml
  External Libraries
  Scratches and Consoles

  Decorator.java
  TextView.java
  VisualComponent.java
  BorderDecorator.java

  1 package decorations;
  2
  3 import component.*;
  4 public abstract class Decorator extends VisualComponent { 2 usages 2 inheritors
  5
  6   private VisualComponent component; 2 usages component: TextView@777
  7
  8   public Decorator( VisualComponent component){ 2 usages
  9     this.component = component;
 10   }
 11
 12   @Override 5 usages 2 overrides
 13   public String draw(){ 1 usage new*
 14     String resultsFromRedirection = this.component.draw();  resultsFromRedirection: "This WA
 15     return resultsFromRedirection;  resultsFromRedirection: "This WAS foo ... "
 16   }
 17
 18 }

```

```

Project ~
  Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis
    .idea
    out
    src
      component
        ContentContainer
        TextView
        VisualComponent
      decorations
        BorderDecorator
        Decorator
        ScrollDecorator
      .gitignore
      Decorator_De_Angelis_Gulyx.iml
  External Libraries
  Scratches and Consoles

  BorderDecorator.java
  TextView.java
  VisualComponent.java
  Decorator.java

  4 public class BorderDecorator extends Decorator { 2 usages
 31   protected String applyBorder(String input){ 1 usage
 32     String output = "[border " + this.borederWidth + "] " + input + "[/border]";
 33     return output;
 34   }
 35
 36   @Override 5 usages
 37   public String draw(){ 1 usage new*
 38     String preliminaryResults = super.draw();  preliminaryResults: "This WAS foo ... "
 39     preliminaryResults = this.applyBorder(preliminaryResults);  preliminaryResults: "This WAS fo
 40     return preliminaryResults;
 41   }
 42
 43
 44 }
 45

```

```

Project ~
  Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis
    .idea
    out
    src
      component
        ContentContainer
        TextView
        VisualComponent
      decorations
        BorderDecorator
        Decorator
        ScrollDecorator
      .gitignore
      Decorator_De_Angelis_Gulyx.iml
  External Libraries
  Scratches and Consoles

  BorderDecorator.java
  TextView.java
  VisualComponent.java
  Decorator.java

  4 public class BorderDecorator extends Decorator { 2 usages
 24   public void decWidth(){ no usages
 27     else
 28       this.borederWidth = 0;
 29   }
 30
 31   protected String applyBorder(String input){ 1 usage input: "This WAS foo ... "
 32   |   String output = "[border " + this.borederWidth + "] " + input + "[/border]";  input: "This WA
 33   |   return output;
 34   |
 35   @Override 5 usages
 37   public String draw(){ 1 usage new*

```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
 component
 ContentContainer
 TextView
 VisualComponent
 decorations
 BorderDecorator
 Decorator
 ScrollDecorator
.gitignore
 Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

Border.java TextView.java VisualComponent.java BorderDecorator.java Decorator.java

```
4   public class BorderDecorator extends Decorator { 2 usages
24     public void decWidth(){ no usages
27       else
28         this.borederWidth = 0;
29     }
30
31     protected String applyBorder(String input){ 1 usage      input: "This WAS foo ... "
32       String output = "[border " + this.borederWidth + "] " + input + "[/border]";  input: "This WA
33     |   return output;  output: "[border 6]This WAS foo ... [/border]"
34   }
35
36   @Override 5 usages
37   public String draw() {
38     String preliminaryResults = super.draw();
39     preliminaryResults = this.applyBorder(preliminaryResults);
40     return preliminaryResults;
41   }
42
43
44 }
```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
 component
 ContentContainer
 TextView
 VisualComponent
 decorations
 BorderDecorator
 Decorator
 ScrollDecorator
.gitignore
 Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

Border.java TextView.java VisualComponent.java BorderDecorator.java Decorator.java

```
4   public class BorderDecorator extends Decorator { 2 usages
24     public void decWidth(){ no usages
27       else
28         this.borederWidth = 0;
29     }
30
31     protected String applyBorder(String input){ 1 usage
32       String output = "[border " + this.borederWidth + "] " + input + "[/border]";
33       return output;
34     }
35
36   @Override 5 usages
37   public String draw() {
38     String preliminaryResults = super.draw();  preliminaryResults: "This WAS foo ... "
39     preliminaryResults = this.applyBorder(preliminaryResults);  preliminaryResults: "This WAS "
40     return preliminaryResults;
41   }
42
43
44 }
```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
 component
 ContentContainer
 TextView
 VisualComponent
 decorations
 BorderDecorator
 Decorator
 ScrollDecorator
.gitignore
 Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

ContentContainer.java TextView.java VisualComponent.java BorderDecorator.java Decorator.java

```
4   public class ContentContainer { new*
14
15
16     public void setContents(VisualComponent contents) { 4 usages new*
17       this.contents = contents;
18     }
19
20     public void display(){ 5 usages new*
21       System.out.println(this.contents.draw());  contents: BorderDecorator@786
22     }
23
24     public String getContents(){ no usages new*
25       return this.contents.draw();
26     }
27
28   public static void main(String args[]){ new*
29     TextView tv = new TextView("This is foo!!!");
30
31     ContentContainer me = new ContentContainer(tv);
32     me.display();
33     System.out.println("
```

E ottengo la stampa.

```

public class ContentContainer {
    new*
    public static void main(String args[]){
        new* args: []
    }

    ContentContainer me = new ContentContainer(tv);
    me.display(); //This is foo!!!
    System.out.println();

    BorderDecorator btv = new BorderDecorator(tv);
    btv: BorderDecorator@786
    me.setContents(btv);
    me.display(); // [border 5] This is foo!!! [/border]
    btv.incWidth();
    tv: TextView@777
    tv.setText("This WAS foo ... ");
    me.display(); // [border 6] This WAS foo ... [/border] me: ContentContainer@779
    System.out.println();

    ScrollDecorator sbtv = new ScrollDecorator(btv);
    me.setContents(sbtv);
    me.display();
    sbtv.scrollTo(offset: -4);
    btv.incWidth();
    tv.setText("This WAS foo ... and it keeps changing!!!");
}

```

Fine del primo decorator.

```

public class ContentContainer {
    new*
    public static void main(String args[]){
        new* args: []
    }

    me.setContents(btv);
    me.display(); // [border 5] This is foo!!! [/border]
    btv.incWidth();
    tv: TextView@777
    tv.setText("This WAS foo ... ");
    me.display(); // [border 6] This WAS foo ... [/border] me: ContentContainer@779
    System.out.println();

    ScrollDecorator sbtv = new ScrollDecorator(btv);
    btv: BorderDecorator@786
    me.setContents(sbtv);
    me.display();
    sbtv.scrollTo(offset: -4);
    btv.incWidth();
    tv.setText("This WAS foo ... and it keeps changing!!!");
    me.display();
}

```

Ora entriamo nel punto più interessante del pattern Decorator:

Il Prof sta aggiungendo un secondo livello di decorazione

```

package decorations;
import component.*;
public class ScrollDecorator extends Decorator {
    private int scrollPosition; scrollPosition: 0
    public ScrollDecorator(VisualComponent component) {
        super(component);
        scrollPosition = 0;
    }
    public void scrollTo(int offset){ this.scrollPosition = offset; }

    protected String applyScroll(String input){
        String output = "[scroll " + this.scrollPosition + "] " + input + "[/scroll]";
        return output;
    }
}

```

component vale btv (il BorderDecorator).

```

Project: Decorator_De_Angelis_Gulyx
src
  component
    ContentContainer
    TextView
    VisualComponent
  decorations
    BorderDecorator
    Decorator
    ScrollDecorator
  .gitignore
  Decorator_De_Angelis_Gulyx.iml

File: Decorator.java
Java code:
1 package decorations;
2
3 import component.*;
4
5 public abstract class Decorator extends VisualComponent {
6     private VisualComponent component;
7
8     public Decorator(VisualComponent component) {
9         this.component = component;
10    }
11
12     @Override
13     public String draw() {
14         String resultsFromRedirection = this.component.draw();
15         return resultsFromRedirection;
16     }
17 }
18

```

this.component = btv

```

Project: Decorator_De_Angelis_Gulyx
src
  component
    ContentContainer
    TextView
    VisualComponent
  decorations
    BorderDecorator
    Decorator
    ScrollDecorator
  .gitignore
  Decorator_De_Angelis_Gulyx.iml

File: ScrollDecorator.java
Java code:
1 package decorations;
2
3 import component.*;
4
5 public class ScrollDecorator extends Decorator {
6     private int scrollPosition;
7
8     public ScrollDecorator(VisualComponent component) {
9         super(component);
10        component = BorderDecorator@786
11        scrollPosition = 0;
12    }
13
14     public void scrollTo(int offset) {
15         this.scrollPosition = offset;
16     }
17
18     protected String applyScroll(String input) {
19         String output = "[scroll " + this.scrollPosition + "]"
20         + input + "[/scroll]";
21         return output;
22     }
23

```

Si è formata una catena:

sbtv (ScrollDecorator)

└─ scrollPosition = 0

└─ component → btv (BorderDecorator)

 └─ borderWidth = 6

 └─ tick = 1

 └─ component → tv (TextView)

 └─ text = "This WAS foo ... "

$$\text{sbtv}(\text{btv}(\text{tv}))).$$

The screenshot shows the IntelliJ IDEA interface. On the left, the Project tool window displays the file structure of the 'Decorator_De_Angelis_Gulyx' project. The 'src' directory contains 'component' and 'decoration' packages. 'component' contains 'ContentContainer', 'TextView', and 'VisualComponent' classes. 'decoration' contains 'BorderDecorator', 'Decorator', and 'ScrollDecorator' classes. Other files visible include '.gitignore' and 'Decorator_De_Angelis_Gulyx.iml'. On the right, the Editor tab bar shows tabs for 'ContentContainer.java', 'TextView.java', 'VisualComponent.java', and 'BorderDecorator.java'. The current file is 'ContentContainer.java'. The code implements the Decorator pattern:

```
public class ContentContainer { new*
    public static void main(String args[]){ new*
        args: []
        me.setContents(btv);
        me.display(); // [border 5] This is foo!! [/border]
        btv.incWidth();
        tv.setText("This WAS foo ... "); tv: TextView@777
        me.display(); // [border 6] This WAS foo ... [/border]
        System.out.println();
    }
    ScrollDecorator sbtv = new ScrollDecorator(btv); btv: BorderDecorator@786 sbtv: ScrollDecorato
    me.setContents(sbtv); me: ContentContainer@779 sbtv: ScrollDecorator@812
    me.display();
    sbtv.scrollTo(offset: -4);
    btv.incWidth();
    tv.setText("This WAS foo ... and it keeps changing!!!");
    me.display();
}
```

```
public class ContentContainer { new *
}
public ContentContainer (VisualComponent contents){ 1 usage new *
|   this.setContents(contents);
}
public void setContents(VisualComponent contents) { 4 usages new *      contents: ScrollDecorat
|   this.contents = contents;  contents: ScrollDecorator@812  contents: BorderDecorator@786
}
public void display(){ 5 usages new *
|   System.out.println(this.contents.draw());
}
public String getContents(){ no usages new *
|   return this.contents.draw();
}
public static void main(String args[]){ new *
}
```

this.contents (cioè me.contents) → sbtv

```
public class ContentContainer { new*  
    public static void main(String args[]){ new*  
        args: []  
        BorderDecorator btv = new BorderDecorator(tv); btv: BorderDecorator@786  
        me.setContents(btv);  
        me.display(); // [border 5] This is foo!!! [/border]  
        btv.incWidth();  
        tv.setText("This WAS foo ... "); tv: TextView@777  
        me.display(); // [border 6] This WAS foo ... [/border]  
        System.out.println();  
  
        ScrollDecorator sbtv = new ScrollDecorator(btv); btv: BorderDecorator@786 sbtv: ScrollDecorato  
        me.setContents(sbtv); sbtv: ScrollDecorator@812  
        me.display(); me: ContentContainer@779  
        sbtv.scrollTo(offset: -4);  
        btv.incWidth();  
        tv.setText("This WAS foo ... and it keeps changing!!!");  
        me.display();  
    }  
}
```

```

public class ContentContainer {
    public ContentContainer() {
        this.setContents(null);
    }

    public ContentContainer(VisualComponent contents) {
        this.setContents(contents);
    }

    public void setContents(VisualComponent contents) {
        this.contents = contents;
    }

    public void display() {
        System.out.println(this.contents.draw());
    }

    public String getContents() {
        return this.contents.draw();
    }
}

```

this.contents = sbtv.

System.out.println(sbtv.draw());

```

public class ScrollDecorator extends Decorator {
    public void scrollTo(int offset) {
        this.scrollPosition = offset;
    }

    protected String applyScroll(String input) {
        String output = "[scroll " + this.scrollPosition + "] " + input + "[/scroll]";
        return output;
    }

    @Override
    public String draw() {
        String preliminaryResults = super.draw();
        preliminaryResults = this.applyScroll(preliminaryResults);
        return preliminaryResults;
    }
}

```

super.draw() chiama il metodo della superclasse Decorator:

```

package decorations;

import component.*;

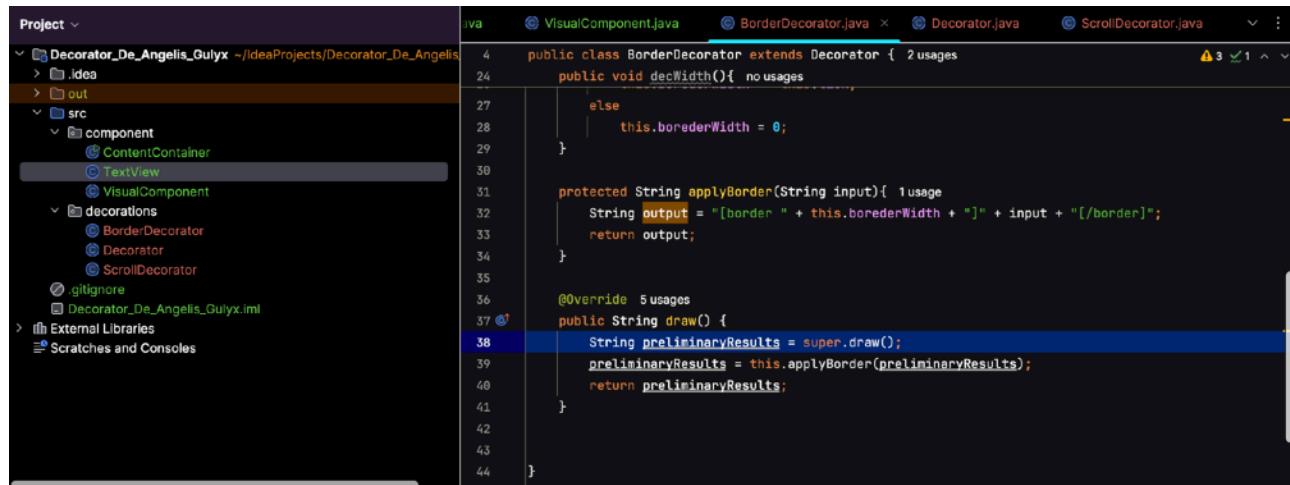
public abstract class Decorator extends VisualComponent {
    private VisualComponent component;

    public Decorator(VisualComponent component) {
        this.component = component;
    }

    @Override
    public String draw() {
        String resultsFromRedirection = this.component.draw();
        return resultsFromRedirection;
    }
}

```

this.component = btv



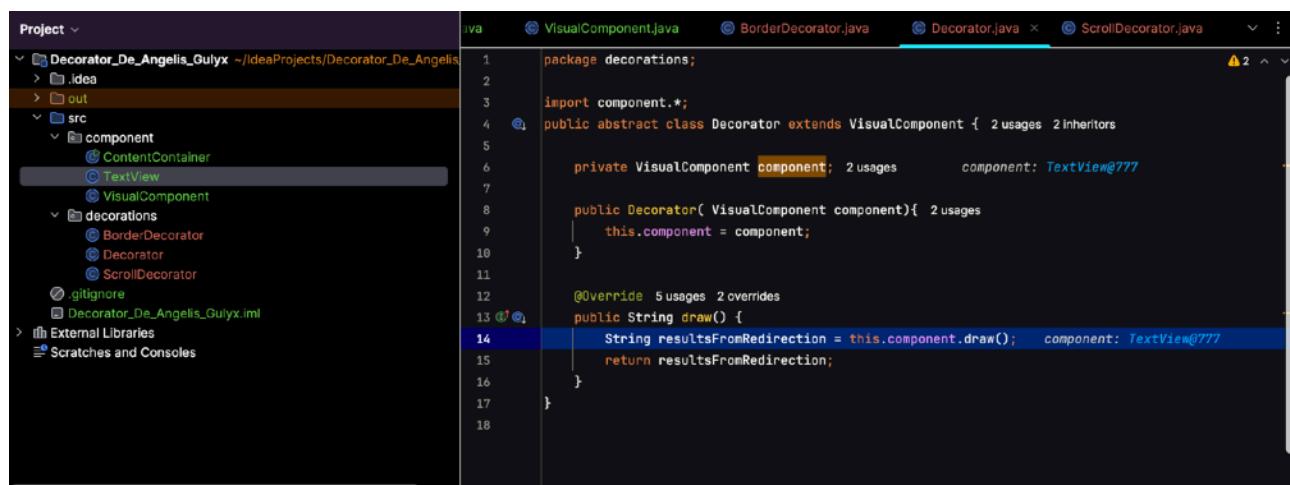
The screenshot shows the IntelliJ IDEA interface with the 'BorderDecorator.java' file open. The code implements the Decorator pattern, specifically adding borders to visual components. It overrides the 'draw()' method to apply a border before calling the superclass's 'draw()' method.

```
public class BorderDecorator extends Decorator {
    public void decoWidth(){ no usages }
    else
        this.borederWidth = 0;
    }

    protected String applyBorder(String input){ 1 usage
        String output = "[border " + this.borederWidth + "]"+ input + "[/border]";
        return output;
    }

    @Override 5 usages
    public String draw() {
        String preliminaryResults = super.draw();
        preliminaryResults = this.applyBorder(preliminaryResults);
        return preliminaryResults;
    }
}
```

super.draw() chiama di nuovo il draw() della superclasse Decorator:



The screenshot shows the IntelliJ IDEA interface with the 'Decorator.java' file open. This abstract class defines the interface for the Decorator pattern, including a component reference and a抽像方法 draw().

```
package decorations;

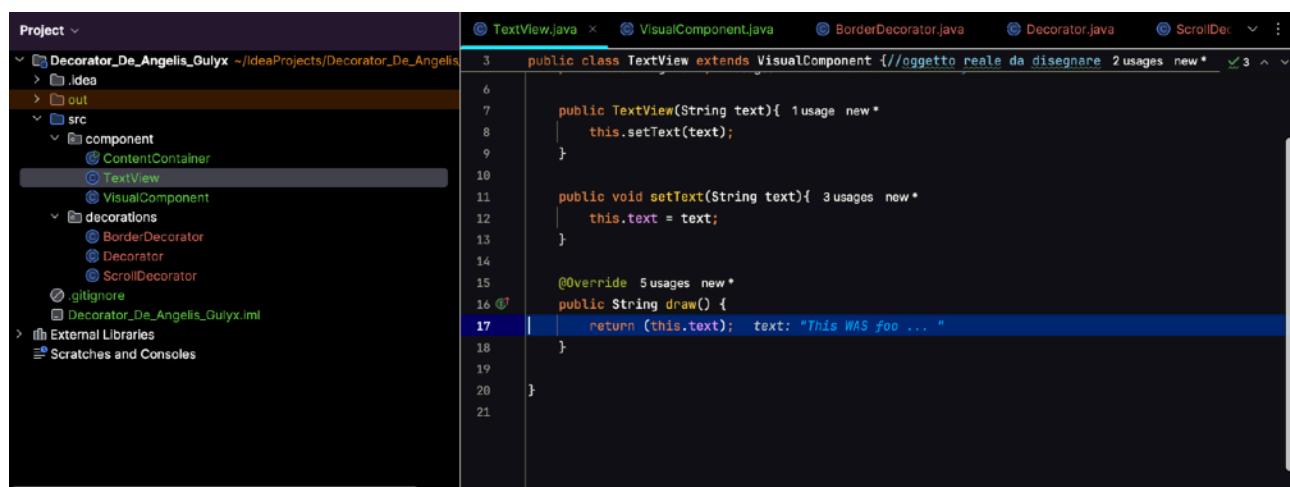
import component.*;
public abstract class Decorator extends VisualComponent { 2 usages 2 inheritors

    private VisualComponent component; 2 usages component: TextView@777

    public Decorator( VisualComponent component){ 2 usages
        this.component = component;
    }

    @Override 5 usages 2 overrides
    public String draw() {
        String resultsFromRedirection = this.component.draw(); component: TextView@777
        return resultsFromRedirection;
    }
}
```

this.component = tv



The screenshot shows the IntelliJ IDEA interface with the 'TextView.java' file open. This class extends 'VisualComponent' and provides a constructor to set initial text and a method to get or set text.

```
public class TextView extends VisualComponent { //oggetto reale da disegnare 2 usages new*
    public TextView(String text){ 1 usage new*
        this.setText(text);
    }

    public void setText(String text){ 3 usages new*
        this.text = text;
    }

    @Override 5 usages new*
    public String draw() {
        return (this.text); text: "This WAS foo ... "
    }
}
```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
component
ContentContainer
TextView
VisualComponent
decorations
BorderDecorator
Decorator
ScrollDecorator
.gitignore
Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

TextView.java VisualComponent.java BorderDecorator.java Decorator.java ScrollDecor...

```
4     public class BorderDecorator extends Decorator { 2 usages
24         public void decWidth(){ no usages
27             else
28                 this.borderWidth = 0;
29         }
30
31         protected String applyBorder(String input){ 1 usage
32             String output = "[border " + this.borderWidth + "]" + input + "[/border]";
33             return output;
34         }
35
36         @Override 5 usages
37         public String draw(){
38             String preliminaryResults = super.draw();
39             preliminaryResults = this.applyBorder(preliminaryResults);
40             return preliminaryResults;
41         }
42
43
44     }
```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
component
ContentContainer
TextView
VisualComponent
decorations
BorderDecorator
Decorator
ScrollDecorator
.gitignore
Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

TextView.java VisualComponent.java BorderDecorator.java Decorator.java ScrollDecor...

```
4     public class BorderDecorator extends Decorator { 2 usages
24         public void decWidth(){ no usages
27             else
28                 this.borderWidth = 0;
29         }
30
31         protected String applyBorder(String input){ 1 usage      input: "This WAS foo ... "
32             String output = "[border " + this.borderWidth + "]" + input + "[/border]";  input: "This WA
33             return output;
34         }
35
36         @Override 5 usages
37         public String draw(){
38             String preliminaryResults = super.draw();
39             preliminaryResults = this.applyBorder(preliminaryResults);
40             return preliminaryResults;
41         }
42
43
44     }
```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
component
ContentContainer
TextView
VisualComponent
decorations
BorderDecorator
Decorator
ScrollDecorator
.gitignore
Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

Java VisualComponent.java BorderDecorator.java Decorator.java ScrollDecorator.java

```
4     public class ScrollDecorator extends Decorator { 2 usages
11     }
12
13     public void scrollTo(int offset){ this.scrollPosition = offset; }
14
15
16     protected String applyScroll(String input){ 1 usage
17         String output = "[scroll " + this.scrollPosition + "]" + input + "[/scroll]";
18         return output;
19     }
20
21
22     @Override 5 usages
23     public String draw(){
24         String preliminaryResults = super.draw();  preliminaryResults: "[border 6]This WAS foo ...
25         preliminaryResults = this.applyScroll(preliminaryResults);  preliminaryResults: "[border 6]
26         return preliminaryResults;
27     }
28
29
30
31     }
```

Project ~

Decorator_De_Angelis_Gulyx ~/ideaProjects/Decorator_De_Angelis

.idea
out
src
component
ContentContainer
TextView
VisualComponent
decorations
BorderDecorator
Decorator
ScrollDecorator
.gitignore
Decorator_De_Angelis_Gulyx.iml

External Libraries
Scratches and Consoles

ContentContainer.java TextView.java VisualComponent.java BorderDecorator.java

```
4     public class ContentContainer { new *
14     }
15
16     public void setContents(VisualComponent contents){ 4 usages new *
17         this.contents = contents;
18     }
19
20     public void display(){ 5 usages new *
21         System.out.println(this.contents.draw());  contents: ScrollDecorator@812
22     }
23
24     public String getContents(){ no usages new *
25         return this.contents.draw();
26     }
27
28     public static void main(String args[]){ new *
29         TextView tv = new TextView("This is foo!!!");
30
31         ContentContainer me = new ContentContainer(tv);
32         me.display(); //This is foo!!!
33         System.out.println("ContentContainer: " + me.getContents());
34     }
35
36
37     }
```

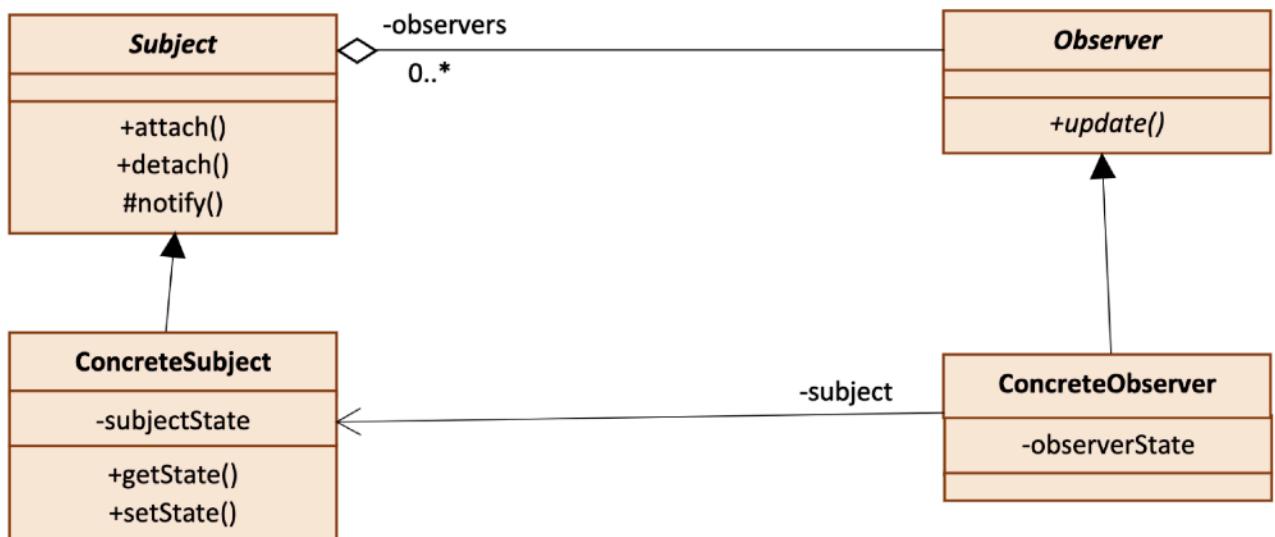
In conclusione non bisogna spaventarsi, perché dietro questa catena di chiamate c'è una logica elegante e potente. L'importante è non spaventarsi davanti ai passaggi ricorsivi: basta seguirli un livello alla volta, ricordando che ogni decoratore non fa altro che "avvolgere" il precedente e aggiungere un piccolo tassello al comportamento finale.

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

6. OBSERVER

Definisce una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo vengono notificati e aggiornati automaticamente.

Struttura:



Quando un oggetto (il *Subject*) cambia stato, tutti gli oggetti interessati (gli *Observer*) vengono **notificati automaticamente** e aggiornati.

Immagina di avere un oggetto principale ad esempio un *sensore di temperatura* e diversi oggetti che vogliono sapere quando la temperatura cambia (display, logger, allarmi, ecc.). Invece di far sì che il sensore chiami manualmente ogni

oggetto, il sensore mantiene una **lista di osservatori**.

Ogni volta che la temperatura cambia, il sensore **notifica** tutti gli iscritti.

```
import java.util.ArrayList;
import java.util.List;

// Subject
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}

// Observer
interface Observer {
    void update(int newState);
}

// Concrete Subject
class Sensor implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int temperature;

    public void setTemperature(int temp) {
        this.temperature = temp;
        notifyObservers();
    }

    public void attach(Observer o) { observers.add(o); }
    public void detach(Observer o) { observers.remove(o); }

    public void notifyObservers() {
        for (Observer o : observers)
            o.update(temperature);
    }
}

// Concrete Observers
class Display implements Observer {
    public void update(int newState) {
        System.out.println("Display: nuova temperatura = " +
newState);
    }
}
```

```

}

class Alarm implements Observer {
    public void update(int newState) {
        if (newState > 30)
            System.out.println("Allarme: temperatura troppo
alta!");
    }
}

// Test
public class Main {
    public static void main(String[] args) {
        Sensor s = new Sensor();
        s.attach(new Display());
        s.attach(new Alarm());

        s.setTemperature(25);
        s.setTemperature(35);
    }
}

```

Output:

```

Display: nuova temperatura = 25
Display: nuova temperatura = 35
Allarme: temperatura troppo alta!

```

Subject (Osservato)

È l'oggetto che ha dei dati che possono cambiare.

Deve permettere agli osservatori di:

- registrarsi (attach)
- disiscriversi (detach)
- ricevere aggiornamenti (notify)

Observer (Osservatore)

È chi “si iscrive” al subject per essere avvisato dei cambiamenti.

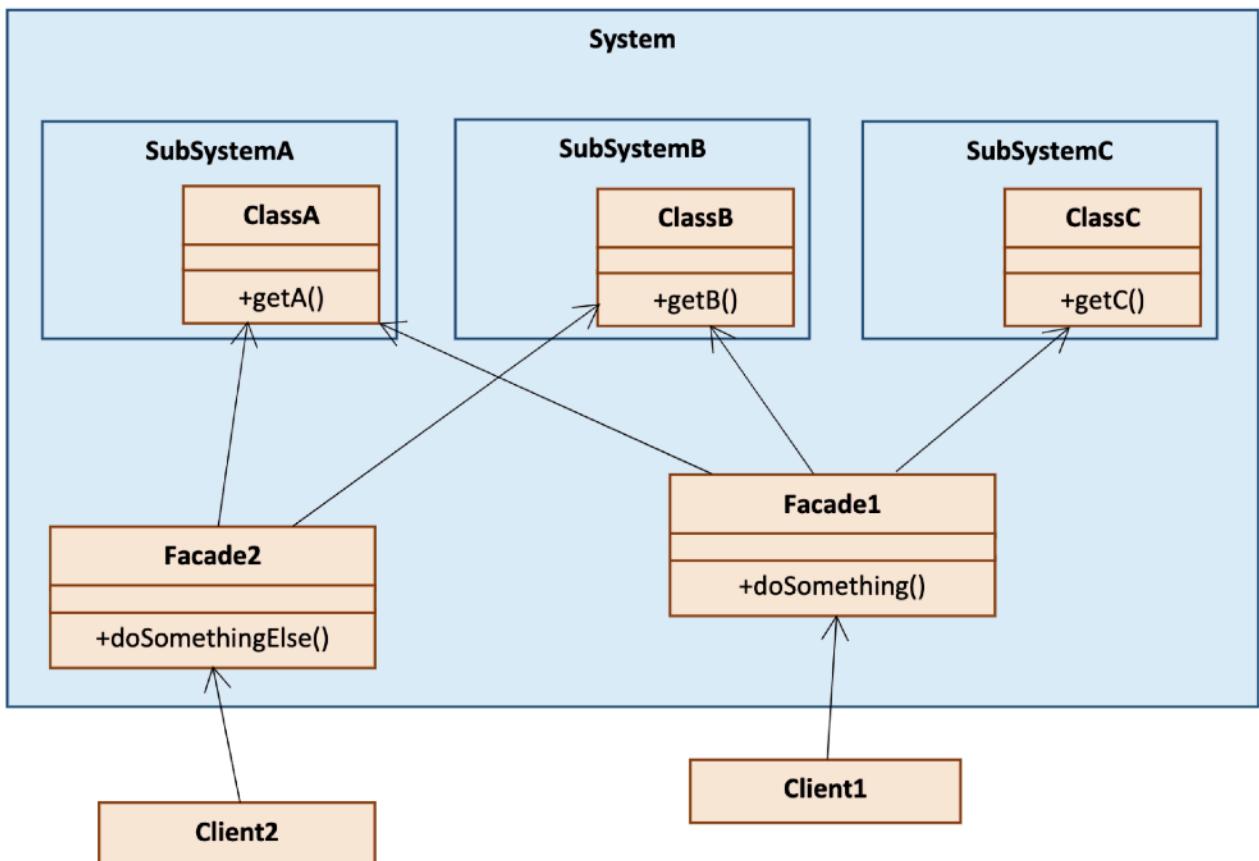
Ha un metodo update() che viene chiamato ogni volta che il subject cambia.

Ruolo	Cosa fa
Subject	Tiene una lista di osservatori e li notifica quando cambia stato.
Observer	Riceve notifiche e si aggiorna di conseguenza.

7 . FACADE

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

È come un *pannello di controllo* che ti permette di usare facilmente un insieme di classi complesse, senza doverle conoscere tutte nel dettaglio.



```
public class FacadeExample {

    // --- Sottosistemi complessi ---
    static class CPU {
        public void freeze() {
            System.out.println("CPU: freeze()");
        }
        public void jump(long position) {
            System.out.println("CPU: jump to " + position);
        }
        public void execute() {
            System.out.println("CPU: execute()");
        }
    }

    static class Memory {
        public void load(long position, byte[] data) {
            System.out.println("Memory: load " + data.length
+ " bytes at position " + position);
        }
    }

    static class HardDrive {
        public byte[] read(long lba, int size) {
            System.out.println("HardDrive: read " + size + " bytes from sector " + lba);
            return new byte[size];
        }
    }

    // --- Facade ---
    static class ComputerFacade {
        private final CPU cpu;
        private final Memory memory;
        private final HardDrive hd;

        public ComputerFacade() {
            this.cpu = new CPU();
            this.memory = new Memory();
            this.hd = new HardDrive();
        }

        public void startComputer() {
            System.out.println(" Avvio del computer in corso... ");
        }
    }
}
```

```

        cpu.freeze();
        byte[] bootData = hd.read(0, 1024);
        memory.load(0, bootData);
        cpu.jump(0);
        cpu.execute();
        System.out.println(" Computer avviato con
successo!");
    }
}

// --- Client ---
public static void main(String[] args) {
    ComputerFacade computer = new ComputerFacade();
    computer.startComputer();
}
}

```

Output:

```

Avvio del computer in corso...
    CPU: freeze()
HardDrive: read 1024 bytes from sector 0
    Memory: load 1024 bytes at position 0
        CPU: jump to 0
        CPU: execute()
Computer avviato con successo!

```

Senza la *Facade*, il codice client dovrebbe sapere come usare CPU, Memory e HardDrive e in quale ordine chiamarli.

Con la *Facade*, tutto viene incapsulato in un'unica interfaccia semplice.

Le classi CPU, Memory e HardDrive rappresentano il sottosistema complesso. Ognuna fa un piccolo pezzo di lavoro, ma non sono semplici da coordinare.

La classe ComputerFacade è la facciata (*facade*).

Nasconde la complessità del sottosistema e fornisce un unico metodo semplice:
`startComputer()`.

Il main (client) **non conosce** i dettagli interni del sistema.

Chiama una sola funzione, e la Facade si occupa di tutto il resto.

I Client comunicano con il sistema attraverso l'interfaccia comune esportata (i.e. Façade). **I Client non hanno in alcun modo accesso agli oggetti dei sottosistemi.**