Model View Controller (MVC)

- E’ considerato un pattern architetturale per le applicazioni interattive
    - individua tre componenti: **Model** per funzionalità principali e dati; **View** per mostrare i dati; **Controller** per prendere gli input dell’utente

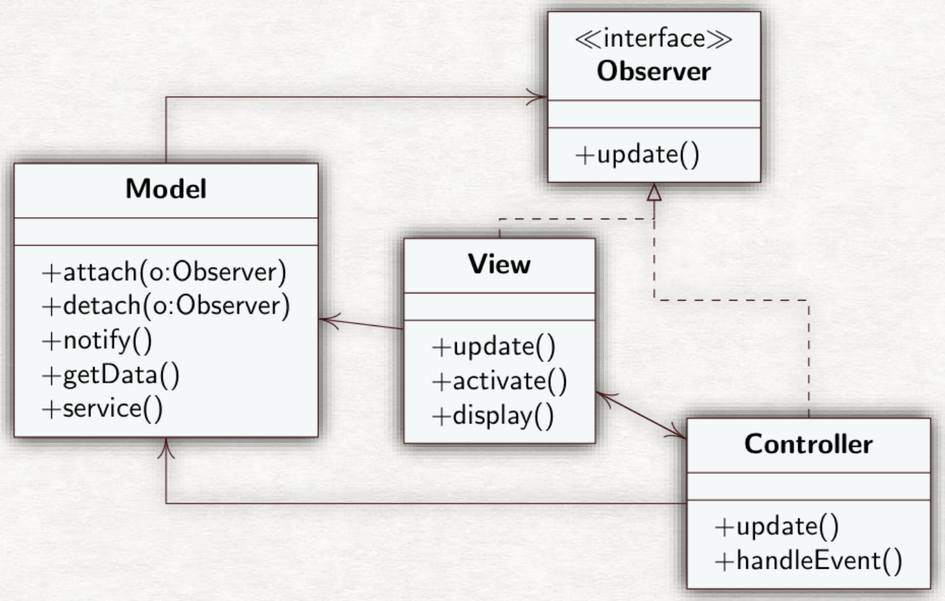
- **Motivazioni**
    - Le interfacce utente possono cambiare, poiché funzionalità, dispositivi o piattaforme cambiano
    - Le stesse informazioni sono presentate in finestre differenti (per es. grafici diversi)
    - Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati
    - I cambiamenti all’interfaccia utente dovrebbero essere facili
    - Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali

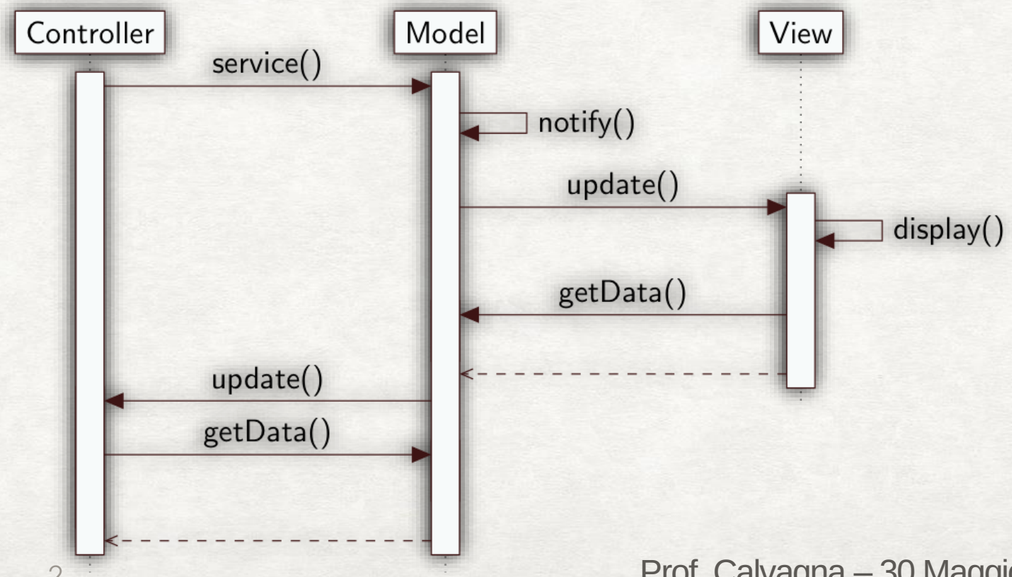
Model View Controller (MVC)

- **Model** incapsula i dati dell'applicazione. E' indipendente dalla rappresentazione degli output e dagli input. E' osservato da View e Controller, cui notifica i cambiamenti di dati

- **View** mostra i dati all'utente. Generalmente ci sono tante View, ogni View è associata a un Controller. View inizializza il proprio Controller, e mostra i dati che legge da Model

- **Controller** riceve gli input dell'utente (da mouse e tastiera) sotto forma di eventi. Traduce gli eventi in richieste di servizio per Model o le passa a View

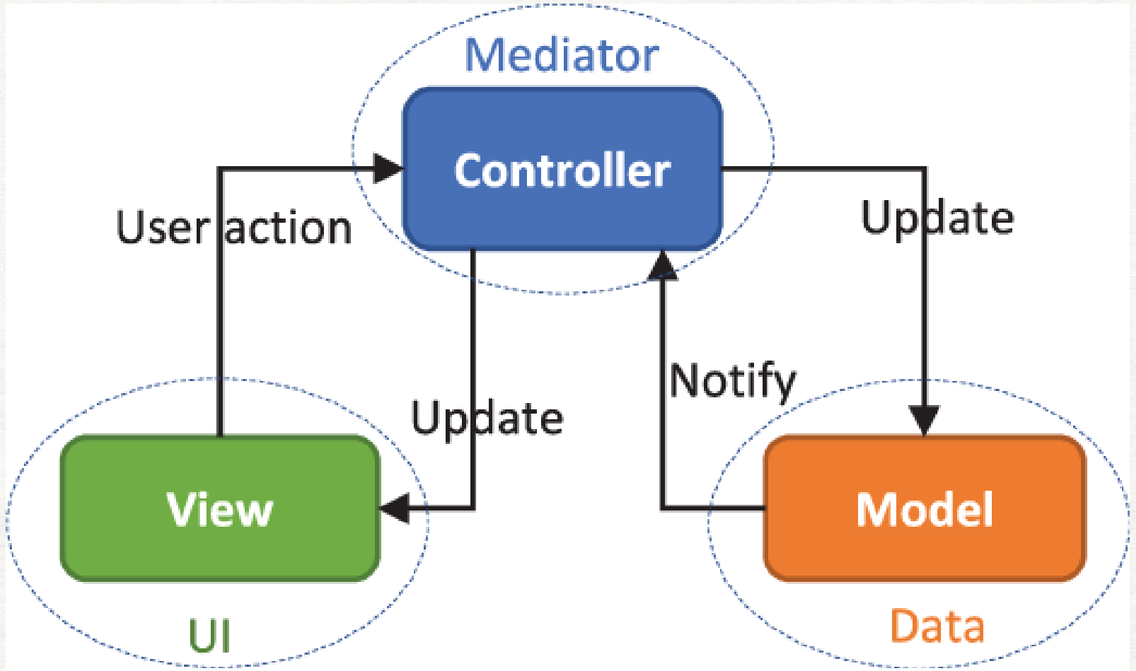


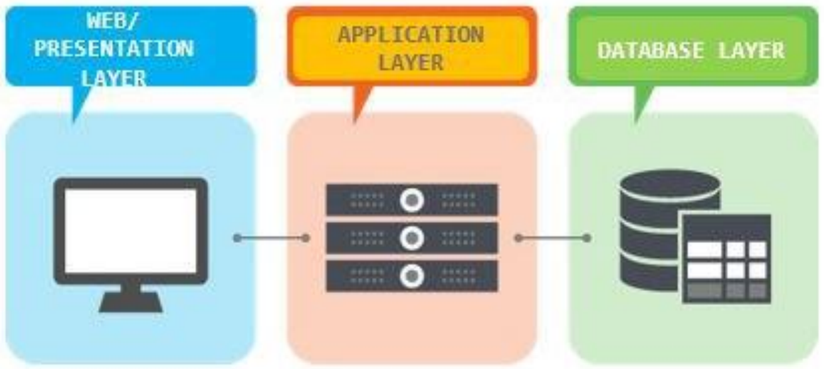


MVC

- Se non consento interazione diretta tra view e model, rappresenta l'equivalente centralizzato dell'architetturale ***three-tier*** per sistemi distribuiti

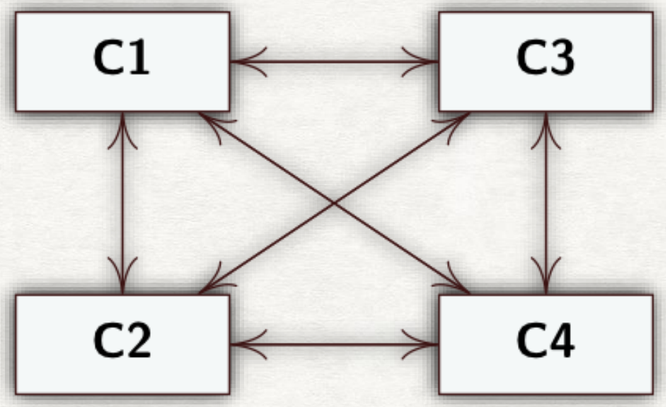
- Vantaggi dovuti alla modularità e disaccoppiamento, ad es. Poter inserire facilmente firewall di protezione tra lato utente (pericoloso) e lato server (intranet)
- Sono applicati il pattern Observer e (in questo caso) anche Mediator





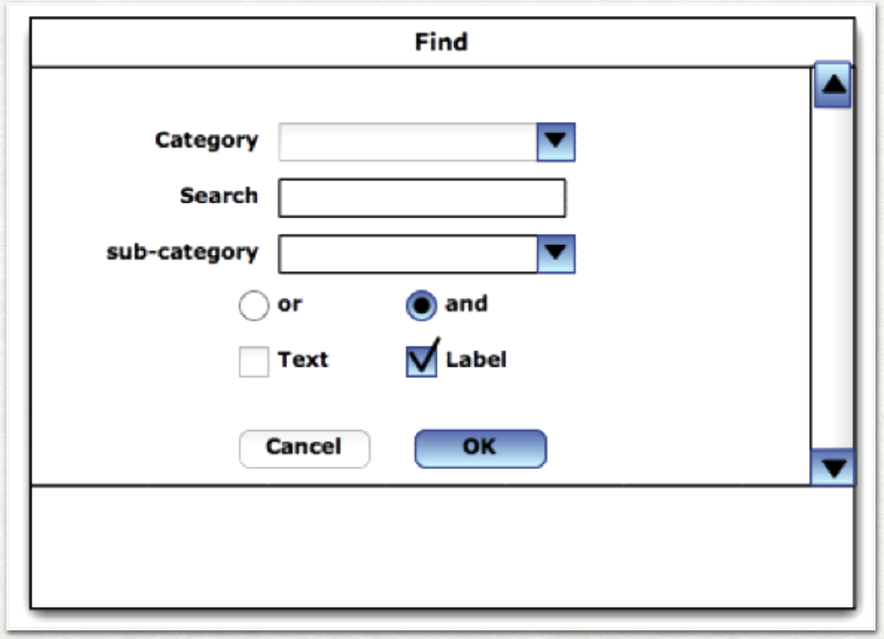
Mediator

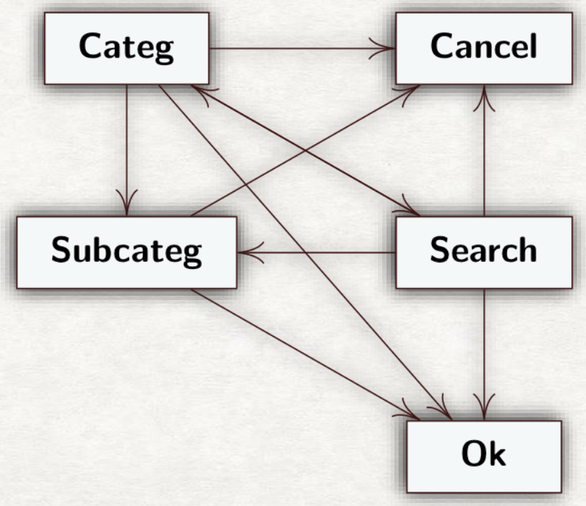
- **Intento**
  - Definisce una interfaccia per la comunicazione tra un gruppo di oggetti che interagisce. Il Mediator promuove il **lasco accoppiamento** fra oggetti poiché evita che essi interagiscano direttamente, e permette di modificare le loro interazioni indipendentemente da essi
- **Motivazione**
  - La distribuzione di responsabilità fra vari oggetti può risultare in molte interazioni fra oggetti (nel caso peggiore un oggetto conosce tutti gli altri) rendendoli **difficilmente riusabili:** l'intero sistema si comporta come se fosse **monolitico**.
  - Diventa difficile **cambiare il comportamento** anche dell'intero sistema poiché è distribuito fra gli oggetti e dovrei **sottoclassarne molti** o tutti.
  - Si possono evitare questi problemi **incapsulando il comportamento collettivo** in un oggetto mediatore separato. Il mediatore serve da **intermediario ed evita che gli oggetti dipendano fra loro**



Mediator

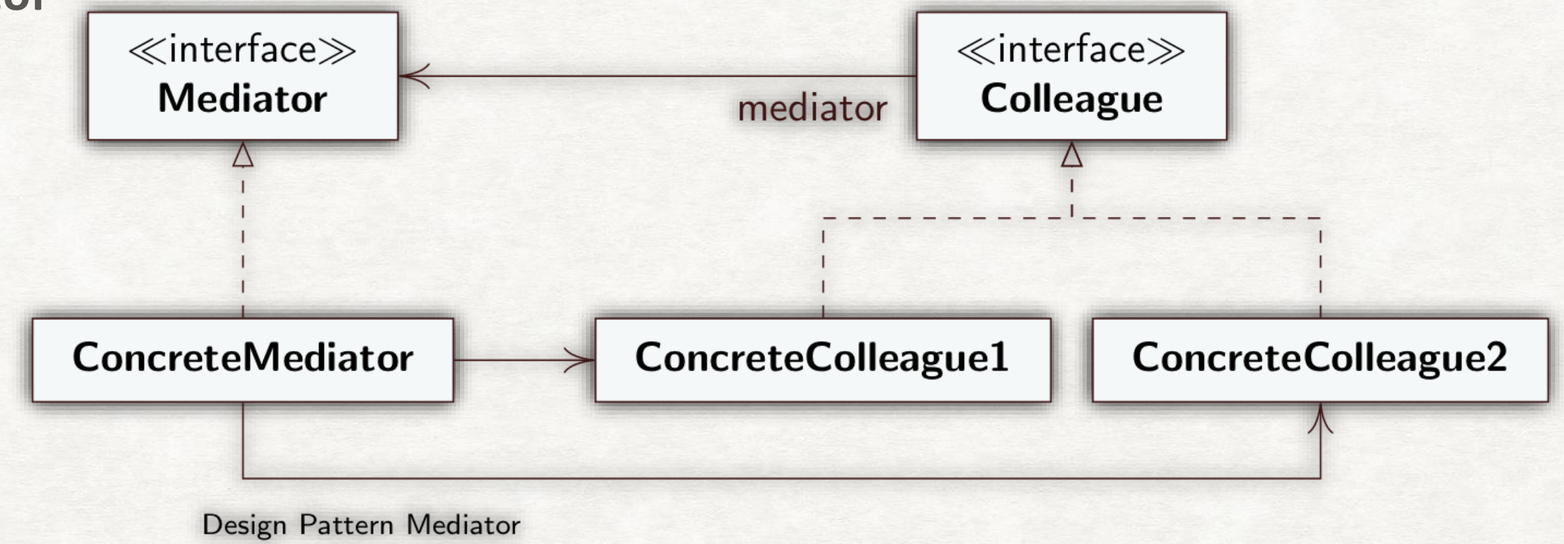
- Per la finestra di ricerca mostrata
  - Ogni elemento visualizzato (testo, lista, bottone, etc.) è controllato da una corrispondente classe
  - Ciascuna classe deve comunicare il suo stato alle altre per far aggiornare la visualizzazione
  - Ciascuna classe (senza un Mediator) chiamerà i metodi di tutte le altre classi, e quindi ciascuna classe è dipendente dalle altre





Mediator

- Soluzione
  - **Isolare le comunicazioni** (complesse) tra oggetti dipendenti (cooperanti) in un sottosistema, creando una classe dedicata **Mediator**



Design Pattern Mediator

  - **ConcreteMediator** implementa il comportamento cooperativo e coordina gli oggetti **Colleague**
  - Ogni Colleague conosce solo il Mediator, e **comunica solo con il Mediator** quando avrebbe comunicato con un altro Colleague
  - I **ConcreteColleague** mandano richieste, e ricevono richieste, da/a un oggetto Mediator. Il Mediator implementa il comportamento cooperativo inoltrando le richieste a opportuni ConcreteColleague
  - l'astrazione **Colleague** serve a consentire di imporre il requisito di un mediator (condiviso e rimpiazzabile) ai concreteColleague

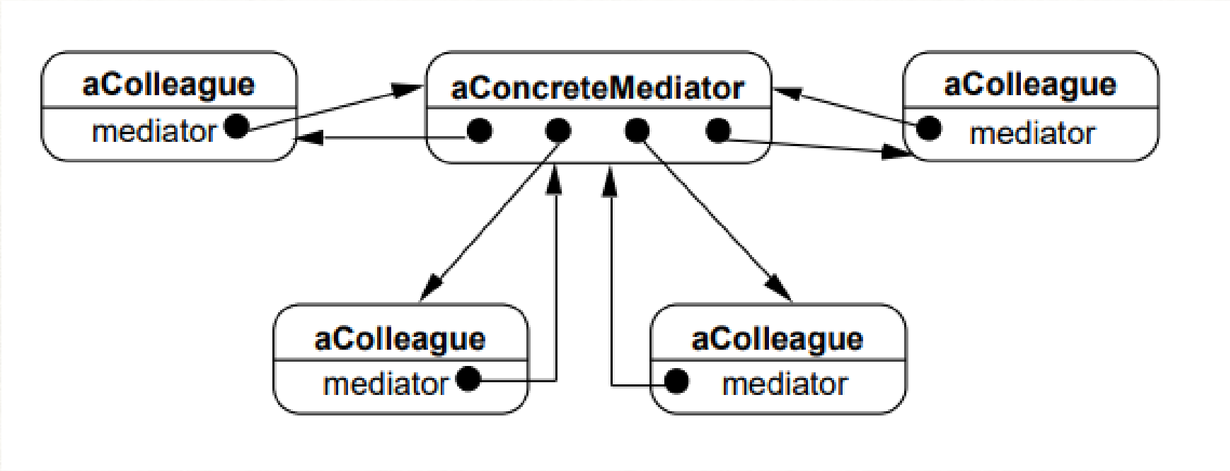
Primo esempio

- Vediamo un programmino prima e dopo l'applicazione del pattern
  - COOLING SYSTEM
    - a simple cooling system consists of a fan, a power supply, and a button. Pressing the button will either turn on or turn off the fan. Before we turn the fan on, we need to turn on the power. Similarly, we have to turn off the power right after the fan is turned off.

- Tre component interdipendenti: Fan, PowerSupply e Button.
- Codice nei folder mediator/nomediator

Mediator

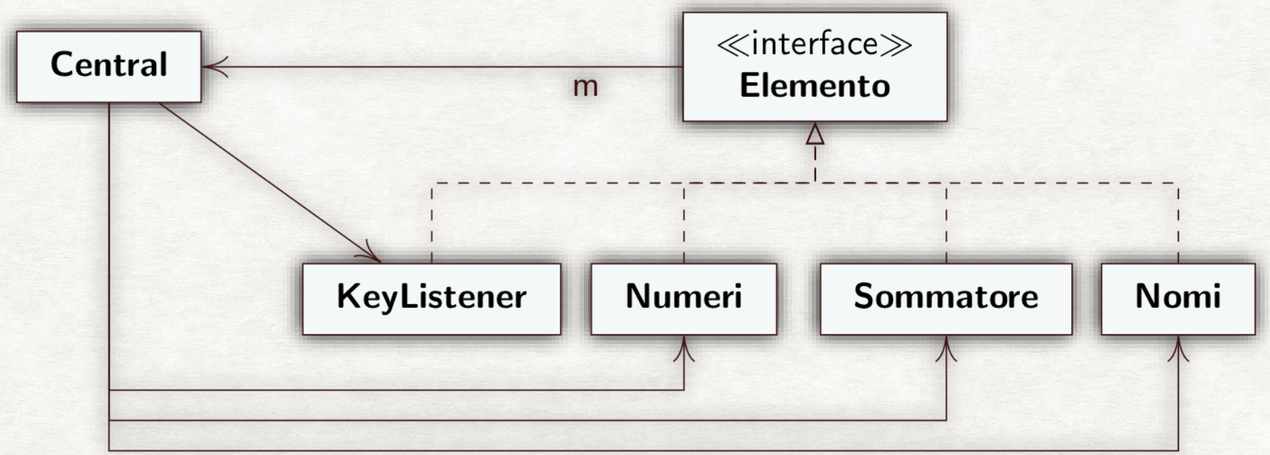
- Struttura a run-time



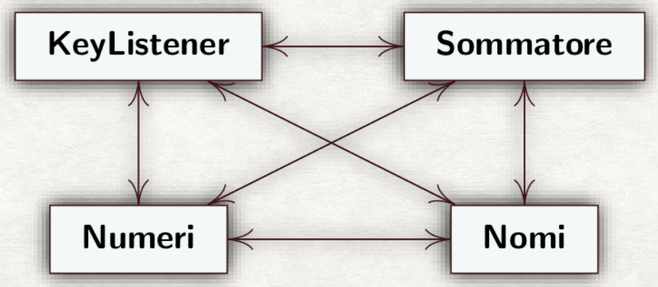
- L'oggetto mediator è responsabile per la gestione (istanzia e mantiene i riferimenti) ed il coordinamento dell'interazione tra i colleghi: invoca i loro servizi su loro richiesta e/o iniziativa propria
- Non è una semplice facciata (facade) : è un direttore che orchestra i suoi musicisti
- Façade non aggiunge funzionalità; mediator ha al suo interno tutta la logica di funzionamento che era distribuita e può sensatamente averne anche di nuova
- I colleghi sono consapevoli del mediator, con cui interagiscono; i componenti del sottosistema sono inconsapevoli dell'esistenza del Façade. Per i client non cambia nulla in entrambi i casi

Altro Esempio: Gioco

- Si legge un dato dalla tastiera e si compiono delle operazioni, su numeri o su stringa in base al dato letto
- La classe KeyListener legge da tastiera un dato e ritorna il valore letto a Central (un Mediator), quest'ultima chiama i metodi dei ConcreteCollegue Numeri, Sommatore, Nomi, in base al tipo di dato letto



- Nel caso non si adottasse il Mediator, le varie classi si chiamerebbero fra loro

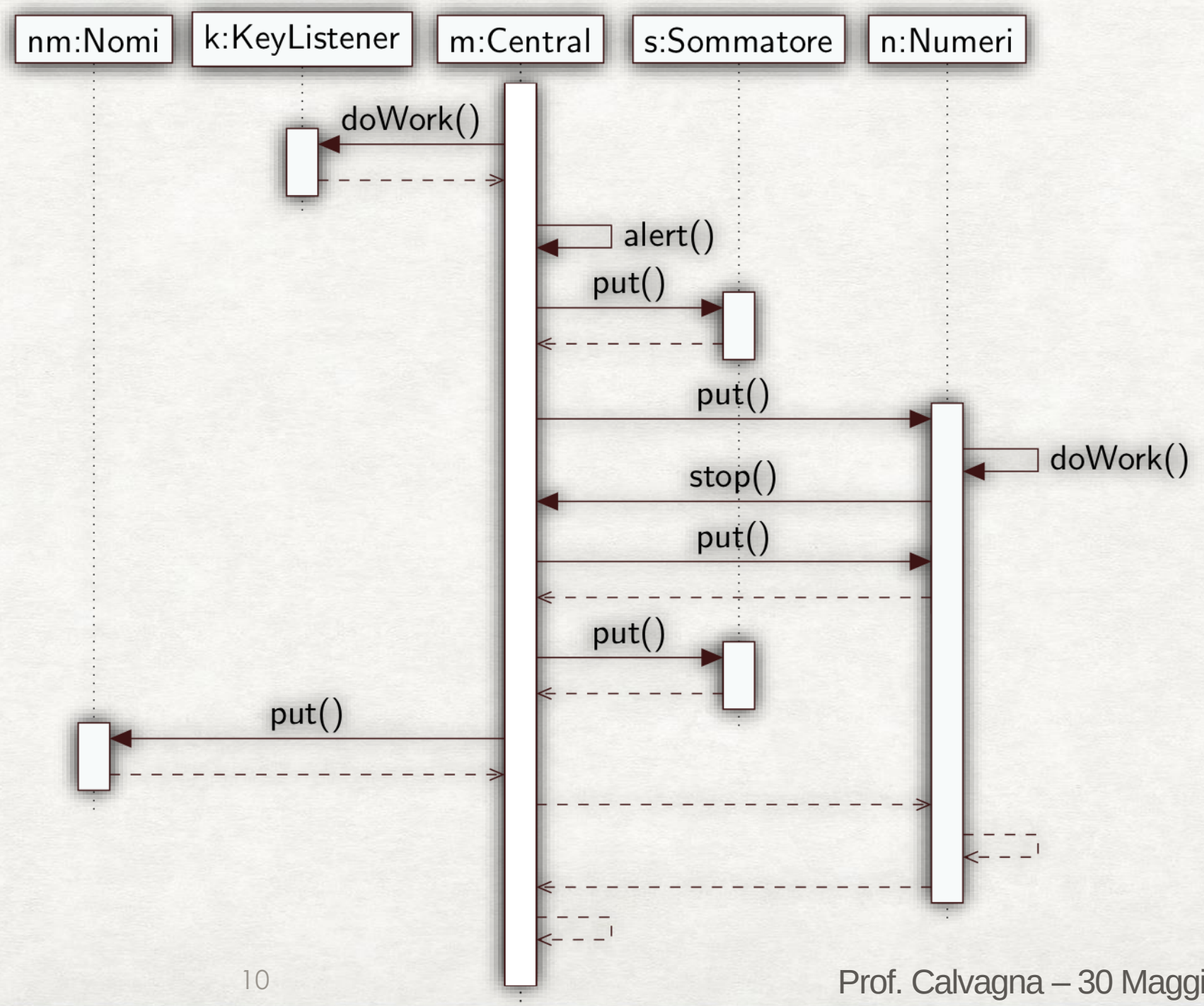


Altro Esempio

- Il Mediator Central avvia la lettura da tastiera tramite il metodo doWork() di KeyListener e ottiene da esso il valore letto, quindi Central chiama put() sugli oggetti interessati al valore letto

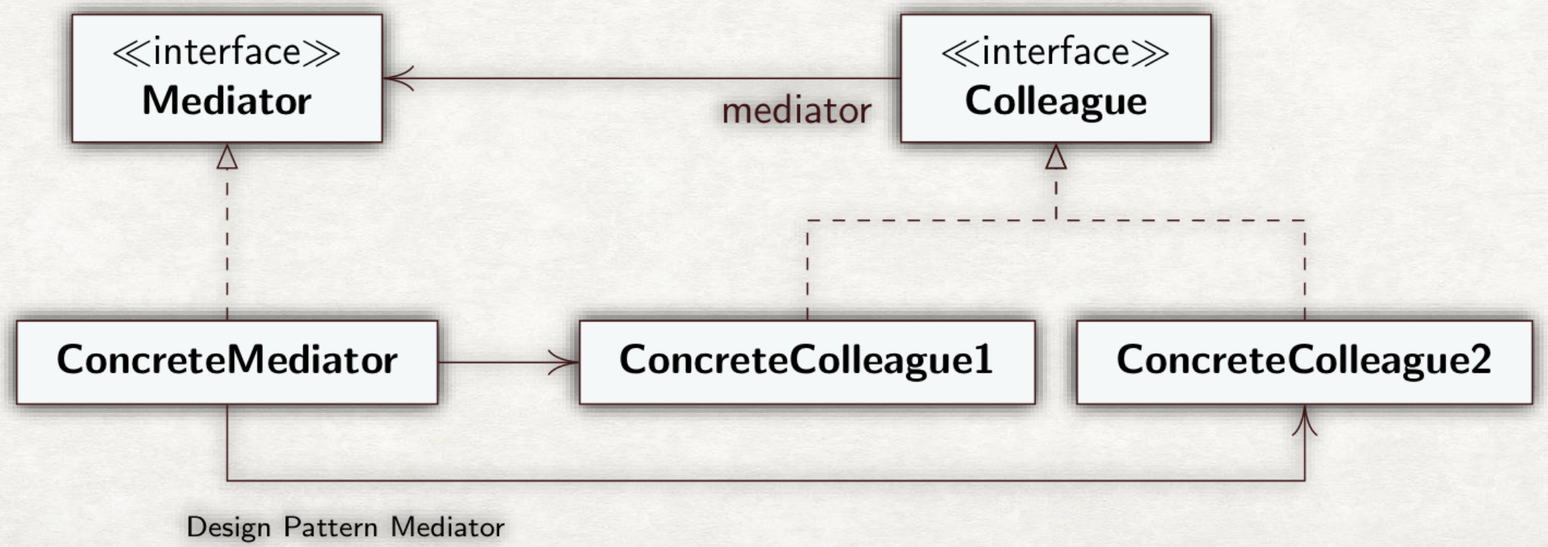
- Quando un oggetto ConcreteColleague riconosce una condizione di arresto, chiama stop() su Central, che avvisa gli altri ConcreteColleague

- In figura si mostra il caso in cui Numeri chiama stop() su Central



Mediator

- **Applicabilità**
  - Usare il Mediator quando
    - Un insieme di oggetti comunicano in modo ben definito ma **intricato**. Le interdipendenze che ne risultano sono non strutturate e difficili da comprendere
    - Riusare un oggetto è difficile poiché esso comunica con tanti altri oggetti
    - Un comportamento che è distribuito fra tante classi dovrebbe essere modificabile senza dover ricorrere a sottoclassi: lo centralizzo nel ConcreteMediator e cambio solo lui.



Design Pattern Mediator

Interazione

- Il con. mediator accede alle interfacce di tutti i concrete colleagues direttamente
- I colleagues accedono al mediator:
  - Mediator con metodi espliciti: corrispondenti a quelli nei colleague
  - Mediator con metodo unico generico: eventualmente passo il colleague e lo identifico
  - Metodo generico in overloading: uno per ogni tipo di concrete colleague
  -
- Negli esempi mostrati…
  - Cooling: il mediator riporta esplicitamente metodi dei colleagues che ora sono invocati dai colleagues su di lui (metodi espliciti).
  - Gioco: il mediator orchestra i colleagues, uniformati ad «Elementi». Interazione col mediator (metodo stop(), unico per tutti).

Metodo generico

```
class DialogDirector {
    private Button button;
    private Fan fan;
    private PowerSupply powerSupply;

    // altro codice

    public void doWork( Colleague me) {
        if ( me == button ) blah
        else if ( me == fan) more blah
        else if ( me == powerSupply ) even more blah
    }
}
```

Metodo generico in overload

```
class DialogDirector {
    private Button button;
    private Fan fan;
    private PowerSupply powerSupply;
   private Button anotherButton;


    // altro codice

    public void doWork(Button b) {
        if ( b == button ) // Process button task
        else if ( b == otherButton) // Process other button task
    }
    public void doWork(Fan f) {
        // Process fan task
    }
    public void doWork(PowerSupply b) {
        // Process button task
    }

}
```

Mediator

- **Conseguenze**
- La maggior parte della complessità che risulta nella gestione delle dipendenze è spostata dagli oggetti cooperanti al Mediator. Questo rende gli oggetti più facili da implementare e mantenere
- Le classi Colleague sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con il codice che gestisce le dipendenze
- Il codice del Mediator non è in genere riusabile poiché la gestione delle dipendenze implementata è solo per una specifica applicazione: può essere direttamente una classe concreta
- Mediator è una astrazione proprio per consentire eventualmente di cambiarne implementazione a parità di colleagues

PAUSA 10 min

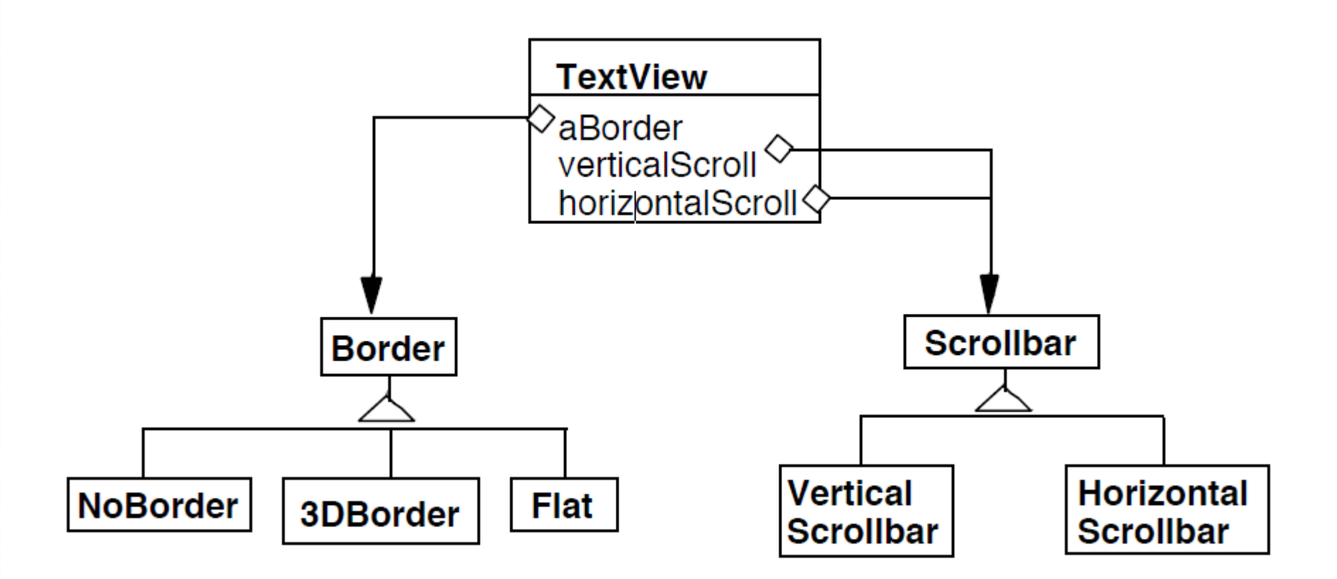
Decorator

- Esempio: pensiamo ad un oggetto grafico TextView che può avere anche:
  - Barra scorrimento verticale
  - Barra scorrimento orizzontale
  - Bordo 3D
  - Bordo piatto
- Combinandole, ci sono una dozzina di varianti della TextView base:
- TextView
- TextViewWithNoBorder&SideScrollbar
- TextViewWithNoBorder&BottomScrollbar
- TextViewWithNoBorder&Bottom&SideScrollbar
- TextViewWith3DBorder
- TextViewWith3DBorder&SideScrollbar
- Etc.

- Non è una buona idea implementare il codice derivando altrettante sottoclassi
- qual è l'alternativa?

Soluzione 1

- Potrei usare un misto di composizione e derivazione
- Ma TextView dipenderebbe da tutte le classi radice di derivazione
- Aggiungere una nuova tipologia di varianti significa dover modificare TextView
- Dovrei modificare anche ogni altro tipo di view a cui la nuova variante è applicabile

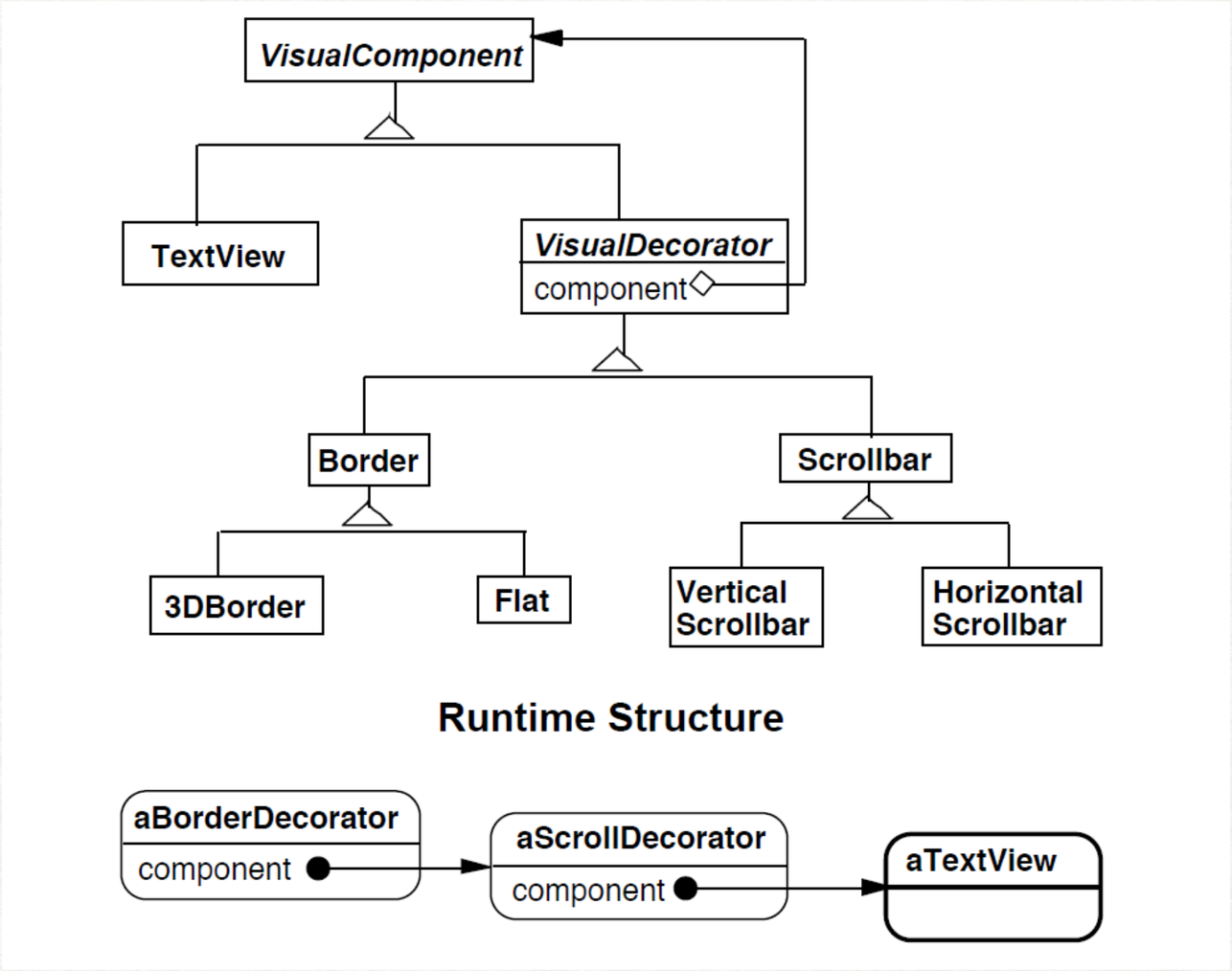



```
class TextView {
      Border myBorder;
      ScrollBar verticalBar;
      ScrollBar horizontalBar;

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

Soluzione 2

- Uso il solo la composizione di oggetti
- TextView non ha nè bordo ne barra
- Aggiungo bordo e barra sopra TextView
- TextView è il **componente** base
- Gli altri sono aggiunte che lo migliorano: **decorazioni**
- **Gli oggetti sono contentuti l'uno dentro l'altro**
- **Ho una catena di oggetti, cui si accede dall'ultima decorazione aggiunta**



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

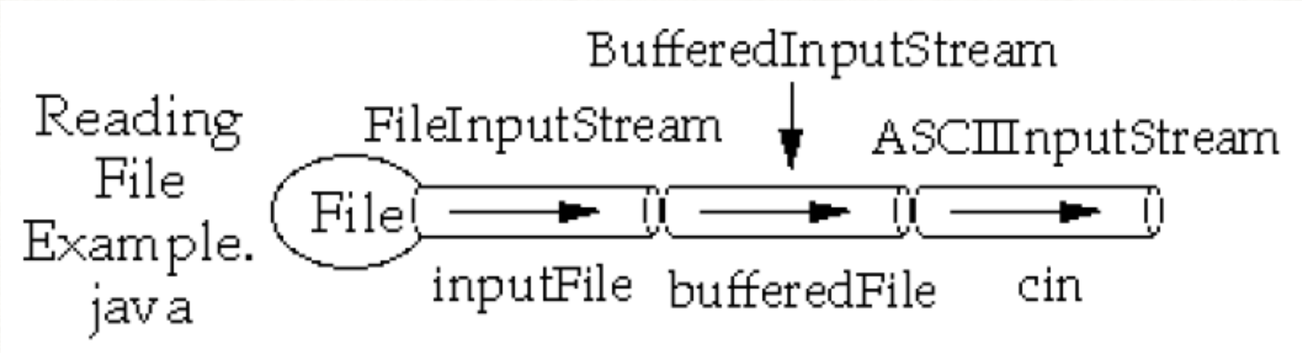
  - Se invece si eredita Testo nella classe Bordo non si avrà flessibilità: non si possono avere bordi se non attorno a testi
  - Il client ottiene un testo incorniciato, interagendo solo con l'oggetto più esterno
  - DecoratorBordo usa i servizi della componente Testo e aggiunge la sua attività prima o dopo l'invio della richiesta ad esso

Decorator

- **Intento**: Aggiungere ulteriori responsabilità ad un oggetto dinamicamente.
    - Alternativa flessibile all'uso di sottoclassi per estendere funzionalità
    - Avvolgo (wrap) il componente in un altro oggetto che aggiunge una (sola) responsabilità. L'oggetto wrapper è un decorator e la sua presenza è trasparente ai client
    - Posso aggiungere a catena moltissime funzionalità
- **Motivazione**
    - Poter aggiungere combinazioni diverse di responsabilità a singoli oggetti, e non all'intera classe.
    - Le responsabilità devono poter essere anche sottratte dinamicamente.
    - Scelta forzata se la creazione di sottoclassi non è praticabile per il numero elevato di possibili estensioni
    - usato in java i/o streams, per la gestione dei Fonts, per le interfacce utente

Java I/O Streams

- Esempio di applicazione del pattern

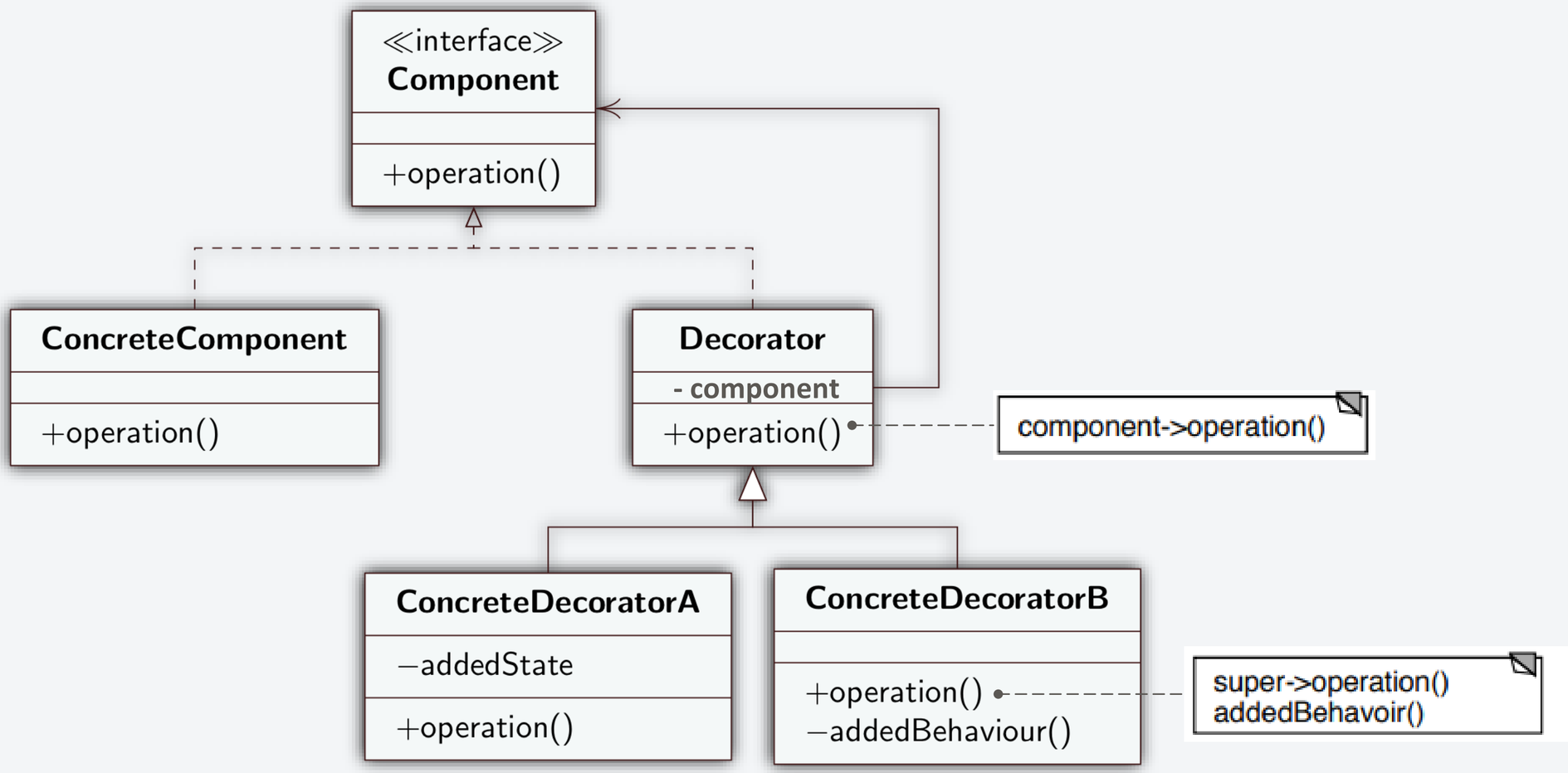


Vediamo piccolo esempio di codice…ReadingFileExample

- Affini al pattern per un verso (pipelining), i nuovi streams funzionali in java 8 non sono esattamente una sua applicazione

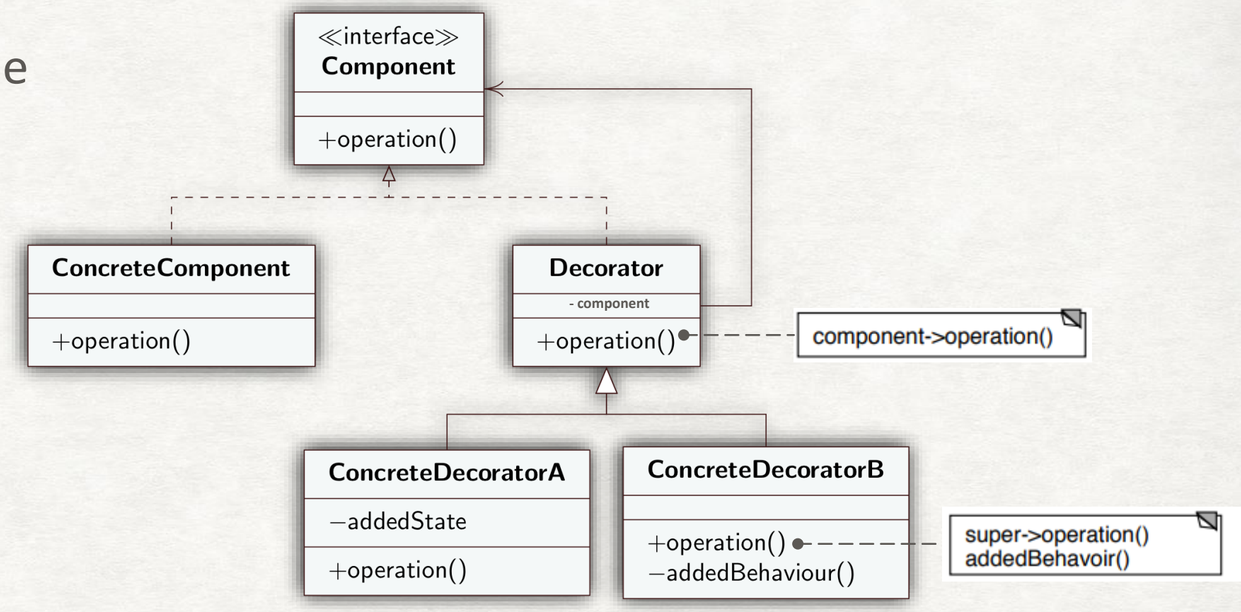
Decorator

- **Struttura**



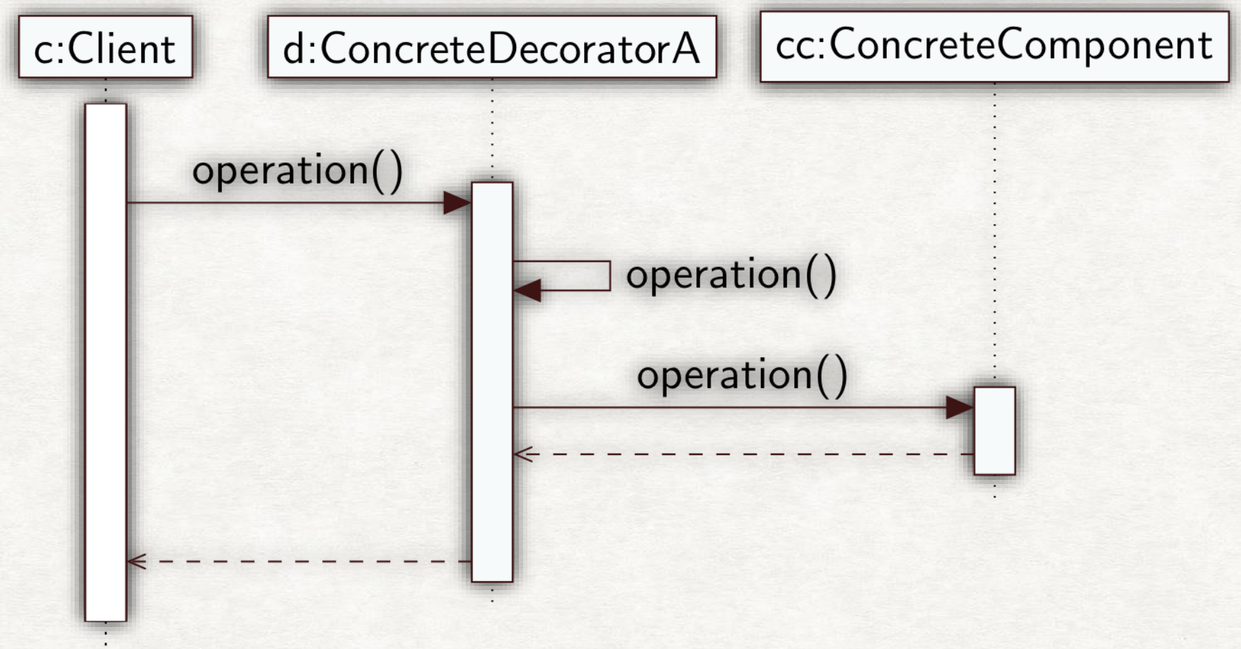
Decorator

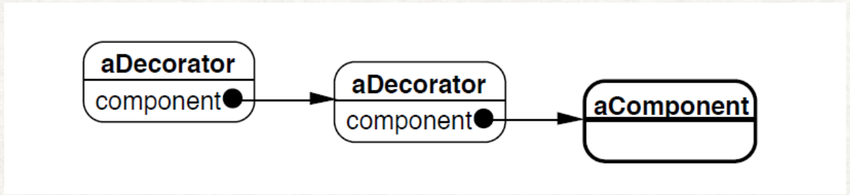
- **Struttura**
  - **Component:** interfaccia comune per gli oggetti che possono avere aggiunte le responsabilità dinamicamente: contiene l'operazione che li lega semanticamente
  - **ConcreteComponent** è l'oggetto base cui poter aggiungere responsabilità
    - Termina la catena di oggetti composti
  - **Decorator** component con dentro un riferimento a un ulteriore Component. Inoltra le richieste di servizio al suo Component interno e svolge la sua operazione prima e/o dopo l'inoltro della richiesta
  - **ConcreteDecorator** implementa una responsabilità aggiuntiva per il Component



Decorator

- Interazioni

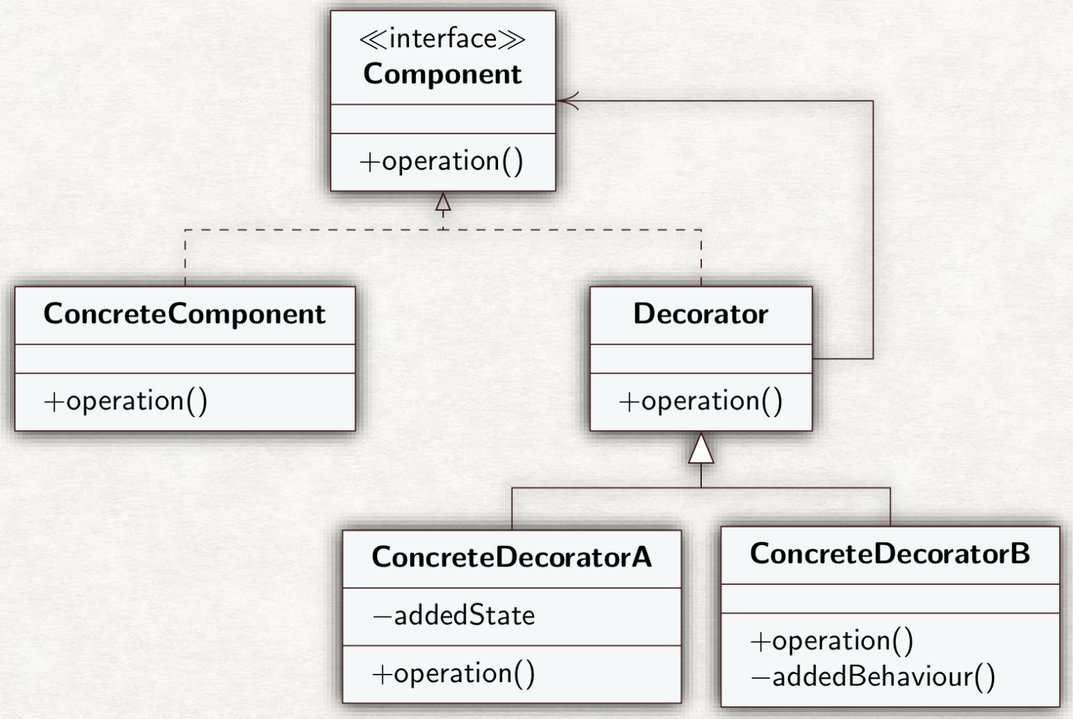




Implementazione

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

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



Conseguenze

- Più **flessibilità** rispetto all'ereditarietà poiché si possono aggiungere responsabilità **dinamicamente**
- La stessa responsabilità può essere aggiunta **più volte**
    - uso più istanze dello stesso ConcreteDecorator
- Si avranno tanti piccoli oggetti, che differiscono nel modo in cui sono interconnessi
- Tante piccole classi, focalizzate su un solo compito molto specifico
- Il ConcreteDecorator ed il ConcreteComponent sono oggetti diversi e di tipo diverso
    - Evitare di affidarsi a check del tipo: if (aComponent instanceof TextView) blah

Conseguenze

- 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
- I decorator possono essere annidati ripetutamente, per aggiungere più responsabilità
- **Change Skin vs change Guts** : modifico a run-time il comportamento di un oggetto, come nel pattern State, ma:
  - L'oggetto con cui interagisco cambia effettivamente, e anche il suo tipo (cambio di pelle)
  - Con State è un cambio interno, non gestibile e non visibile dal client (cambio di viscere)
  - Inoltre aggiungo funzionalità invece che rimpiazzarla, come accadeva nello State

Esempio

- Altro esempio su VSCODE…AppMessaggio

Esercizio per casa

- 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