



◀ Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java

▲
Torna su

Singleton: Pattern o Anti-Pattern? ▶



INTRODUZIONE

Ciao Coders,

dopo un primo sguardo nei precedenti articoli ai pattern creazionali più noti, oggi vi parlo di uno dei pattern strutturali più utilizzati: il **Decorator**. Quando è stata introdotta la programmazione orientata a oggetti, l'ereditarietà era il modello principale utilizzato per estendere la funzionalità dell'oggetto. Non sempre però è l'approccio più giusto. Infatti è stato dimostrato che estendere gli oggetti usando l'ereditarietà spesso si traduce in una gerarchia di classi che esplode, fenomeno noto come **Exploding class hierarchy**. Giusto per farvi capire il fenomeno della **Exploding class hierarchy** voglio riportarvi uno schema di



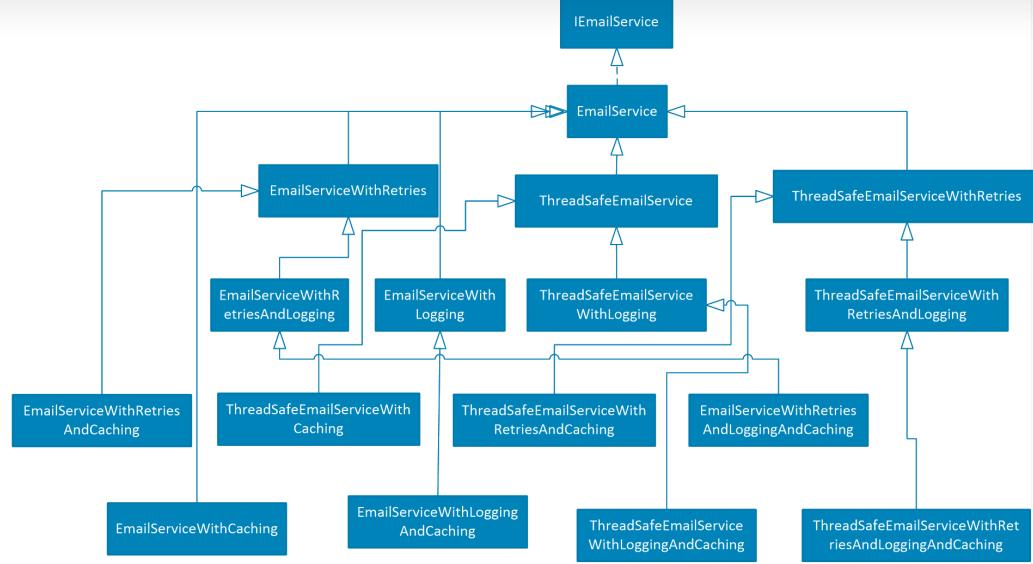
Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java



Torna su



Singleton: Pattern o Anti-Pattern?



Unico mio commento a tutto questo: CAOS !!!

In aggiunta a quanto detto dobbiamo considerare che diversi linguaggi di programmazione popolari come Java e C# non supportano l'ereditarietà multipla, il che limita i vantaggi di questo approccio.

Il design pattern **Decorator** fornisce un'alternativa flessibile all'ereditarietà per estendere la funzionalità degli oggetti. Tale pattern consente di arricchire dinamicamente, a **run-time**, un oggetto con nuove funzionalità: è possibile impilare uno o più decorator uno sopra l'altro, ciascuno aggiungendo nuove



- Un Decorator agisce a runtime a differenza dell'ereditarietà che estende con le sottoclassi il comportamento della classe padre in fase di compilazione.
- Un Decorator può operare su qualsiasi implementazione di una determinata interfaccia, eliminando la necessità di creare sottoclassi di un'intera gerarchia di classi.
- La sottoclasse aggiunge comportamento al momento della compilazione e la modifica interessa tutte le istanze della classe originale; il decorator pattern può fornire nuovi comportamenti in fase di esecuzione per i singoli oggetti.
- L'uso del modello decorator porta a codice pulito e testabile. I servizi creati con l'ereditarietà non possono essere testati separatamente dalla sua classe padre perché non esiste un meccanismo per sostituire una classe padre con uno stub.

Il decorator pattern è quindi molto utile e utilizzato dai programmatore più esperti, i quali lo utilizzano al posto della ereditarietà in situazioni in cui è necessario aggiungere/modificare a runtime il comportamento di un oggetto senza scomodare

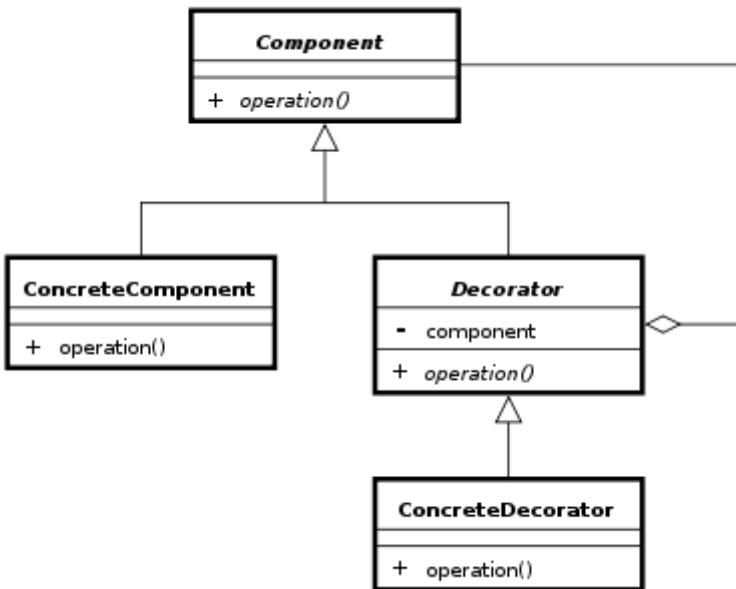


```
1. java.io.BufferedReader;  
2. java.io.BufferedWriter;  
3. java.io.FileReader;  
4. java.io.Reader;
```



Andiamo adesso a descrivere i componenti del Decorato Pattern e il loro funzionamento riportando il diagramma UML originale del GoF.

SCHEMA DEL PATTERN



Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java



Torna su



Singleton: Pattern o Anti-Pattern? >



- **Component**: rappresenta l'interfaccia dell'oggetto che dovrà essere decorato dinamicamente,
- **ConcreteComponent**: rappresenta l'oggetto a cui andranno aggiunte le nuove funzionalità,
- **Decorator**: rappresenta l'interfaccia tra il Component e i ConcreteDecorator, possiede un riferimento al Component e un'interfaccia a esso conforme,
- **ConcreteDecorator**: rappresentano gli oggetti che aggiungono le nuove funzionalità ai ConcreteComponent.

Passiamo adesso alla pratica con un esempio di utilizzo del Decorator in Java

ESEMPIO - PANINOTECA ROSSI



Kotlin: Che cos'è e cosa serve sapere ai programmatori Java



Torna su



Singleton: Pattern o Anti-Pattern?

Il dominio applicativo di questo esempio è quello della paninoteca Rossi che vende diversi tipi di panini, i quali sono identificati da un nome e un prezzo. Nel nostro esempio prevediamo due tipologie di panino: Hamburger e CheeseBurger. Entrambe le tipologie di Panino non hanno al loro interno la Maionese e il Ketchup; se il cliente vuole aggiungere uno o più ingredienti extra deve pagare una somma aggiuntiva. Il nostro esercizio è quello di stampare a video per ogni panino il nome e il relativo prezzo (considerando in caso di aggiunta di ingredienti extra anche quest'ultimi nel prezzo totale). Senza l'uso del decorator, anche un dominio applicativo così semplice e piccolo, comporterebbe una gerarchia di classi medio-grande; se volessimo modellare l'aggiunta di un ingrediente con l'ereditarietà avremmo le seguenti Classi:

- Hamburger (classe per modellare un Hamburger)
 - HamburgerMaionese (sotto classe di Hamburger per modellare un Hamburger su cui è stato aggiunto l'ingrediente extra Maionese)
 - HamburgerKetchup (sotto classe di Hamburger per modellare un Hamburger su cui è stato aggiunto l'ingrediente extra Ketchup)



- CheeseBurger (classe per modellare un CheeseBurger)
 - CheeseBurgerMaionese (sotto classe di CheeseBurger per modellare un Hamburger su cui è stato aggiunto l'ingrediente extra Maionese)
 - CheeseBurgerKetchup (sotto classe di CheeseBurger per modellare un Hamburger su cui è stato aggiunto l'ingrediente extra Ketchup)
 - CheeseBurgerKetchupMaionese (sotto classe di CheeseBurger per modellare un Hamburger su cui è stato aggiunto l'ingrediente extra Ketchup e quello Maionese)

9 classi per modellare un dominio così semplice! E se il numero delle tipologie di ingredienti aggiuntivi crescesse? La gerarchie delle classi esploderebbe (**Exploding class hierarchy**).

Risolviamo questo problema con il decorator pattern:

Component: Prima di tutto abbiamo bisogno di una prima classe astratta da cui deriveranno tutti i prodotti della nostra paninoteca: la classe **Consumation** modellerà la generica consumazione del



```
1. public abstract class Consumation {  
2.  
3.     String productName = "";  
4.  
5.     public String getProductName() {  
6.         return productName;  
7.     }  
8.  
9.     public abstract double getPrice();  
10.}
```



Decorator: Abbiamo poi bisogno della classe

ExtraAdditionDecorator che modellerà ogni possibile aggiunta non prevista ad un prodotto: ad esempio un ingrediente aggiuntivo in un panino. Tale classe fa da classe base per i Decorator, ovvero modella tutti gli ingredienti aggiuntivi.

```
1. public abstract class ExtraAdditionDecorator extends  
Consumation {  
2.     protected Consumation consumation;  
3.  
4.     @Override  
5.     public abstract String getProductName();  
6.  
7. }
```

ConcreteComponent: Andiamo adesso a definire un po' di entità del nostro dominio; ovvero dei prodotti venduti alla paninoteca Rossi. Nel nostro esempio vengono vendute due tipologie di panino:

```
1. public class CheeseBurger extends Consumation {
```



Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java



Torna su



Singleton: Pattern o Anti-Pattern?

```
8. }
9. }
```

```
1. public class Hamburger extends Consumation {
2.     public Hamburger() {
3.         productName = "Hamburger";
4.     }
5.     @Override
6.     public double getPrice() {
7.         return 2.00;
8.     }
9. }
```

ConcreteDecorator: Infine andiamo a definire un paio di decorator concreti; nel nostro esempio due tipi di ingredienti non previsti e compresi nei prezzi del menu: ExtraKetchupDecorator e ExtraMaioneseDecorator

```
1. public class ExtraMaioneseDecorator extends
ExtraAdditionDecorator {
2.
3.
4.     public ExtraMaioneseDecorator(Consumation consumation) {
5.         this.consumation = consumation;
6.     }
7.
8.     @Override
9.     public String getProductName() {
10.        return consumation.getProductName() + " con extra
maionese";
11.    }
12.
13.    @Override
14.    public double getPrice() {
15.        return consumation.getPrice() + 0.20;
16.    }
17. }
```

```

6. }
7.
8.     @Override
9.     public String getProductName() {
10.         return consumation.getProductName() + " con extra
ketchup";
11.    }
12.
13.    @Override
14.    public double getPrice() {
15.        return consumation.getPrice() + 0.10;
16.    }
17. }

```

MAIN:Riportiamo adesso un semplice main:

```

1. public class Main {
2.
3.     public static void main(String[] args) {
4.         //Hamburger
5.         Consumation hamburger = new Hamburger();
6.         System.out.println("Prodotto:" +
hamburger.productName +
7.             " di prezzo " + String.format("%.2f", hamburger.getPrice()
));
8.
9.
10.        Consumation cheeseburger = new CheeseBurger();
11.
12.
13.        //voglio aggiungere la maionese al burger
14.        Consumation hamburgerConMaionese = new
ExtraMaioneseDecorator(hamburger);
15.        System.out.println("Prodotto:" +
hamburgerConMaionese.getProductName() +
16.            " di prezzo " + String.format("%.2f",
hamburgerConMaionese.getPrice()));
17.
18.        //voglio aggiungere la maionese e il ketchup al burger
19.        Consumation hamburgerConMaioneseKetchup = new
ExtraKetchupDecorator(new ExtraMaioneseDecorator(hamburger));
20.        System.out.println("Prodotto:" +
hamburgerConMaioneseKetchup.getProductName() +
21.

```

```
26.     cheeseburgerConMaionese.getProductName() +  
27.         " di prezzo " + String.format("%.2f",  
cheeseburgerConMaionese.getPrice()));  
28.  
29.    }  
30. }
```



Come mostra il codice precedente, all'interno del metodo main vengono create a runtime due istanze di Hamburger e CheeseBurger ma anche tre istanze di panino non definite come componenti concrete del dominio ma create grazie all'uso di decorator:

- hamburgerConMaionese
- hamburgerConMaioneseKetchup
- cheeseburgerConMaionese

Per ognuno di questi panini andremo a visualizzare a video il nome del prodotto (comprensivo di ingrediente extra se utilizzato) e il prezzo.

Ecco l'output del main

```
1. Prodotto:Hamburger di prezzo 2,00  
2. Prodotto:Hamburger con extra maionese di prezzo 2,20  
3. Prodotto:Hamburger con extra maionese con extra ketchup di  
prezzo 2,30
```



Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java



Torna su



Singleton: Pattern o Anti-Pattern?

extra; il tutto utilizzando il decorator pattern in alternativa alla ereditarietà.



CONCLUSIONI

Abbiamo mostrato le potenzialità di questo pattern; ci tengo a precisare che oltre a semplificare la gerarchia delle classi e migliorare la testabilità, il decorator pattern incoraggia gli sviluppatori a scrivere codice che rispetta i principi di progettazione SOLID. Infatti il decorator pattern è spesso utile per aderire al (Principio di singola responsabilità) , in quanto consente di dividere la funzionalità tra classi con aree di interesse uniche e distinte aggiungendo nuove funzionalità ai nuovi oggetti senza modificare le classi esistenti (Open-Closed Principle). Qualcuno penserà: è il Santo Graal dei pattern? La risposta è no, come ogni volta rispondo ai commenti sui design pattern. Non esiste l'anti pattern e il Santo Graal dei pattern. Ogni pattern ha il suo caso di utilizzo dove l'utilizzo semplifica la vita e la qualità del codice ma in caso di abuso e uso in situazioni per cui non è stato progettato, diventa una vera e propria complicazione. Ha quindi anche il decorator dei potenziali svantaggi:



Kotlin: Che cos'è e cosa serve sapere ai programmatore Java



Torna su



Singleton: Pattern o Anti-Pattern?

aggiungono alcun comportamento aggiuntivo devono essere implementati come metodi di inoltro per mantenere il comportamento esistente. Al contrario, l'ereditarietà richiede solo sottoclassi per implementare metodi che modificano o estendono il comportamento della classe base.

- I decorator possono complicare il processo di creazione dell'istanza del componente perché non solo devi istanziare il componente ma avvolgerlo anche in un certo numero di decorator.
- Può essere complicato avere decorator che tengono traccia di altri decorator perché guardare indietro in più strati della catena dei decorator inizia a spingere il modello decorator oltre il suo vero intento.

Consiglio quindi di utilizzare il decorator nelle seguenti situazioni:

- Quando le responsabilità e i comportamenti degli oggetti dovrebbero essere dinamicamente modificabili.
- Le implementazioni concrete dovrebbero essere disaccoppiate da responsabilità e comportamenti.



mano che aggiungi più comportamenti a una classe base, ti troverai presto a gestire l'incubo della manutenzione, poiché viene creata una nuova classe per ogni combinazione possibile. In questi casi il decorator pattern fornisce un'alternativa migliore a troppe sottoclassi e alla temuta **Exploding class hierarchy**.

BIBLIOGRAFIA

- Design Patterns: Elements of Reusable Object-Oriented Software (Autori: Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm)
- [is-inheritance-dead](#)
- <https://neilmorgan.wordpress.com/2010/02/07/decorator-pattern-pros-and-cons/>

decorator java OOP pattern



A PROPOSITO DI ME

Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java

▲
Torna su

Singleton: Pattern o Anti-Pattern? ▶



DARIO FRONGILLO



Fondatore di Italiancoders e dell'iniziativa devtalks.
Software architect con un forte background in Java,
Architettura REST, framework Spring , Database
Design, progettazione e sviluppo di SPA e RIA Web
application con framework Javascript. Attualmente
mi occupo di sviluppo soluzioni software in ambito
Banking e Finance: in particolare progettazione e
sviluppo di applicativi web based per la
realizzazione di sistemi di trading, interfacce con i
mercati finanziari e di servizi real time.

ALTRO



Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java

▲
Torna su

Singleton: Pattern o Anti-Pattern? ➤

Nei bilanci salute e risultati: Axa attiva iniziative per garantire l'equilibrio psico-fisico dei 1800 addetti

La Repubblica



Le razze di cani più costose al mondo

Il Mondo dei Cani

Prima di incontrare persone, si raccomanda l'uso di questo prodotto antivirale scientificamente approvato

Agi

Many failed before. Will you complete the Trial?

Hero Wars

Siediti prima di vedere la villa dove vive Maria De Filippi

RetroPages

I segreti dei 20 più grandi trucchi di magia finalmente svelati

7 Commenti

ItalianCoders

normativa sulla privacy 1 Accedi

Consiglia

Tweet

Condividi

Ordina dal più recente



Partecipa alla discussione...

ENTRA CON

O REGISTRATI SU DISQUS

Nome



Kotlin: Che cos'è e cosa serve sapere ai programmatori Java



Torna su

Singleton: Pattern o Anti-Pattern?

pattern lo uccido. Inutile complicazione.



Inoltre se progettata bene NON DEVE succedere che per 1 panino con varianti si abbiano n sottoclassi. sono solo proprietà.

In breve, grazie dell'articolo, ma l'esempio su cui hai applicato il pattern mi indice, di nuovo, ad evitarlo

^ | v • Rispondi • Condividi ›



Magallo • 2 anni fa

Cito te stesso:

"I decorator possono complicare il processo di creazione dell'istanza del componente perché non solo devi istanziare il componente ma avvolgerlo anche in un certo numero di decorator."

Esattamente la stessa cosa che ho pensato mentre leggevo l'articolo quando sono arrivato a leggere l'esempio del main. Effettivamente è veramente "brutto" veder create istanze di panini in quel modo. C'è un modo (magari usando in combinazione altri design pattern), per rendere più elegante questa parte che rappresenta in effetti un grosso inconveniente?

1 ^ | v • Rispondi • Condividi ›



Supermenn • 2 anni fa • edited

Mi piace l'esempio terra-terra, facile da immaginare e da tenere in mente tutto insieme anche per un testone come me.

P.s. Spero che Rossi si decida a fare anche i cari vecchi panini con la mortadella o prosciutto decorati con provola o rucola :D

2 ^ | v • Rispondi • Condividi ›



Paolo Rizzello admin ➔ Supermenn • 2 anni fa



Kotlin: Che cos'è e cosa serve sapere ai programmatore Java



Torna su



Singleton: Pattern o Anti-Pattern? ➔



Articolo ben fatto, chiaro, completo e mirato!!! Grazie mille! Continua così!

^ | v • Rispondi • Condividi >



Ivan86to • 3 anni fa

Ciao, ottimo articolo, sto sviluppando un piccolo ERP con funzioni di contabilità e fatturazione, devo poter ampliare le funzionalità del sistema in base alle richieste dei vari clienti ma ci terrei a tenere il core unico, secondo te il pattern decorator è quello che può fare al mio caso? Leggendo il tuo articolo mi pare di sì tu cosa ne pensi?

^ | v • Rispondi • Condividi >



Mattia Marchiani • 3 anni fa

I NOSTRI PARTNER



Kotlin: Che cos'è e cosa serve sapere ai programmatori Java



Torna su



Singleton: Pattern o Anti-Pattern?



Privacy Policy



Kotlin: Che cos'è e cosa serve sapere ai
programmatori Java



Torna su

Singleton: Pattern o Anti-Pattern?