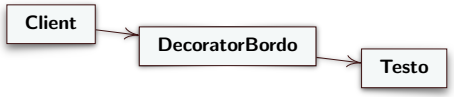
Decorator

- **Intento:** Aggiungere ulteriori responsabilità ad un oggetto dinamicamente. I decorator forniscono un'alternativa flessibile all'implementazione di sottoclassi per estendere funzionalità
- **Motivazione**
  - Alcune volte si vogliono aggiungere responsabilità a singoli oggetti, e non all'intera classe. Per aggiungere responsabilità si potrebbe far uso dell'ereditarietà, tuttavia non si avrebbe flessibilità: un client non può controllare come e quando *decorare* un componente
  - Un approccio più flessibile è racchiudere (wrap) il componente in un altro oggetto che aggiunge una responsabilità. L'oggetto *wrapper* è un decorator. Il decorator è conforme all'interfaccia che decora, quindi la sua presenza è trasparente agli oggetti che usano il componente
  - Le responsabilità possono essere sottratte dinamicamente. I decorator possono essere annidati ricorsivamente, per aggiungere più responsabilità
  - A volte la creazione di sottoclassi non è praticabile. Un numero grande di estensioni produrrebbe un numero enorme di sottoclassi per gestire tutte le combinazioni

Decorator

- Esempio
  - Si vuol aggiungere la proprietà `Bordo` ad un componente `Testo`
  - Se si eredita `Testo` dalla classe `Bordo`, gli oggetti della sottoclasse avranno questa proprietà, ma non si avrà flessibilità: non si possono avere oggetti senza tale proprietà
  - Alternativa flessibile, inserire il componente `Testo` dentro un oggetto che aggiunge `Bordo`
  - L'oggetto che racchiude, chiamato `DecoratorBordo`, manda la richiesta al componente `Testo` e aggiunge altre attività prima o dopo l'invio della richiesta

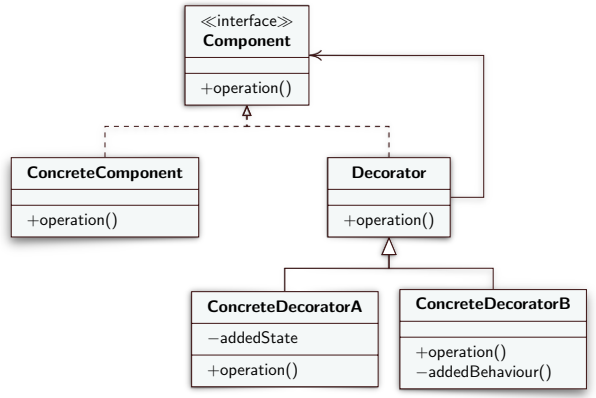


Decorator

- Soluzione
  - **Component** definisce l'interfaccia per gli oggetti che possono avere aggiunte le responsabilità dinamicamente
  - **ConcreteComponent** definisce un oggetto su cui poter aggiungere responsabilità
  - **Decorator** mantiene un riferimento a un oggetto Component e definisce un'interfaccia conforme a quella di Component. Decorator inoltra le richieste al suo oggetto Component e può fare altre operazioni prima e dopo l'inoltro della richiesta
  - **ConcreteDecorator** implementa la responsabilità aggiunta al Component

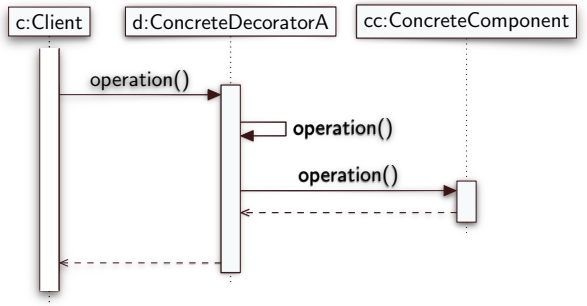
Decorator

- Struttura



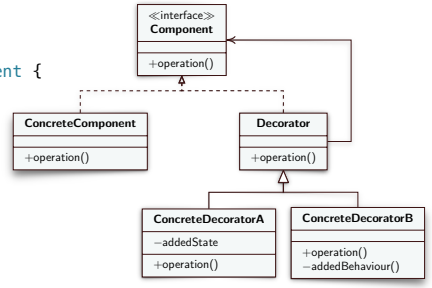
Decorator

- Interazioni



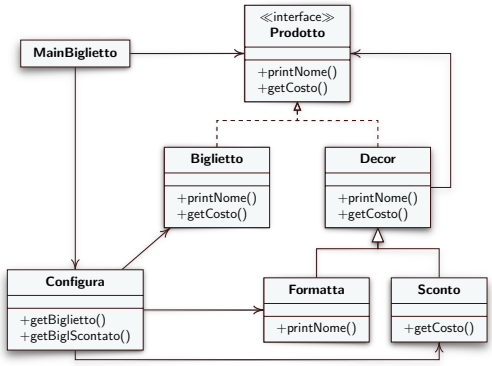
Implementazione

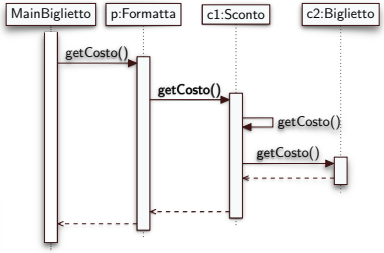
```
interface Component {
    public void operation();
}
class ConcreteComponent implements Component {
    @Override public void operation() {
        // ...
    }
}
class Decorator implements Component {
    private final Component innerC;
    public Decorator(final Component c) {
        innerC = c;
    }
    @Override public void operation() {
        innerC.operation();
    }
}
class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component c) {
        super(c);
    }
    @Override public void operation() {
        super.operation();
        // ...
    }
}

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



Esempio

- Vi sono alcuni prodotti, ad es. `Biglietto`, ognuno con un nome ed un costo; e si hanno diversi modi di calcolare il costo dei prodotti, in base agli sconti e diversi modi di stampare i dettagli
- Si vuol poter combinare a runtime sconti (`Sconto`) e messaggi (`Formatta`) sui prodotti, per singole istanze di `Biglietto`





Conseguenze

- Più flessibilità rispetto all'ereditarietà poiché si possono aggiungere responsabilità dinamicamente
- La stessa responsabilità può essere aggiunta più volte, semplicemente aggiungendo due istanze dello stesso ConcreteDecorator
- Prevedere per le classi in alto nella gerarchia tutte le responsabilità che servono significherebbe avere per esse troppe responsabilità. I ConcreteDecorator sono invece indipendenti e permettono di aggiungere responsabilità successivamente
- L'identità (tipo e riferimento) del ConcreteDecorator non è quella del ConcreteComponent, quindi non si dovrebbero confrontare i riferimenti
- Si avranno tanti piccoli oggetti, che differiscono nel modo in cui sono interconnessi