Metodologie di Programmazione

Lezione 31: Design pattern (parte 3)

Lezione 31: Sommario

UNITELMA SAPIENZA
SAPIENZA UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

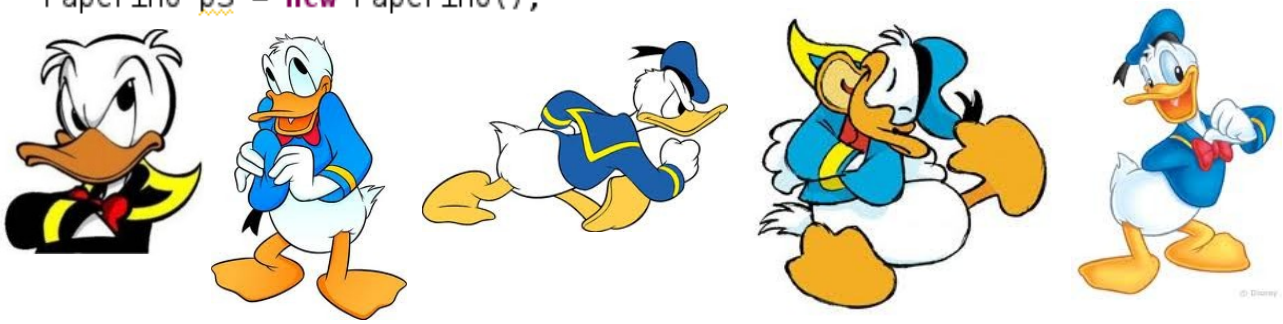
- Singleton pattern
- Decorator pattern
- Command/Callback pattern

Possono esistere classi con un'unica istanza?

- Certamente! Ad esempio: Paperino
- Tuttavia nulla mi impedisce di creare più istanze DISTINTE!

```
public class Paperino extends PersonaggioDisney
{
    public void lucidaMoneteDelloZione()
    {
        System.out.println("swish swish swish...");
    }

    public static void main(String[] args)
    {
        Paperino p1 = new Paperino();
        Paperino p2 = new Paperino();
        Paperino p3 = new Paperino();

    }
}
```

Ma questo non ha senso!



Il Singleton Pattern: forzare un unico punto di accesso


```
public class Paperino extends PersonaggioDisney
{
    static private Paperino istanza;

    static public Paperino getInstance()
    {
        if (istanza == null) istanza = new Paperino();
        return istanza;
    }

    private Paperino()
    {
        // costruisci l'unico oggetto
    }

    public static void main(String[] args)
    {
        Paperino p = Paperino.getInstance();
        Paperino p2 = Paperino.getInstance();
        assert(p == p2);
    }
}
```

Unica istanza della classe (campo statico!)

Punto di accesso per la costruzione

Costruttore privato: nessuno può chiamarlo, tranne metodi (statici) della classe stessa

Paperino vs. Paperinik

- Ricordate l'esercizio Disney vs. Marvel?
- Bisognava fare in modo che i personaggi avessero un'identità **pubblica** e un'identità **privata**
- Ma come fare in modo di non creare un'istanza per Paperino e un'altra istanza per Paperinik? O addirittura istanze multiple di Paperino o Paperinik?

Paperino vs. Paperinik

- Con il pattern Singleton!

Ho un'unica istanza: di Paperinik (che è anche Paperino)

Paperino implementa il singoletto

Paperinik ha un costruttore privato, quindi inaccessibile all'esterno

```java
/**
 * Paperino e Paperinik sono un'unica entità:
 * Questo è possibile solo con il Singleton Pattern!
 *
 * @author navigli
 */
public class Paperino extends Personaggio implements DoppiaVita
{
    /**
     * Identita' segreta
     */
    static private Paperinik paperinik;

    static public Paperino getInstance()
    {
        if (paperinik == null) paperinik = new Paperinik();
        return paperinik;
    }

    private Paperino() { }

    @Override
    public Supereroe assumiIdentitaSegreta() { return paperinik; }

    @Override
    public Personaggio assumiIdentitaPubblica() { return this; }

    /**
     * Implementa Paperinik
     * @author navigli
     *
     */
    static private class Paperinik extends Paperino implements Supereroe
    {
        private Paperinik() { }

        @Override
        public Supereroe assumiIdentitaSegreta() { return this; }

        @Override
        public Personaggio assumiIdentitaPubblica() { return this; }

        @Override
        public void attacca(Supereroe s)
        {
            System.out.println("Potere degli stivaletti a molla!");
        }
    }
}
```

Che principio abbiamo utilizzato?

Principio di design:
Fai in modo che la classe modelli la realtà e **non** permetta cose concettualmente insensate

- Il Singleton Pattern permette di obbligare/vincolare la costruzione di un unico oggetto mediante un unico punto di accesso statico e un costruttore privato

Come associare diverse implementazioni a un'astrazione

- Supponiamo di modellare un oggetto Automobile mediante due classi astratte:

```
abstract public class Automobile
{
    abstract public void guida();
}
```

```
abstract public class AutomobileConTarga extends Automobile
{
    private String targa;
    private Color colore;

    public AutomobileConTarga(String targa, Color colore)
    {
        this.targa = targa;
        this.colore = colore;
    }

    public String getTarga() { return targa; }
    public Color getColore() { return colore; }
}
```

Come associare diverse implementazioni a un'astrazione

Abbiamo due tipi concreti di automobili

```
public class Ferrari extends AutomobileConTarga
{
    public Ferrari(String targa, Color colore)
    {
        super(targa, colore);
    }

    public void guida() { System.out.println("Vrroooooooom!!!"); }
}
```

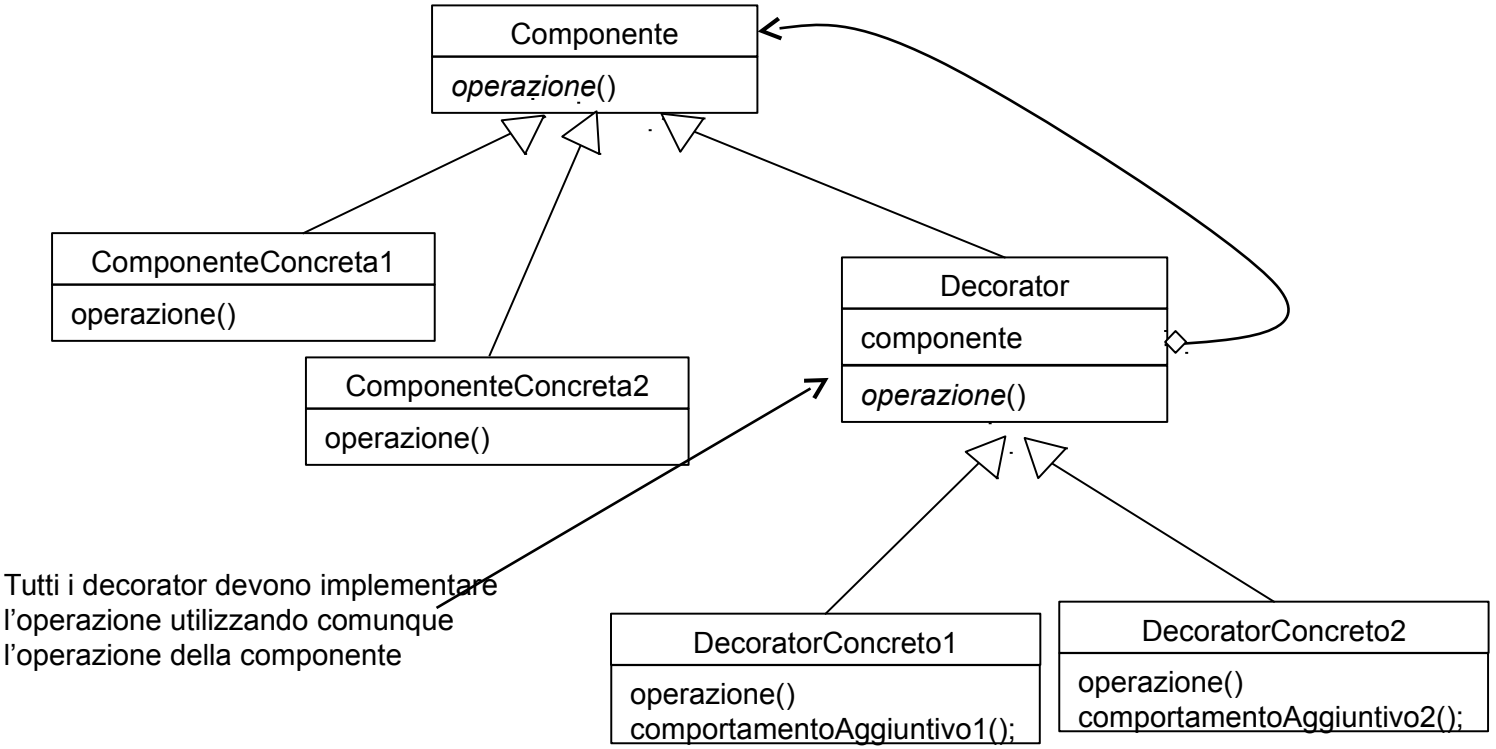
```
public class Cinquecento extends AutomobileConTarga
{
    public Cinquecento(String targa, Color colore)
    {
        super(targa, colore);
    }

    public void guida() { System.out.println("Po po po po po po po po..."); }
}
```

- Supponiamo di voler associare una rappresentazione grafica alle automobili

Il Decorator Pattern

- E' possibile aggiungere nuove responsabilità a un oggetto **senza che esso lo sappia**
- Il Decorator:
  - Estende la classe astratta dell'oggetto
  - E' costruito con un'istanza concreta della classe astratta dell'oggetto
  - Inoltra le richieste di tutti i comportamenti all'oggetto (componente)
  - Effettua azioni aggiuntive (ad esempio, il disegno 2d o 3d)

Il Decorator Pattern: The Big Picture



Il Decorator Pattern per rappresentare graficamente le automobili


```
abstract public class DecoratorAutomobile extends Automobile
{
    protected Automobile automobile;

    public DecoratorAutomobile(Automobile automobile)
    {
        this.automobile = automobile;
    }

    abstract public void guida();
}
```

Componente da decorare

Decoratore costruito con la componente da decorare

Rendiamo astratto il comportamento da “decorare”

Il Decorator Pattern per rappresentare graficamente le automobili


```
public class DecoratorAutomobile2D extends DecoratorAutomobile
{
    private Immagine2D immagine;

    public DecoratorAutomobile2D(Automobile automobile, Immagine2D immagine)
    {
        super(automobile);
        this.immagine = immagine;
    }

    public void guida()
    {
        automobile.guida();
        immagine.setX(immagine.getX()+1);
        immagine.disegna();
    }
}
```

Incapsula la rappresentazione 2D dell'automobile

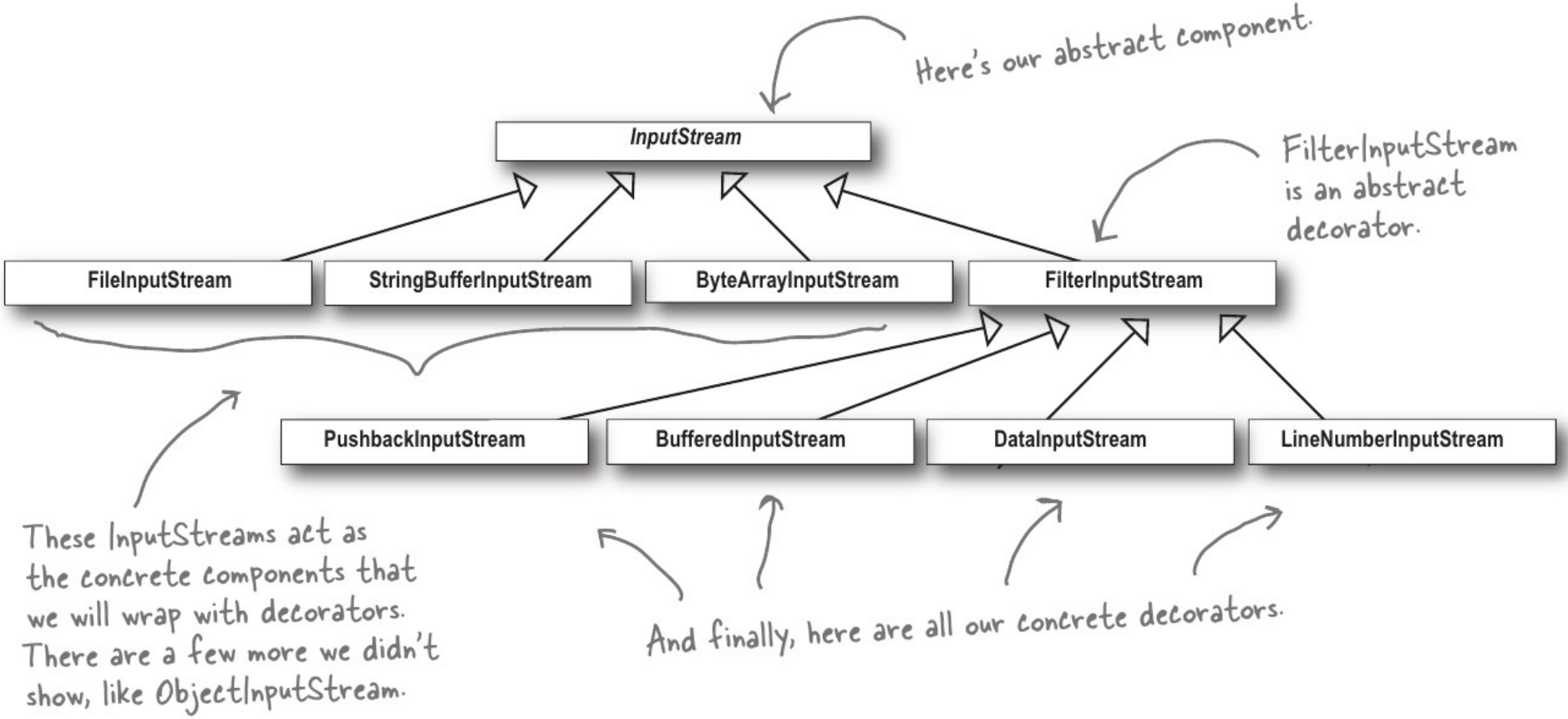
Incapsula la rappresentazione 3D dell'automobile

```
public class DecoratorAutomobile3D extends DecoratorAutomobile
{
    private Immagine3D immagine;

    public DecoratorAutomobile3D(Automobile automobile, Immagine3D immagine)
    {
        super(automobile);
        this.immagine = immagine;
    }

    @Override
    public void guida()
    {
        automobile.guida();
        immagine.setX(immagine.getX()+1);
        immagine.setZ(immagine.getZ()+3);
        immagine.disegna();
    }
}
```

Comportamento di base dell'oggetto decorato

Comportamento aggiuntivo del decorator

I Decorator nelle API I/O di Java



Che principio abbiamo utilizzato?

Principio di design:
Le classi dovrebbero essere aperte all'estensione ma chiuse alla modifica
(Open-Closed Principle)

- Il Decorator Pattern permette di estendere le funzionalità/responsabilità di una classe, senza modificare il codice della stessa
- E' un'alternativa all'estensione mediante subclassing che permette la "composizione" a piacimento

Command/Callback pattern

- A volte è necessario effettuare richieste a oggetti **senza sapere nulla relativamente all'operazione richiesta**
- L'operazione potrà eseguita **in futuro**, quando necessario
- Per fare ciò, è necessario rendere l'operazione modulare in modo che possa essere associata a un oggetto
- E in seguito diverse associazioni possano essere fatte dinamicamente

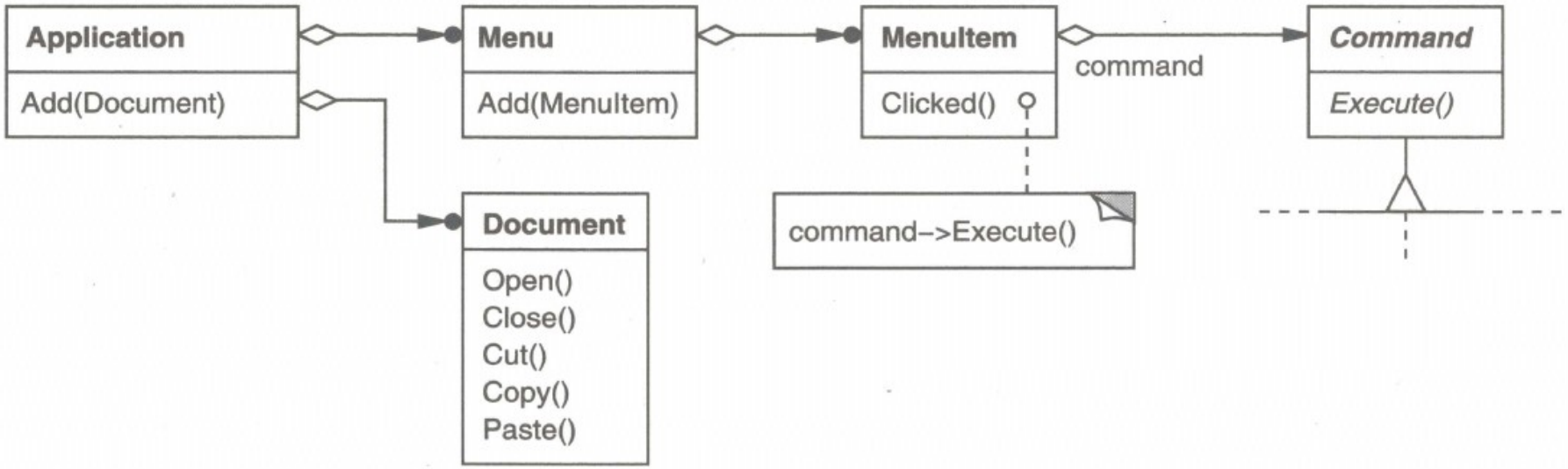
Come fare?

- Si crea un'interfaccia che espone il metodo generale:

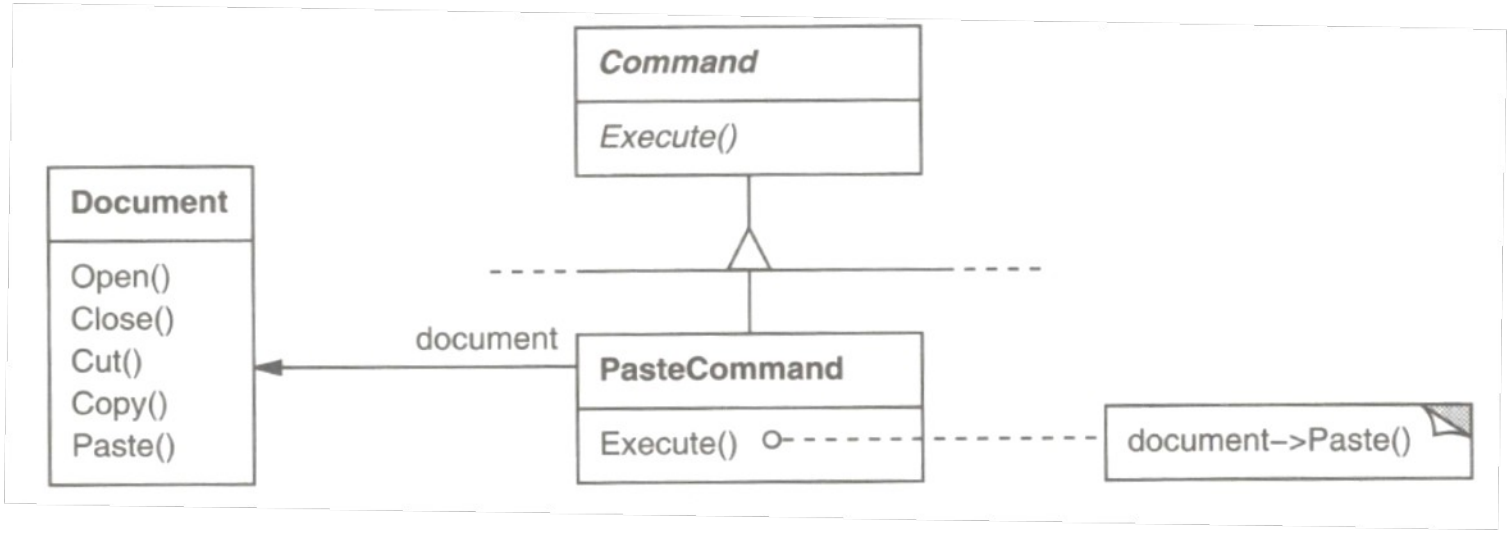
```
public interface Callback
{
    public void execute();
}
```

- Ogni funzione concreta implementa l'interfaccia **Callback**

Un esempio: associare delle operazioni a un menù



Un esempio: associare delle operazioni a un menù



Quali pattern abbiamo visto?

- **Behavioral pattern (relativi al comportamento):**
  - Strategy Pattern
    - Per rendere flessibile, dinamica ed estensibile la gestione dei comportamenti
  - Observer/Observable
    - Per comunicare uno stato (o evento) a oggetti interessati al suo aggiornamento (o accadimento)
  - Command
    - Per codificare funzioni da salvare in/passare a un oggetto e chiamare in seguito
  - Iterator (!!!)
    - Fornisce un modo per accedere a una collezione in modo sequenziale senza esporne la rappresentazione sottostante

Quali pattern abbiamo visto?

- **Creational pattern (relativi alla creazione di oggetti):**
  - SimpleFactory e Factory
    - Per separare la creazione degli oggetti dal resto della logica
  - Singleton
    - Per impedire la creazione di più di un'istanza di una classe
- **Structural pattern (relativi all'organizzazione delle classi):**
  - Decorator
    - Per fornire l'implementazione grafica di un oggetto

Per approfondire...