Decorator

Strutturale

Motivazione

Un oggetto deve implementare una operazione composta da un numero **molteplice e variabile** di singole responsabilità.

- Ad es. disegnare un oggetto grafico **TextView** che può avere anche, opzionalmente:
  - Barra scorrimento verticale
  - Barra scorrimento orizzontale
  - Bordo 3D
  - Bordo piatto

Prevedere nella sua classe l'implementazione di tutte le responsabilità che servono significherebbe avere per essa troppe responsabilità

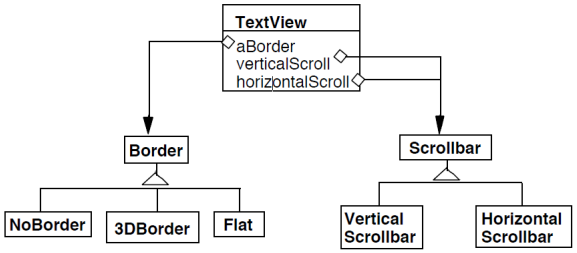
Sopratutto se è una classe posizionata in alto nella gerarchia

Estendere la base

- Dichiaro la classe base
- Derivo una sottoclasse per ogni possibile combinazione di opzioni
- Proliferazione delle classi
- codice duplicato
- Alternative?

- TextView
- TextViewWithNoBorder&SideScrollbar
- TextViewWithNoBorder&BottomScrollbar
- TextViewWithNoBorder&Bottom&SideScrollbar
- TextViewWith3DBorder
- TextViewWith3DBorder&SideScrollbar
- . . .

Soluzione 1, non ottimale



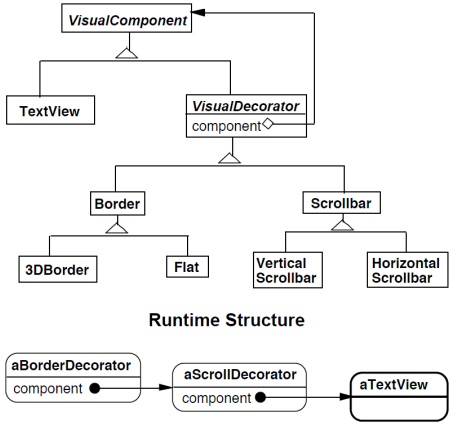

```
class TextView {
    Border myBorder;
    ScrollBar verticalBar;
    ScrollBar horizontalBar;

    public void draw() {
        myBorder.draw();
        verticalBar.draw();
        horizontalBar.draw();
        //code to draw self
    }
etc.
}
```

- Uso la composizione invece dell'ereditarietà
- Ho una sola classe per ogni opzione aggiuntiva

- Ma la classe textview dipende da tutte le tipologie di classi di opzioni
- Aggiungere una opzione richiede la modifica di TextView
- E di ogni altra classe cui l'opzione era applicabile…

Soluzione 2: Decorator pattern

- Definisco l'interfaccia **Component**, implementata dal ***componente base*** e da un ***Decorator***
- Nel Decorator ho un riferimento ad un componente successivo: un altro decorator o il componente base.
- Tutte le tipologie di opzioni sono sottoclassi di Decorator (sono dei decoratori)
- TextView ha al suo interno solo il suo codice, non ha né bordo né barre
- Creo a run-time una catena di funzionalità successive, conclusa dalla classe base



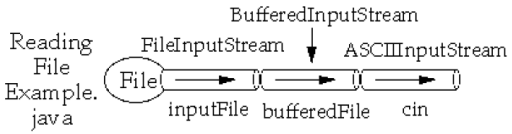
Decorator

- Con il pattern Decorator
  - Se si vuol aggiungere la proprietà `Bordo` al componente `Testo:`
    - Inserisco il componente `Testo` dentro un altro oggetto, che a sua volta aggiunge il `Bordo`

  - Il client ottiene un testo incorniciato, **interagendo solo con l'oggetto più esterno**
  - `DecoratorBordo` usa i servizi della componente `Testo` e aggiunge la sua attività **prima o dopo** l'invio della richiesta ad esso (obj. oriented recursion)

Java I/O stream library

Usato nelle liberie Java per l'i/o da file, per la gestione dei fonts, per le interfacce utente AWT/Swing



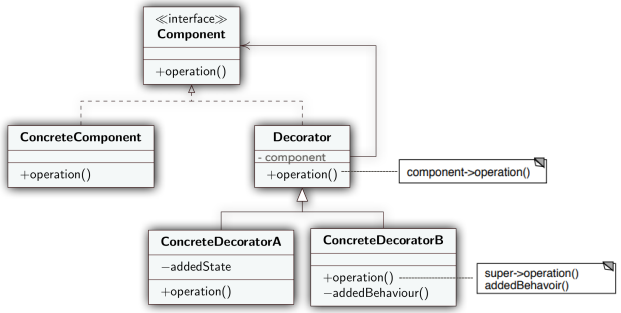
- Da non confondere con i nuovi streams funzionali in java 8
- Esempio: ReadingFileExample

Decorator

- **Intento:** Modificare il comportamento di un oggetto dinamicamente.
  - L'oggetto cambia tipo e comportamento
  - Alternativa all'uso di sottoclassi per estendere implementazioni
  - **Flessibile,** perchè definisco la funzionalità complessiva a run-time
  - Avvolgo (wrap) il componente base in un altro oggetto che aggiunge implementazione alla sua.
  - L'oggetto wrapper **è un decorator** e la sua presenza **è trasparente ai client,** che lo usano attraverso l'interfaccia Component

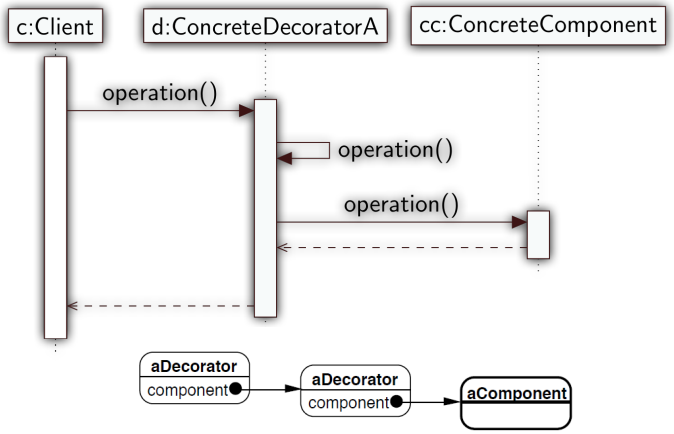
Decorator: struttura

- **Component**: interfaccia comune per comporre dinamicamente l'implementazione di operation()
- **ConcreteComponent:** l'oggetto base cui aggiungere responsabilità
- Fornisce l'implementazione fissa di operation()
- Termina la catena di oggetti composti



- **Decorator:** un Component con un riferimento a un ulteriore Component.
- Implementa operation() inoltrando la richiesta al suo Component successore
- **ConcreteDecorator:** implementa una responsabilità aggiuntiva per il Component e svolge la sua operazione prima e/o dopo l'inoltro della richiesta

Interazioni



Implementazione

```
interface Component {
    public void operation();
}
```

```
class ConcreteComponent implements Component {
    @Override public void operation() {
        // impl. base dell'operazione
    }
}
```

```
class Decorator implements Component {
    private final Component nextComp;

    public Decorator(Component c) {
        nextComp = c;
    }
    @Override public void operation() {
        nextComp.operation();
    }
}
```

```
class ConcreteDecoratorA extends Decorator {

    public ConcreteDecoratorA(Component c) {
        super(c);
    }
    @Override public void operation() {
        super.operation();
        // ...funzionalità aggiuntiva
    }
}
```

Set-up di un oggetto (catena) con funzionalità estesa:

```
Component c = new ConcreteDecoratorA(new ConcreteComponent());
```


Esempio su vscode: Applicazione Messaggi

Conseguenze

- Scelta forzata se la creazione di sottoclassi non è praticabile, per il numero elevato di varianti, o se si vuole poter estendere anche in seguito
- Si avranno **tante piccole classi**, indipendenti e focalizzate su un solo compito, molto specifico
- A run-time, **interconnetto tanti piccoli oggetti in sequenza**, come i passi di un algoritmo, ottenendo un oggetto complesso che offre una funzionalità **nuova,** più elaborata delle esistenti
- Posso **aggiungere, sottrarre, spostare, ripetere** a piacimento le istanze di micro funzionalità, a tempo d'esecuzione, componendo o **modificando l'algoritmo eseguito**
- Molto più flessibile di un algoritmo da scrivere a priori in una classe
- L'oggetto, la catena di oggetti, che implementa l'algoritmo non ha un tipo (ADT) nominale, ma solo la facciata d'uso (Component): ho creato un **prototipo**

State vs Decorator

- **Change Skin vs change Guts** : modifico a run-time il comportamento di un oggetto, (l'effetto di un messaggio ad un oggetto) come nel pattern State, ma:
  - L'oggetto con cui il client interagisce cambia tipo (cambio pelle)
  - Con State, il cambio di oggetto è solo interno (cambio viscere)
  - Qui aggiungo/sottraggo funzionalità al comportamento base
  - Con State, *sostituisco* (interamente) la funzionalità del caso base con alternative
  - Qui, istanzio di volta in volta solo le funzionalità aggiuntive che uso
  - Con State, tutti gli stati sono necessari per avere un funzionamento completo