

# 13 . Design Pattern GoF

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2023/24

Università degli Studi di Torino - Dipartimento di Informatica

## Attenzione!



©2024 Copyright for this slides by Matteo Baldoni. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).


<https://creativecommons.org/licenses/by/4.0/>

## Si noti che

questi lucidi sono basati sul libro di testo del corso "C. Larman, *Applicare UML e i Pattern*, Pearson, 2016" e sul documento "F. Guidi Polanco, *GoF's Design Patterns in Java*, Politecnico di Torino, 2002, disponibile all'indirizzo:

<http://eii.pucv.cl/pers/guidi/designpatterns.htm>.

# Table of contents

- 
1. Creazionali: Abstract Factory e Singleton
  2. Strutturali: Adapter, Composite e Decorator
  3. Comportamentali: Observer, State, Strategy e Visitor

# I design pattern GoF

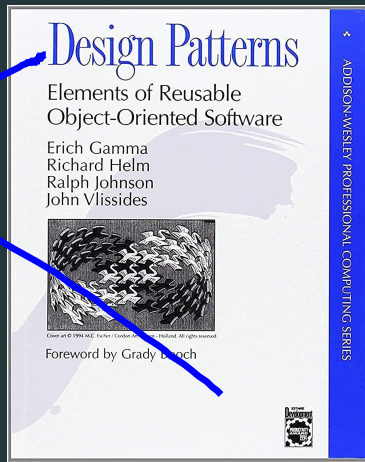
(Dai lucidi: 11 . *General Responsibility Assignment Software Patterns, GRASP*)

La nozione di pattern ebbe origine con i *pattern architettonici* (di costruzione) di *Christopher Alexander*.

I pattern per il software ebbero origine negli anni ottanta con *Ken Beck* (famoso anche per *Extreme Programming*) che riconobbe il lavoro svolto da Alexander nell'architettura, e furono sviluppati da Beck con *Ward Cunningham*.

# I design pattern GoF

(Dai lucidi: 11. *General Responsibility Assignment Software Patterns, GRASP*)  
Nel 1994 viene pubblicato il libro *Design Pattern* di *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* che descrive 23 pattern per la programmazione OO. Questi sono diventati noti come design pattern **GoF** (**Gang-of-Four**, "banda dei quattro").



# I design pattern GoF

I GoF sono più degli “schemi di progettazione avanzata” che “principi” (come nel caso di GRASP).

- Ciascun design pattern **descrive una soluzione progettuale comune a un problema di progettazione ricorrente**
- I design pattern GoF sono classificati in base al loro scopo:
  - **creazionale**
  - **strutturale**
  - **comportamentale**

# I design pattern GoF creazionali

Risolvono problematiche inerenti l'istanziamento degli oggetti:

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Lazy Initialization**
- **Prototype Pattern**
- **Singleton**
- **Double-check Locking**

# I design pattern GoF creazionali

Risolvono problematiche inerenti l'istanziamento degli oggetti:

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Lazy Initialization**
- **Prototype Pattern**
- **Singleton**
- **Double-check Locking**



# I design pattern GoF strutturali

Risolvono problematiche inerenti la struttura delle classi e degli oggetti:

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

# I design pattern GoF strutturali

Risolvono problematiche inerenti la struttura delle classi e degli oggetti:

- **Adapter**
- Bridge
- **Composite**
- **Decorator**
- Façade
- Flyweight
- Proxy

# I design pattern GoF comportamentali

Forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti:

- Chain of Responsibility
- Command
- Event Listener
- Hierarchical Visitor
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

# I design pattern GoF comportamentali

Forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti:

- Chain of Responsibility
- Command
- Event Listener
- Hierarchical Visitor
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

## GoF: preferire la composizione rispetto all'ereditarietà tra classi!

- Ereditarietà di classi:
  - Definiamo un oggetto in termini di un altro
  - Riutilizzo **white-box**: la visibilità della superclasse è la visibilità della sottoclasse (la **sottoclasse può accedere ai dettagli implementativi della superclasse**)
- Composizione di oggetti:
  - Le funzionalità sono ottenute **assemblando o componendo** gli oggetti per avere funzionalità più complesse
  - Riutilizzo **black-box**: i dettagli interni non sono conosciuti

# Favorire composizione rispetto ereditarietà

**GoF: preferire la composizione rispetto all'ereditarietà tra classi perché aiuta a mantenere le classi incapsulate e coese. La delegazione permette di rendere la composizione tanto potente quanto l'ereditarietà**

- Ereditarietà di classi:
  - **Definita staticamente**, non è possibile cambiarla a tempo di esecuzione: se una classe estende un'altra, questa relazione è definita nel codice sorgente, non può cambiare a runtime
  - **Una modifica alla sopraclasse potrebbe avere ripercussioni indesiderate** sul funzionamento di una classe che la estende (**non rispetta l'incapsulamento**)<sup>1</sup>
- Composizione di oggetti:
  - Se una classe usa un'altra classe, questa potrebbe essere referenziata attraverso una interfaccia, a runtime potrebbe esserci **una qualsiasi altra classe che implementa l'interfaccia**
  - La composizione attraverso un'interfaccia **rispetta l'incapsulamento**, solo una modifica all'interfaccia comporterebbe ripercussioni

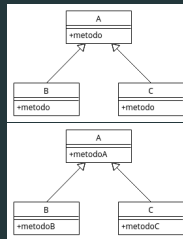
---

<sup>1</sup><http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.8552&rep=rep1&type=pdf>

# Favorire composizione rispetto ereditarietà

Il meccanismo di ereditarietà può essere utilizzato in due modi diversi:

- **Polimorfismo:** le sottoclassi possono essere scambiate una per l'altra, possono essere 'castate' in base al loro tipo, nascondendo il loro effettivo tipo alle classi cliente
- **Specializzazione:** le sottoclassi guadagnano elementi e proprietà rispetto la classe base, creando versioni specializzate rispetto alla classe base



I pattern GoF suggeriscono di diffidare della specializzazione, la quasi totalità dei pattern utilizza l'ereditarietà per creare polimorfismo.

In breve...

Il riuso è meglio ottenerlo attraverso il meccanismo di delega piuttosto che attraverso il meccanismo di ereditarietà, almeno per quanto riguarda l'utilizzo della specializzazione.

## Creazionali: Abstract Factory e Singleton

---



# Abstract Factory

## Abstract Factory

**Nome:** Abstract Factory

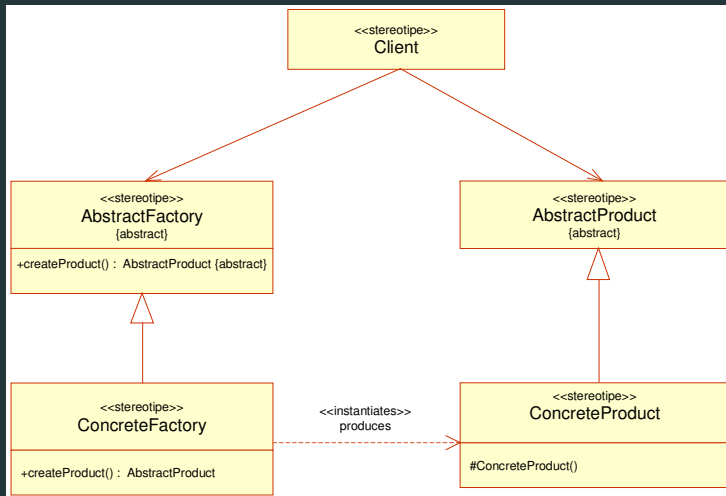
**Problema:** Come creare famiglie di classi correlate che implementano un'interfaccia comune?

**Soluzione:** Definire un'interfaccia factory (la factory astratta). Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Opzionalmente, definire una vera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono.

# Abstract Factory

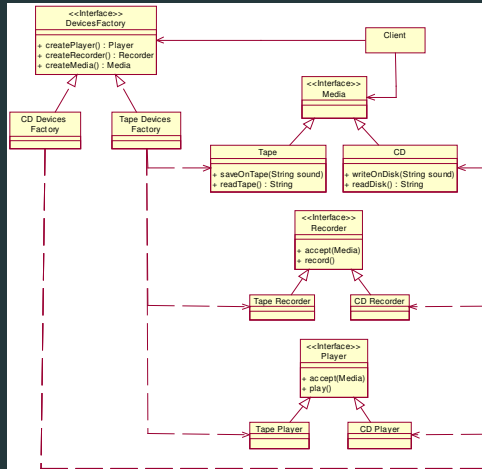
- Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che il cliente che li utilizza non abbia conoscenza delle loro concrete classi. Questo consente:
  - di assicurarsi che il cliente crei soltanto prodotti vincolati fra di loro
  - l'utilizzo di diverse famiglie di prodotti da parte dello stesso cliente
- Una variante comune di Abstract Factory consiste nel creare una classe astratta factory a cui si accede utilizzando il pattern Singleton
- È usata nelle librerie Java per la creazione di famiglie di elementi GUI per diversi sistemi operativi e sottosistemi GUI

# Struttura del pattern Abstract Factory



# Applicazione del pattern Abstract Factory

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 6–12.



# Singleton

## Singleton

**Nome:** Singleton

**Problema:** È consentita (o richiesta) esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

**Soluzione:** Definisci un metodo statico (di classe) della classe che restituisce l'oggetto singleton.

- Il “Singleton” pattern definisce una classe della quale è possibile la istanziazione di un unico oggetto, tramite l’invocazione a un metodo della classe, incaricato della produzione degli oggetti
- Le diverse richieste di istanziazione comportano la restituzione di un riferimento allo stesso oggetto
- In UML un singleton viene illustrato con un “1” nella sezione del nome, in alto a destra

# Struttura del pattern Singleton

<<stereotype>>

**Singleton**

-instance:Singleton

-Singleton()

+getNewInstance():Singleton

©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

# Singleton

Tre diverse implementazioni in Java:

- **Singleton come classe statica**: non è un vero e proprio Singleton, si lavora con la classe statica, non un oggetto. Questa classe statica ha metodi statici che offrono i servizi richiesti
- **Singleton creato da un metodo statico**: una classe che ha un metodo statico che deve essere chiamato per restituire l'istanza del Singleton. L'oggetto verrà istanziato **solo la prima volta**. Le successive sarà restituito un riferimento allo stesso oggetto (inizializzazione **pigra** <sup>2</sup>)
- **Singleton multi-thread**: versione multi-thread della soluzione precedente

---

<sup>2</sup>Contrapposta all'inizializzazione **golosa**: l'oggetto verrà istanziato quando la classe è caricata. È preferibile quella pigra perché se non si accede mai all'istanza viene evitato il lavoro della creazione. L'inizializzazione potrebbe essere complessa.

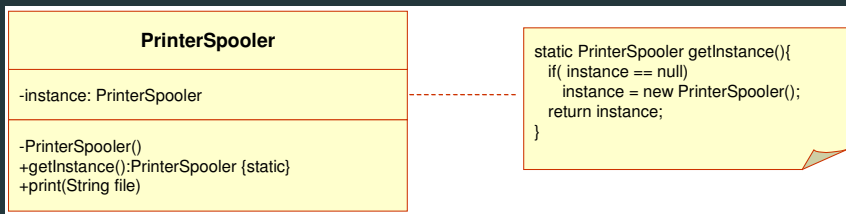


Il *Singleton* creato da un metodo statico è preferibile rispetto al *Singleton* come classe statica per tre motivi:

- I metodi d'istanza consentono la ridefinizione nelle sottoclassi e il **raffinamento della classe singleton in sottoclassi**
- La maggior parte dei **meccanismi di comunicazione remota** orientati agli oggetti supporta l'accesso remoto solo a metodi d'istanza
- Una classe non è sempre un singleton in tutti i contesti applicativi

# Applicazione del pattern Singleton

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 38–41.



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

## Singleton: si ponga attenzione a

- Presenza di Singleton in virtual machine multiple
- Singleton caricati contemporaneamente da diversi class loader
- Singleton distrutti dal garbage collector e dopo caricati quando sono necessari
- Presenza di istanze multiple come sottoclassi di un Singleton
- Copia di Singleton come risultato di un doppio processo di deserializzazione

## Strutturali: Adapter, Composite e Decorator

---

## Adapter

**Nome:** Adapter

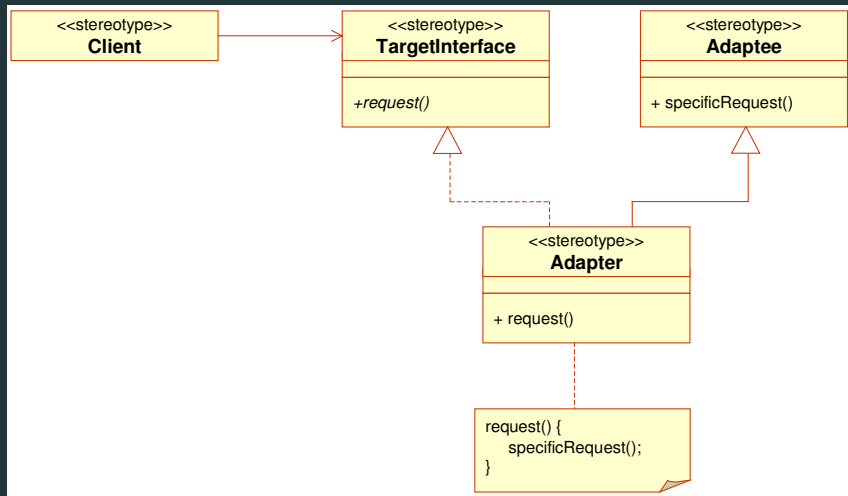
**Problema:** Come gestire **interfacce incompatibili, o fornire un'interfaccia stabile a comportamenti simili ma con interfacce diverse?**

**Soluzione:** Converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio.

- Si consideri una coppia di oggetti software in una relazione client-server. Si parla di interfacce incompatibili quando l'oggetto server **offre servizi di interesse per l'oggetto client ma l'oggetto client vuole fruire di questi servizi in una modalità diversa da** quella prevista dall'oggetto server **(interfacce incompatibili)**
- Ci sono più oggetti server che **offrono servizi simili; questi oggetti hanno interfacce simili ma diverse tra loro**. Un oggetto client vuole fruire dei servizi offerti da uno tra questi oggetti server **(componenti simili con interfacce diverse)**

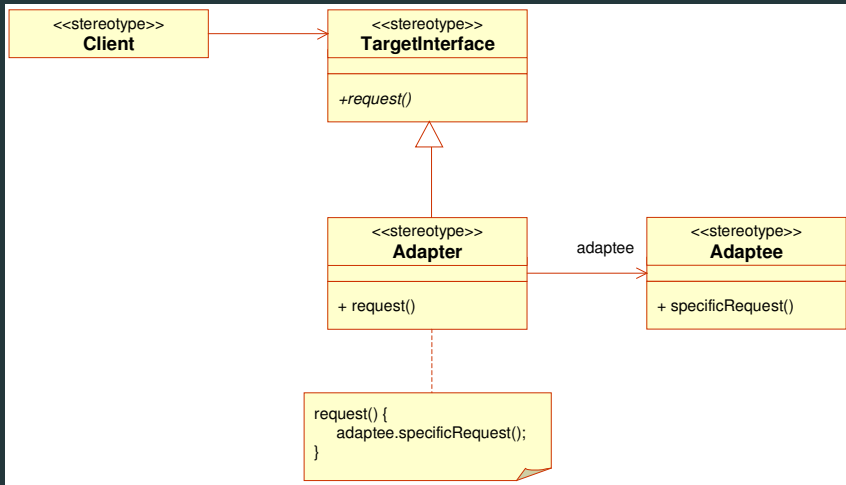
- In generale, un adattatore riceve richieste dai suoi client, per esempio da un oggetto dello strato di dominio, nel formato client dell'adattatore
- L'adattatore poi adatta, trasforma, una richiesta ricevuta in una richiesta nel formato del server
- L'adattatore invia la richiesta al server
- Se il server fornisce una risposta, lo fa nel formato del server
- L'adattatore adatta, trasforma, la risposta ricevuta dal server in una risposta nel formato del client e poi la restituisce al suo client

# Struttura del pattern Adapter (Class)



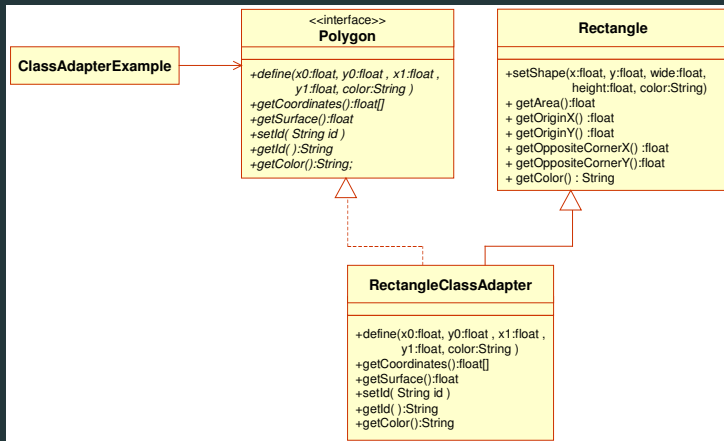


# Struttura del pattern Adapter (Object)



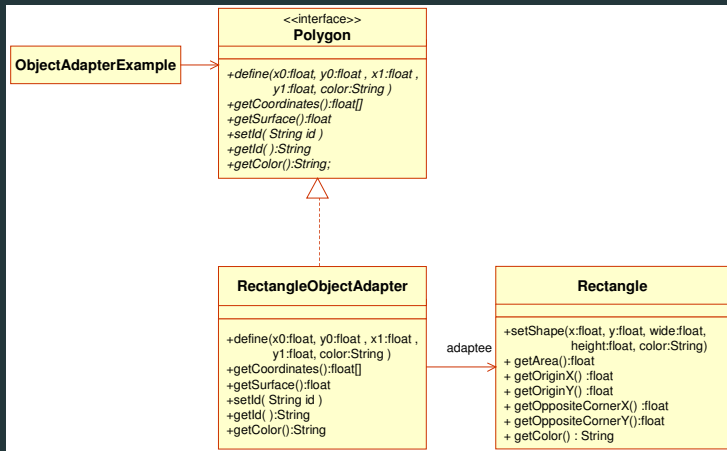
# Applicazione del pattern Adapter (Class)

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 42–49.



# Applicazione del pattern Adapter (Object)

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 42–49.



## Composite

**Nome:** Composite

**Problema:** Come trattare un gruppo o una struttura composta di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto (atomico)?

**Soluzione:** Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

Consente la costruzione di **gerarchie di oggetti composti**. Gli oggetti composti possono essere formati da oggetti singoli, oppure da altri oggetti composti.

Questo pattern è utile nei casi in cui si vuole:

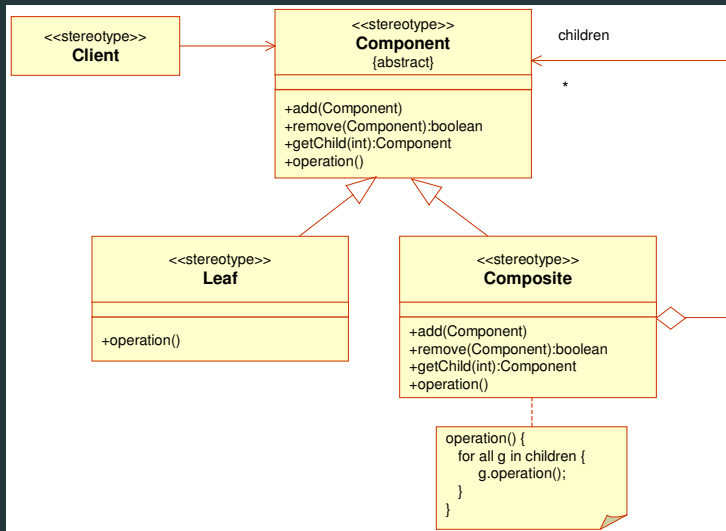
- Rappresentare gerarchie di oggetti **tutto-parte**
- Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti
- È nota anche come **struttura ad albero, composizione ricorsiva, struttura induttiva**: foglie e nodi hanno la stessa funzionalità
- Implementa la stessa interfaccia per tutti gli elementi contenuti

Il pattern definisce la classe astratta componente che deve essere estesa in due sottoclassi:

- Una che rappresenta i singoli componenti (foglia)
- L'altra che rappresenta i componenti composti e che si implementa come contenitore di componenti

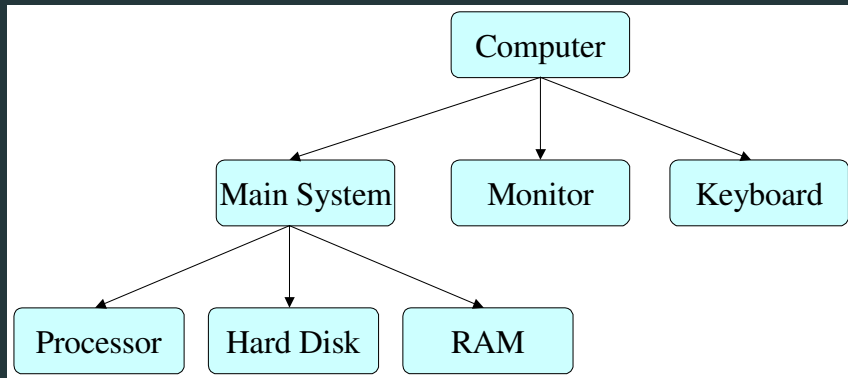
Nota: i componenti composti possono immagazzinare sia componenti singoli sia altri contenitori.

# Struttura del pattern Composite



# Applicazione del pattern Composite

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 57–62.

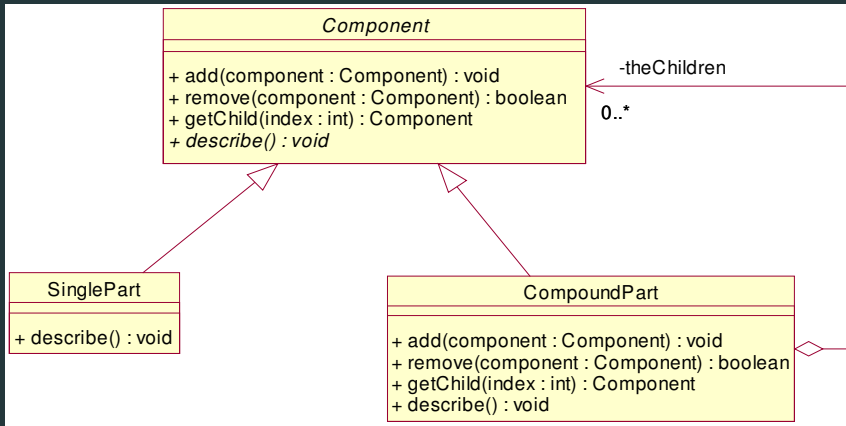


©F. Guidi Polanco. GoF's Design patterns in Java, 2002.



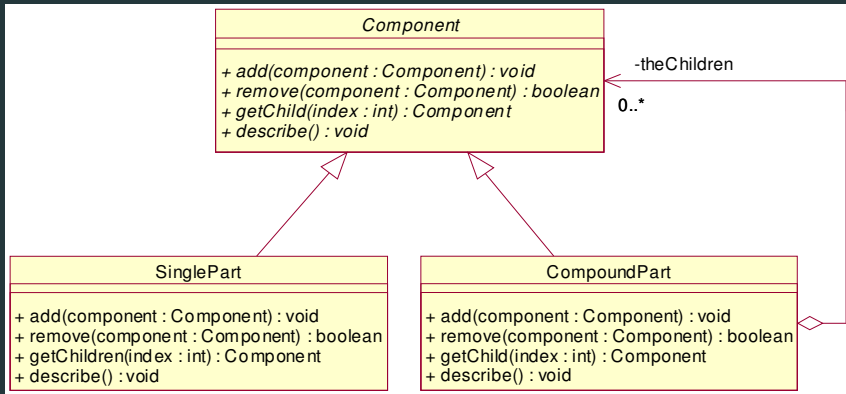
# Applicazione del pattern Composite

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 57–62.

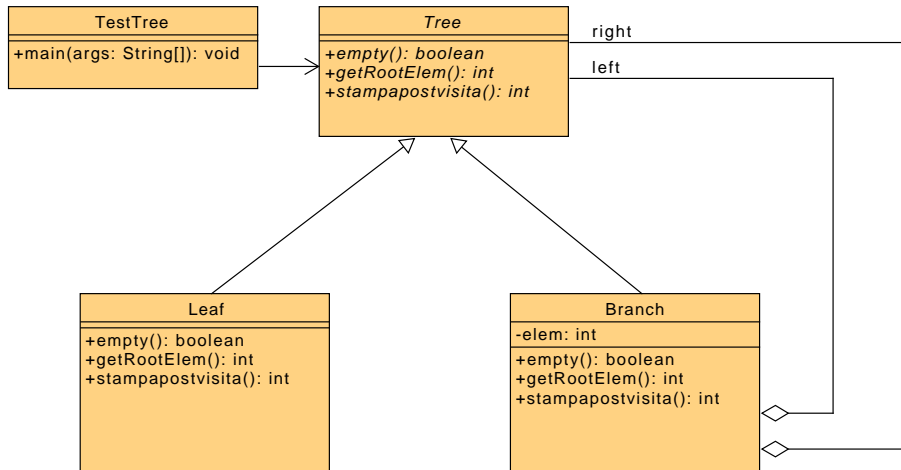


# Applicazione del pattern Composite (abstract component)

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 62–64.



# Applicazione del pattern Composite: Tree



## Applicazione del pattern Composite: Tree

```
1      public abstract class Tree {
2          public abstract boolean empty();
3          public abstract int getRootElem();
4          public abstract void stampapostvisita();
5      }
6
7      public class TestTree {
8          public static void main(String[] args) {
9              ...
10             System.out.println("Stampo albero postvisita");
11             t.stampapostvisita();
12             System.out.println("");
13         }
14     }
```

# Applicazione del pattern Composite: Tree

```
1      public class Leaf extends Tree {  
2          public Leaf() { }  
3          public boolean empty() { return true; }  
4          public int getRootElem() {  
5              assert false;  
6              return 0;  
7          }  
8          public void stampapostvisita() { }  
9      }
```

## Applicazione del pattern Composite: Tree

```
1      public class Branch extends Tree {
2          private int elem;
3          private Tree left;
4          private Tree right;
5          public Branch(int elem, Tree left,
6              Tree right) {
7              this.elem = elem;
8              this.left = left;
9              this.right = right;
10         }
11         public boolean empty() { return false; }
12         public int getRootElem() { return elem; }
13         public void stampapostvisita() {
14             right.stampapostvisita();
15             left.stampapostvisita();
16             System.out.print(this.getRootElem() + " ");
17         }
18     }
```

## In breve

Il pattern composite permette di costruire strutture **ricorsive** (ad esempio un albero di elementi) in modo che ad un cliente (una classe che usa la struttura) **l'intera struttura sia vista come una singola entità. Quindi l'interfaccia alle entità atomiche (foglie) è esattamente la stessa dell'interfaccia delle entità composte.** In essenza tutti gli elementi della struttura hanno la **stessa interfaccia** senza considerare se sono composti o atomici.

## Decorator

**Nome:** Decorator

**Problema:** Come permettere di assegnare **una o più responsabilità aggiuntive ad un oggetto in maniera dinamica ed evitare il problema della relazione statica?** Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?

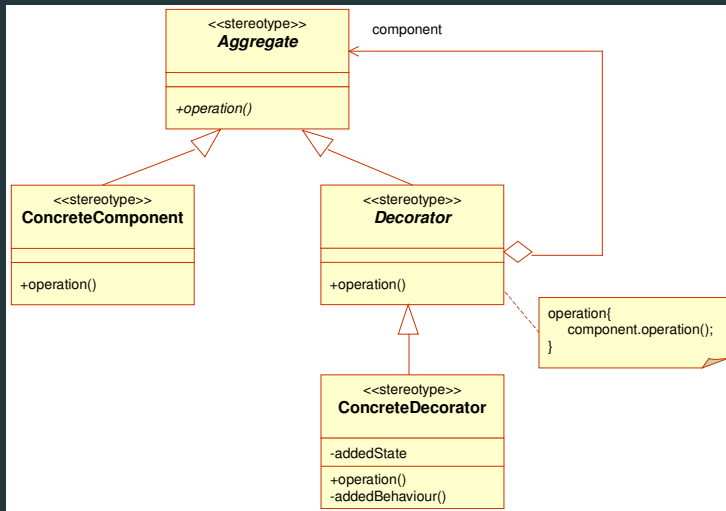
**Soluzione:** **Inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalità.**

Noto anche come **Wrapper.**



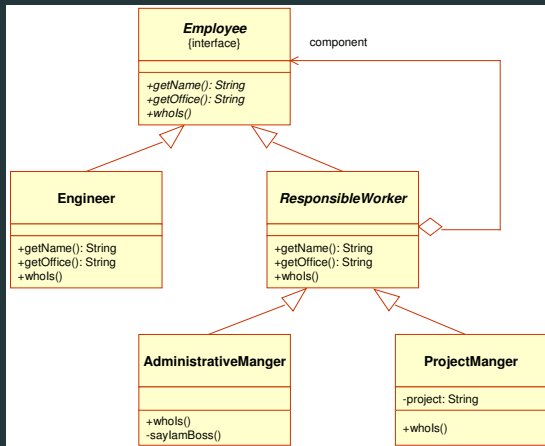
- Permette di aggiungere responsabilità ad oggetti individualmente, dinamicamente e in modo trasparente, ossia senza impatti sugli altri oggetti
- Le responsabilità possono essere ritirate
- Permette di evitare l'esplosione delle sottoclassi per supportare un ampio numero di estensioni e combinazioni di esse oppure quando le definizioni sono nascoste e non disponibili alle sottoclassi

# Struttura del pattern Decorator



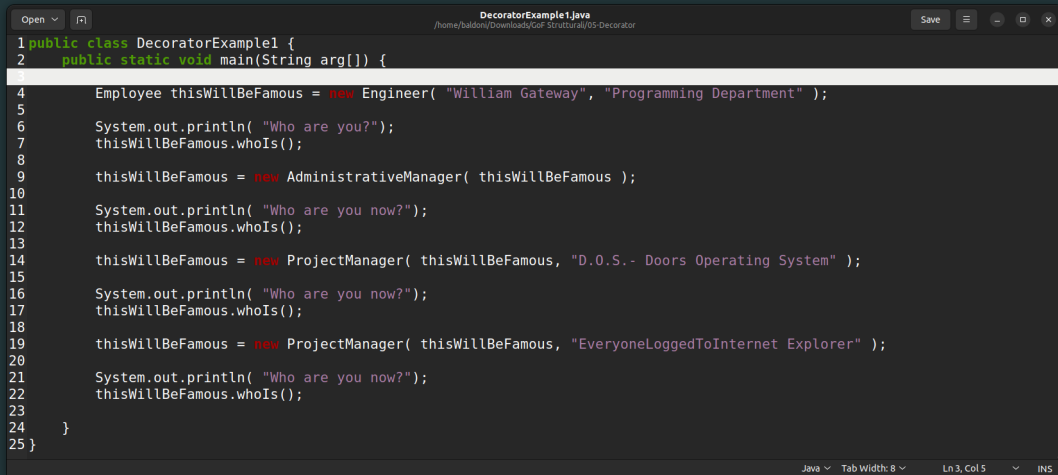
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



```
1 public class DecoratorExample1 {
2     public static void main(String arg[]) {
3
4         Employee thisWillBeFamous = new Engineer( "William Gateway", "Programming Department" );
5
6         System.out.println( "Who are you?" );
7         thisWillBeFamous.whoIs();
8
9         thisWillBeFamous = new AdministrativeManager( thisWillBeFamous );
10
11        System.out.println( "Who are you now?" );
12        thisWillBeFamous.whoIs();
13
14        thisWillBeFamous = new ProjectManager( thisWillBeFamous, "D.O.S.- Doors Operating System" );
15
16        System.out.println( "Who are you now?" );
17        thisWillBeFamous.whoIs();
18
19        thisWillBeFamous = new ProjectManager( thisWillBeFamous, "EveryoneLoggedToInternet Explorer" );
20
21        System.out.println( "Who are you now?" );
22        thisWillBeFamous.whoIs();
23
24    }
25 }
```

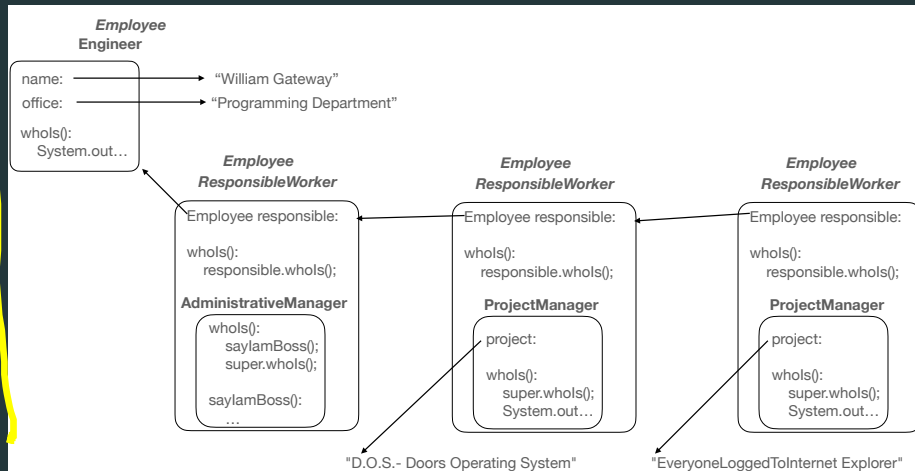
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.

```
baldoni@Shizuka: ~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-Decorator
baldoni@Shizuka:~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-Decorator$ java DecoratorExample1
Who are you?
I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
Who are you now?
I am a boss. I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
Who are you now?
I am a boss. I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
I am the Manager of the Project:D.O.S.- Doors Operating System
Who are you now?
I am a boss. I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
I am the Manager of the Project:D.O.S.- Doors Operating System
I am the Manager of the Project:EveryoneLoggedInToInternet Explorer
baldoni@Shizuka:~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-Decorator$
```

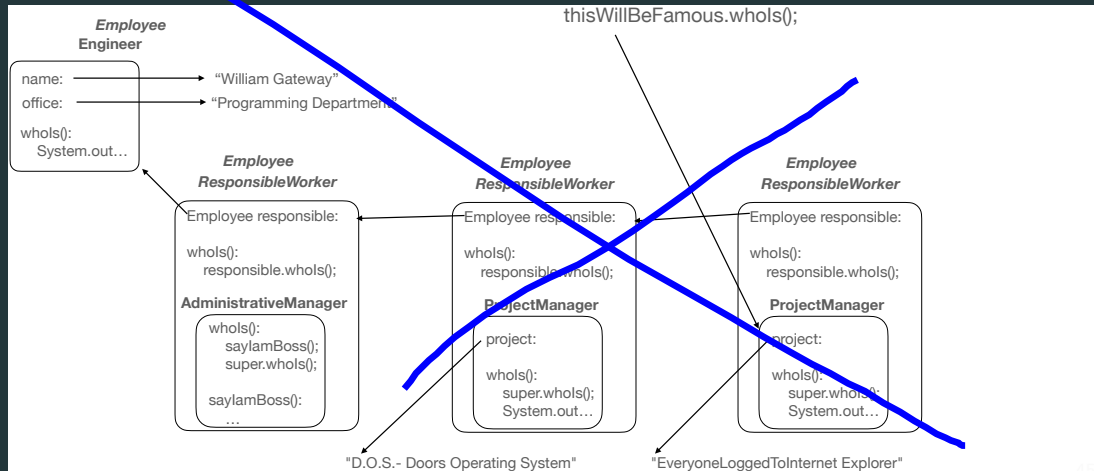
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



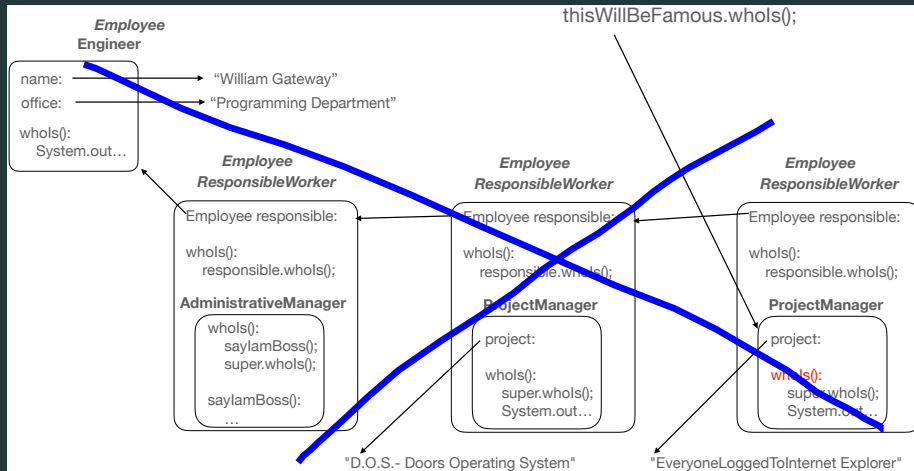
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



## Applicazione del pattern Decorator

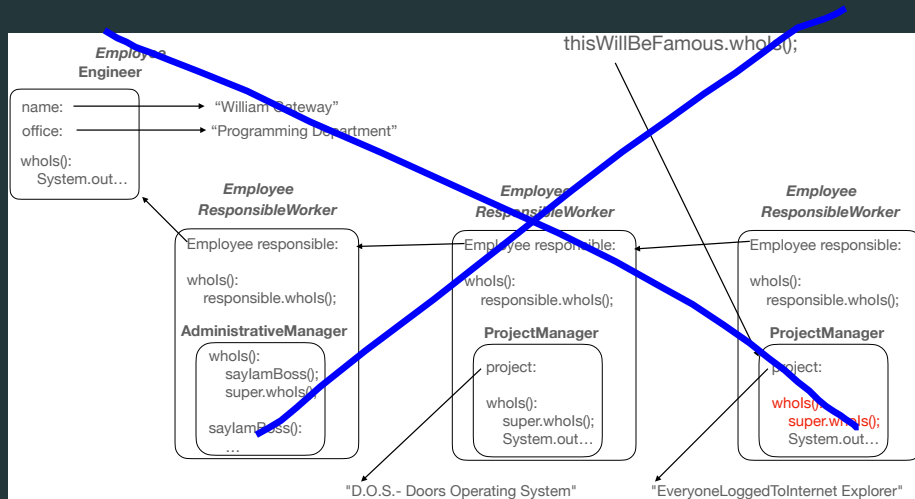
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.





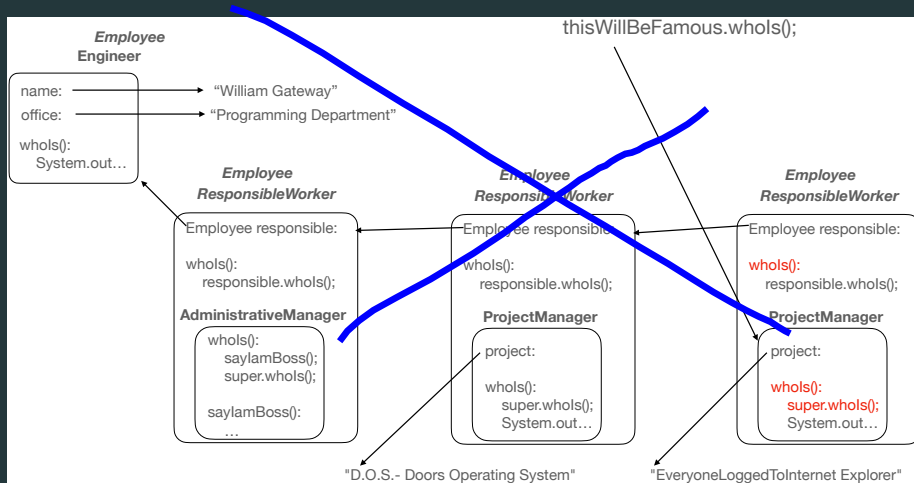
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



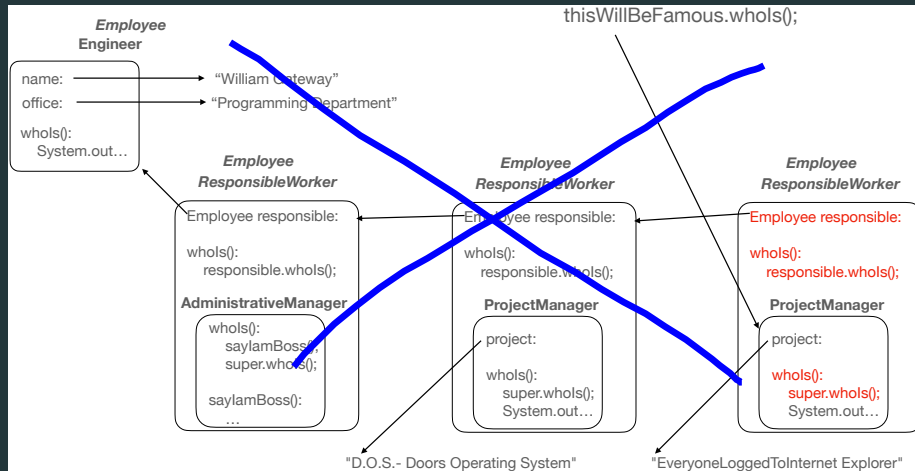
## Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



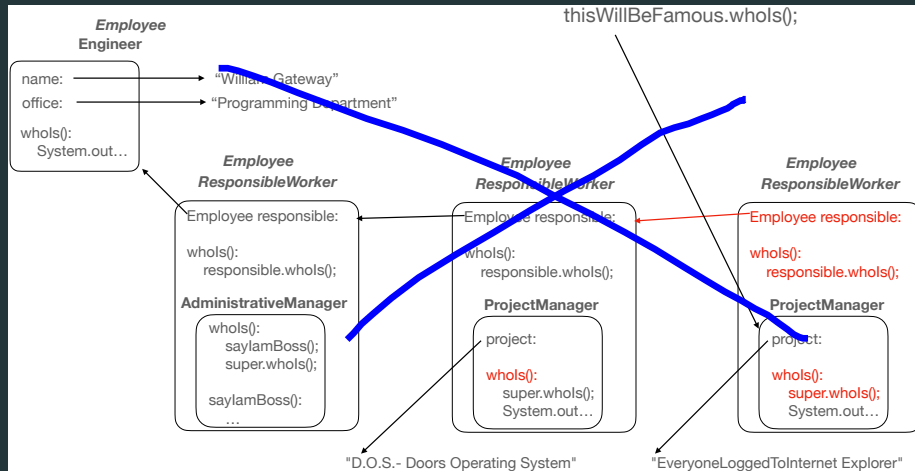
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



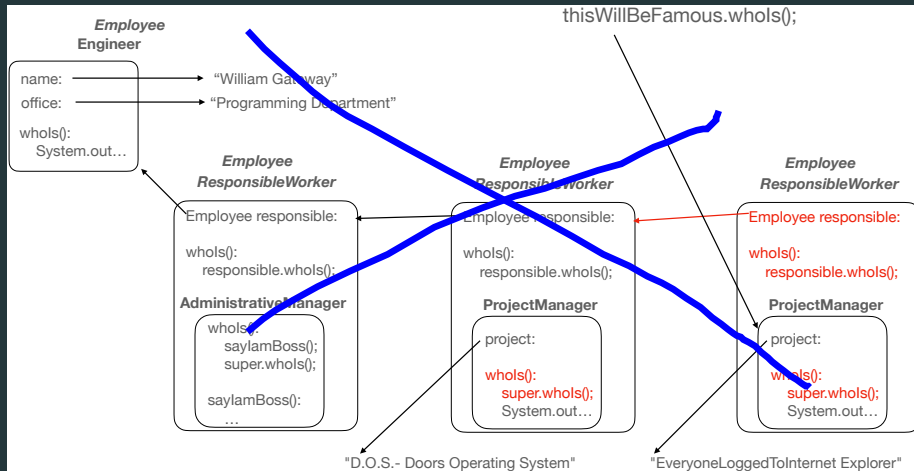
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



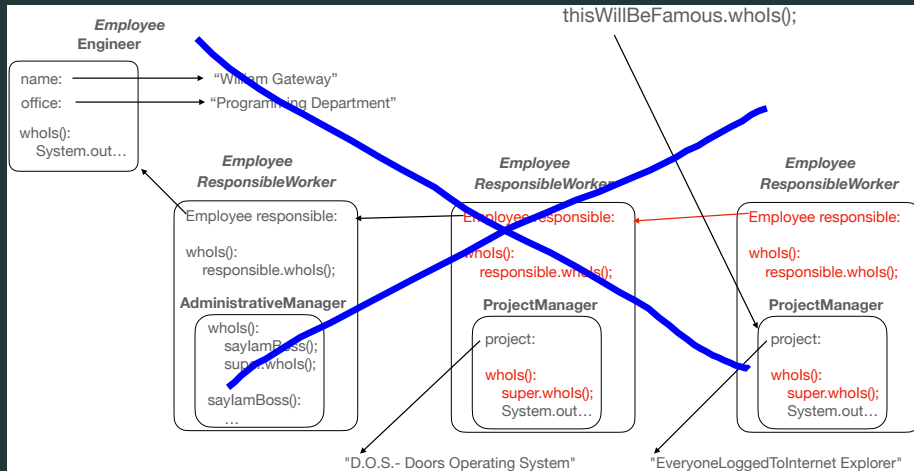
## Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



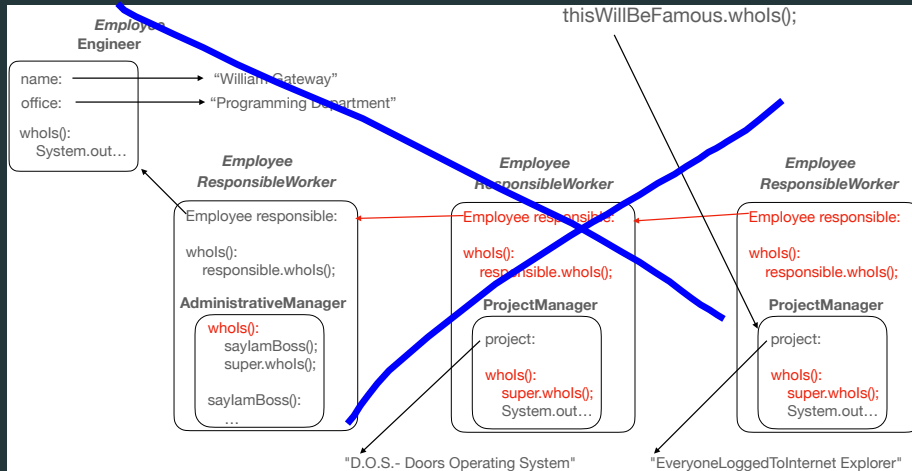
## Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



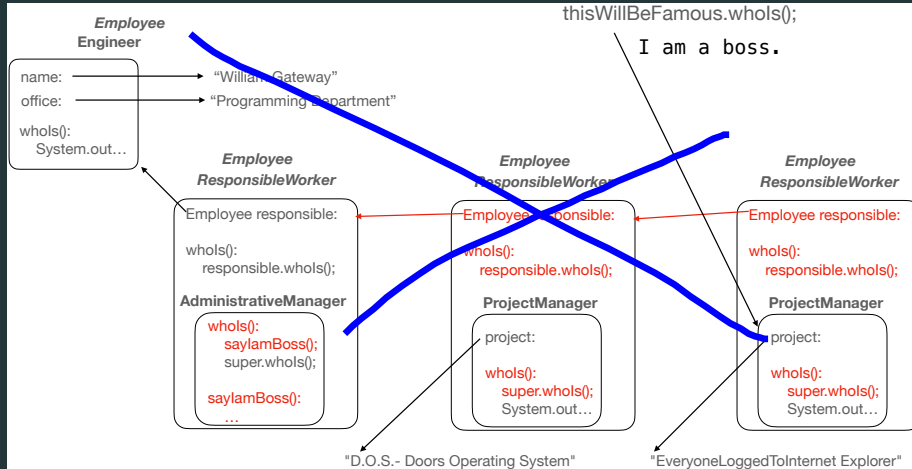
## Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



## Applicazione del pattern Decorator

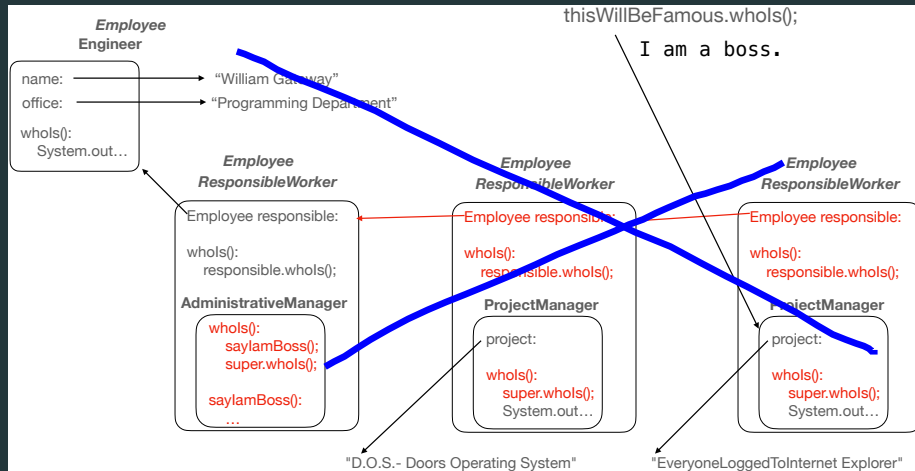
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.





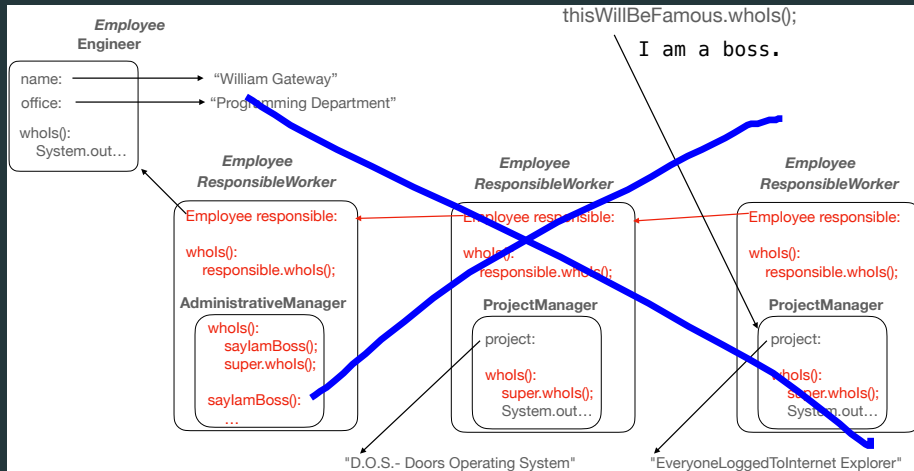
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



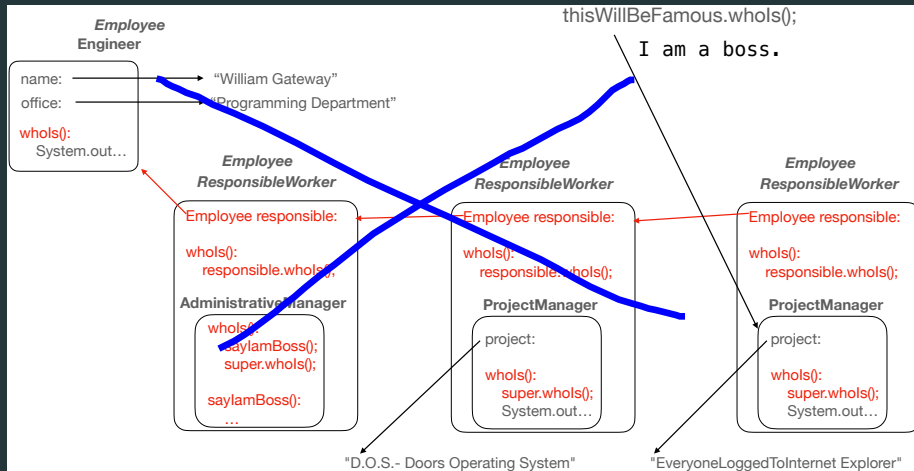
## Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



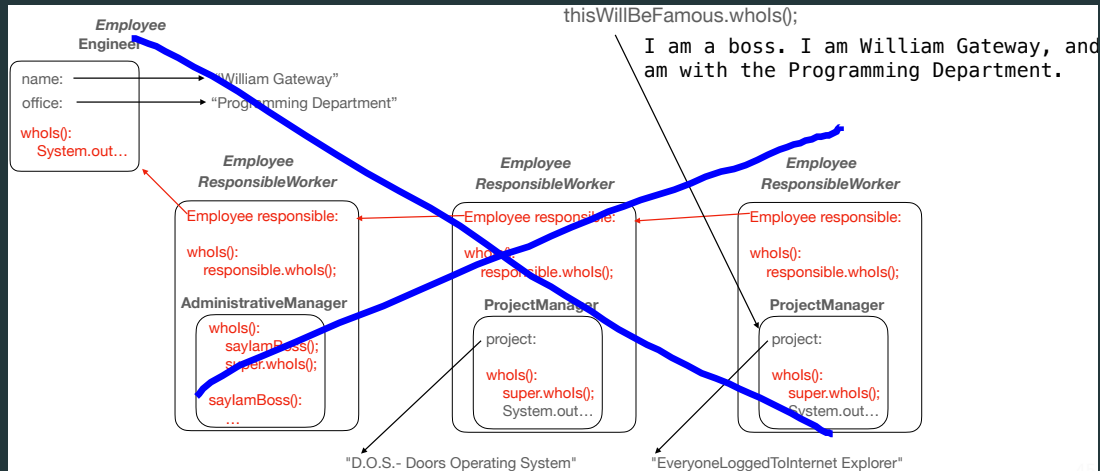
## Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



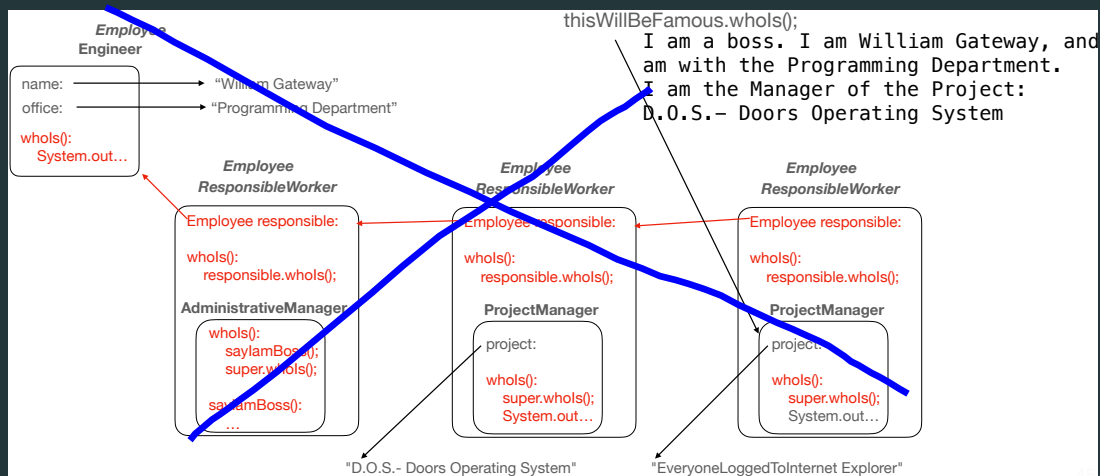
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



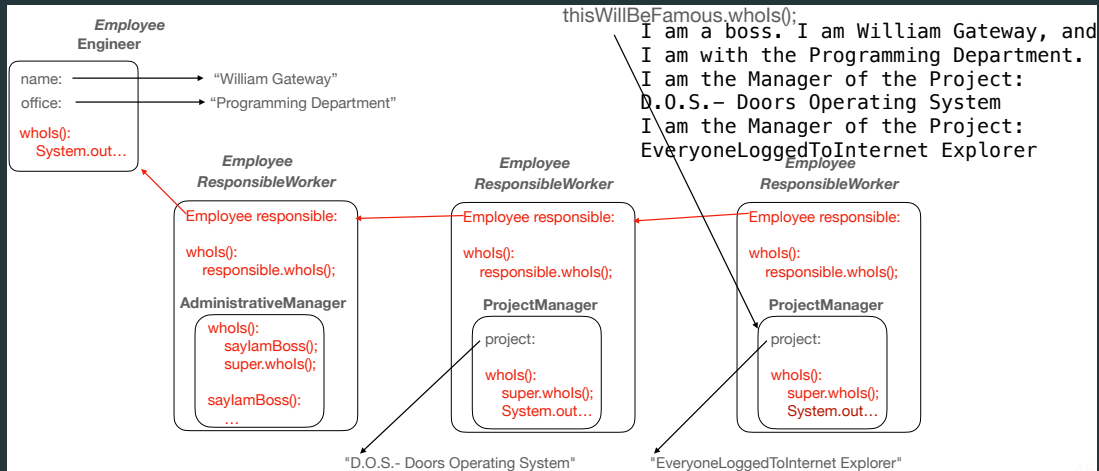
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.

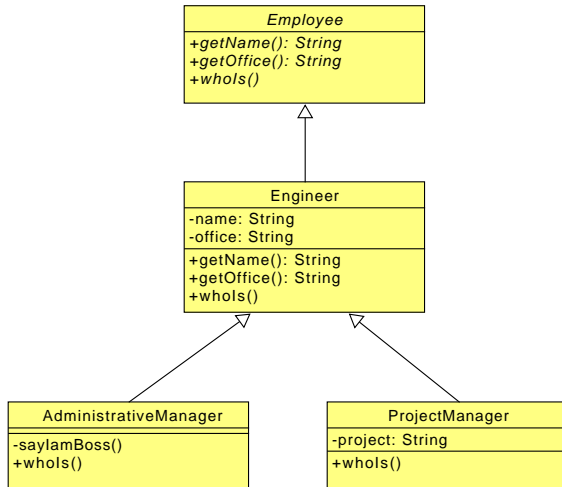


# Applicazione del pattern Decorator

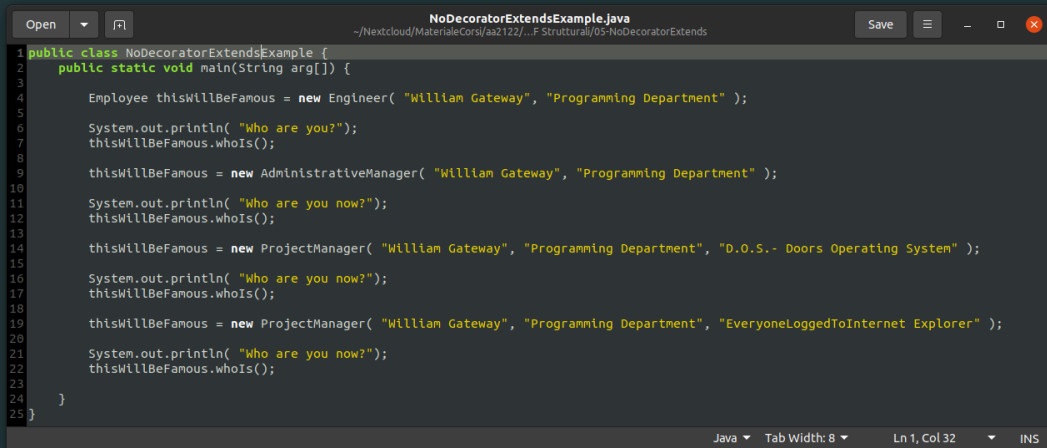
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



## Senza usare Decorator: Estensione



# Senza usare Decorator: Estensione



```
1 public class NoDecoratorExtendsExample {
2     public static void main(String arg[]) {
3
4         Employee thisWillBeFamous = new Engineer( "William Gateway", "Programming Department" );
5
6         System.out.println( "Who are you?" );
7         thisWillBeFamous.whoIs();
8
9         thisWillBeFamous = new AdministrativeManager( "William Gateway", "Programming Department" );
10
11        System.out.println( "Who are you now?" );
12        thisWillBeFamous.whoIs();
13
14        thisWillBeFamous = new ProjectManager( "William Gateway", "Programming Department", "D.O.S. - Doors Operating System" );
15
16        System.out.println( "Who are you now?" );
17        thisWillBeFamous.whoIs();
18
19        thisWillBeFamous = new ProjectManager( "William Gateway", "Programming Department", "EveryoneLoggedToInternet Explorer" );
20
21        System.out.println( "Who are you now?" );
22        thisWillBeFamous.whoIs();
23
24    }
25 }
```

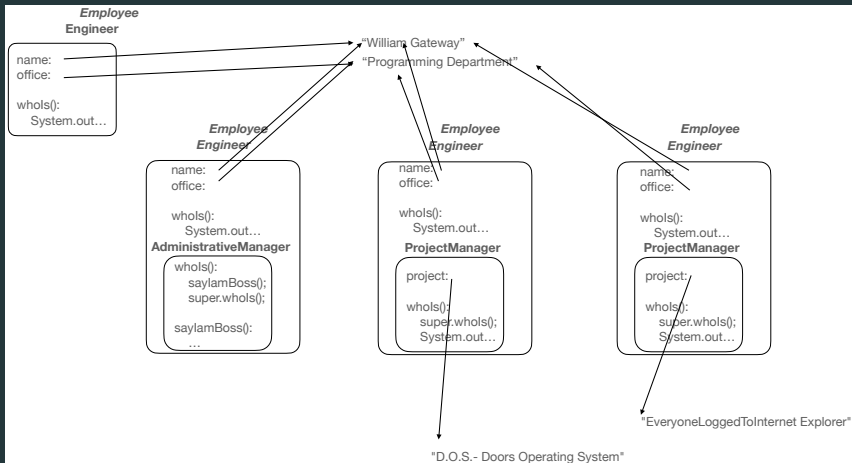
Java Tab Width: 8 Ln 1, Col 32 INS



## Senza usare Decorator: Estensione

```
baldoni@Shizuka: ~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-NoDecoratorExtends
baldoni@Shizuka:~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-NoDecoratorExtends$ java NoDecorator
ExtendsExample
Who are you?
I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
Who are you now?
I am a boss. I am William Gateway (AdministrativeManager@3e3abc88), and I am with the Programming Department.
Who are you now?
I am William Gateway (ProjectManager@53d8d10a), and I am with the Programming Department.
I am the Manager of the Project:D.O.S.- Doors Operating System
Who are you now?
I am William Gateway (ProjectManager@214c265e), and I am with the Programming Department.
I am the Manager of the Project:EveryoneLoggedInToInternet Explorer
baldoni@Shizuka:~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-NoDecoratorExtends$
```

## Senza usare Decorator: Estensione



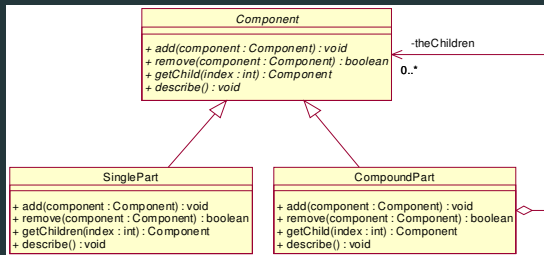
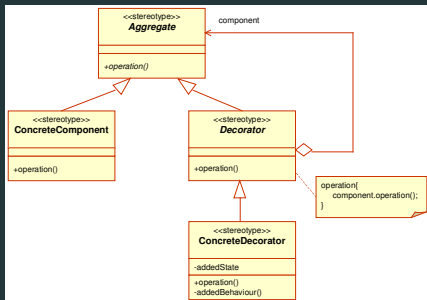
Non è assolutamente quanto desideriamo! Non è possibile ottenere lo stesso risultato! Oggetti diversi, administrative manager e project manager in esclusione, project manager di un solo progetto, ...

## In breve

Il pattern decorator permette ad una entità di **contenere** completamente un'altra entità così che l'utilizzo del decorator sia identico all'entità contenuta. Questo consente al decorator di **modificare il comportamento e/o il contenuto di tutto ciò che sta in** capsulato senza cambiare l'aspetto esteriore dell'entità. Ad esempio, è possibile utilizzare un decorator per **aggiungere l'attività di logging dell'elemento contenuto senza cambiare il comportamento di questo.**

## Composite vs Decorator

- **Il pattern composite:** fornisce un'interfaccia comune a elementi atomici (foglie) e composti
- **Il pattern decorator:** fornisce caratteristiche aggizionali ad elementi atomici (foglie), mantenendo un'interfaccia comune



## Comportamentali: Observer, State, Strategy e Visitor

---

# Observer

## Observer

**Nome:** Observer

**Problema:** Diversi tipi di oggetti subscriber (abbonato) sono interessati ai cambiamenti di stato o agli eventi di un oggetto publisher (editore). Ciascun subscriber vuole reagire in un suo modo proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

**Soluzione:** Definisci un'interfaccia subscriber o listener (ascoltatore). Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

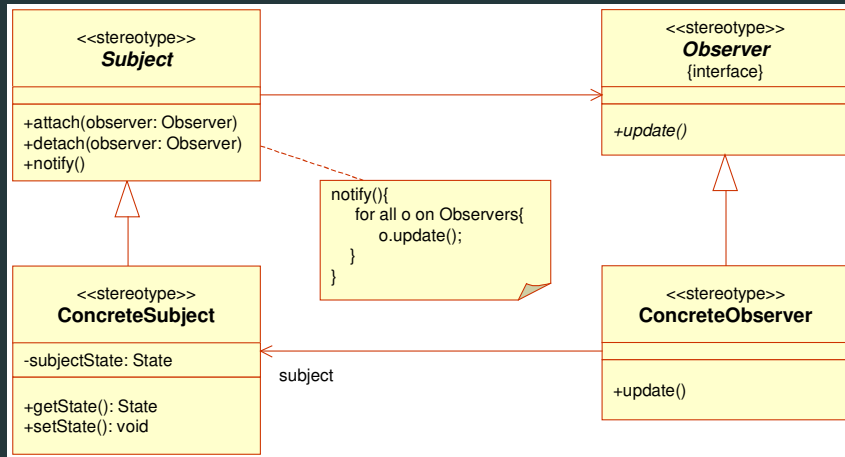
Noto anche come Dependents, Publish-Subscribe.

# Observer

- Definisce una dipendenza tra oggetti di tipo **uno-a-molti**: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, **essi vengono automaticamente aggiornati**
- L'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti notificati: **le due tipologie di oggetti sono disaccoppiati**
- Il numero degli oggetti affetti dal cambiamento di **stato di un oggetto non è noto a priori**
- Fornisce un modo per accoppiare in maniera debole degli oggetti che devono comunicare (eventi). I publisher conoscono i subscriber solo attraverso un'interfaccia, e i subscriber possono registrarsi (o cancellare la propria registrazione) dinamicamente con il publisher

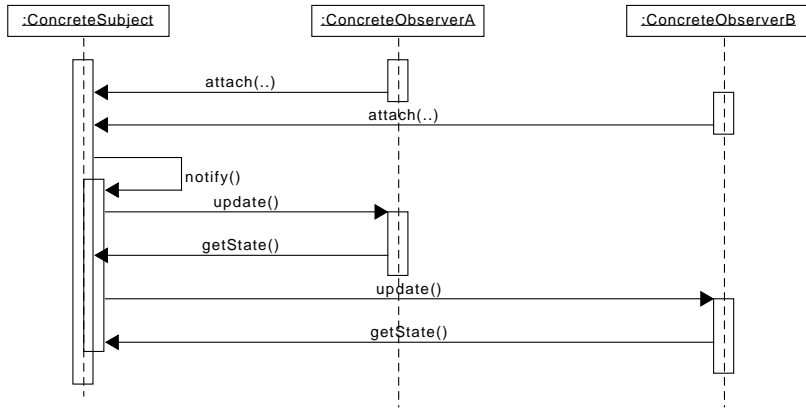
Spesso associato al pattern architetturale Model-View-Controller (MVC): le modifiche al modello sono notificate agli osservatori che sono le viste.

# Struttura del pattern Observer



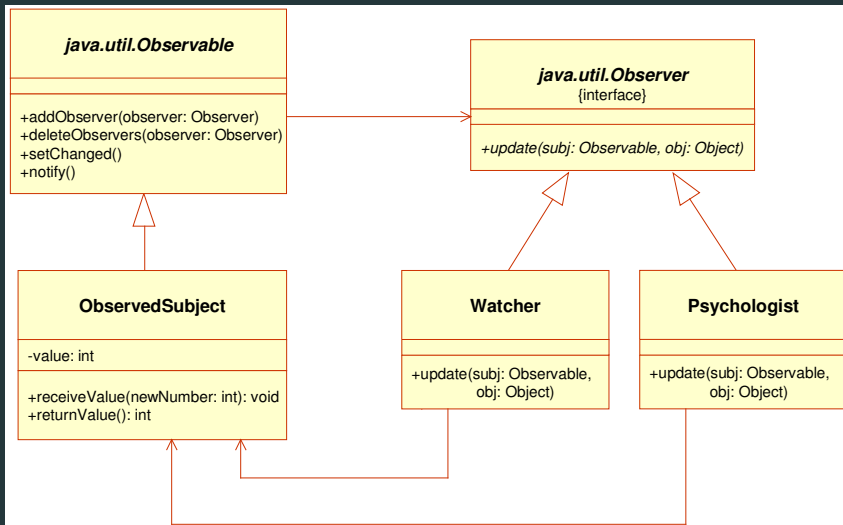


# Struttura del pattern Observer



# Applicazione del pattern Observer

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 129–135.



## State

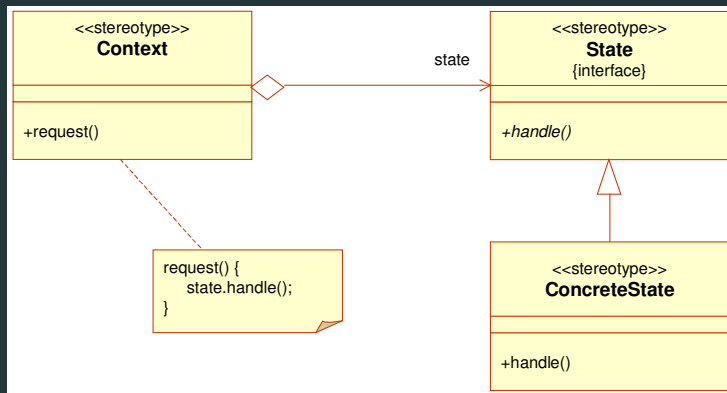
**Nome:** State

**Problema:** Il comportamento di un oggetto **dipende da suo stato** e i suoi metodi contengono logica condizionale per casi che riflette le azioni condizionali che dipendono dallo stato. **C'è un'alternativa alla logica condizionale?**

**Soluzione:** **Crea delle classi stato per ciascuno stato, che implementano un'interfaccia comune. Delega le operazioni che dipendono dallo stato** dall'oggetto contesto all'oggetto stato corrente corrispondente. Assicura che **l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente.**

- Permette ad un oggetto di modificare il suo comportamento quando cambia il suo stato interno
- Può sembrare che l'oggetto modifichi la sua classe

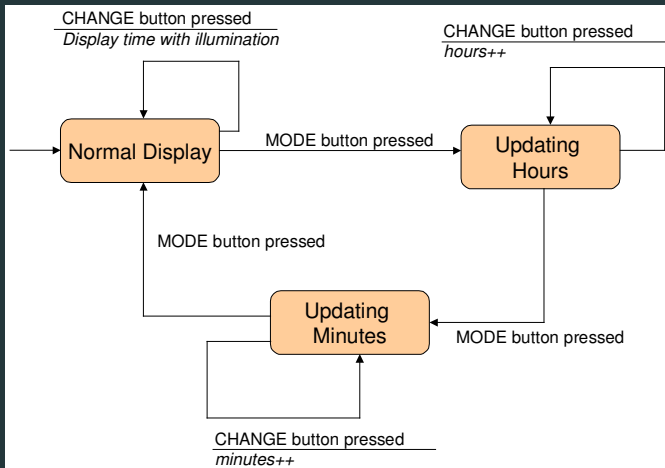
# Struttura del pattern State



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

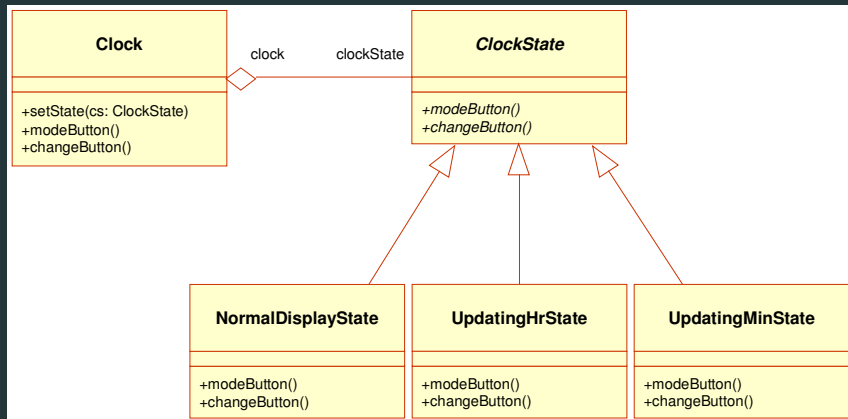
# Applicazione del pattern State

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 136–141.



# Applicazione del pattern State

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 136–141.



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

## Strategy

**Nome:** Strategy

**Problema:** Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? **Come progettare per consentire di modificare questi algoritmi o politiche?**

**Soluzione:** Definisci ciascun algoritmo/politica/strategia in una classe separata, con **un'interfaccia comune.**

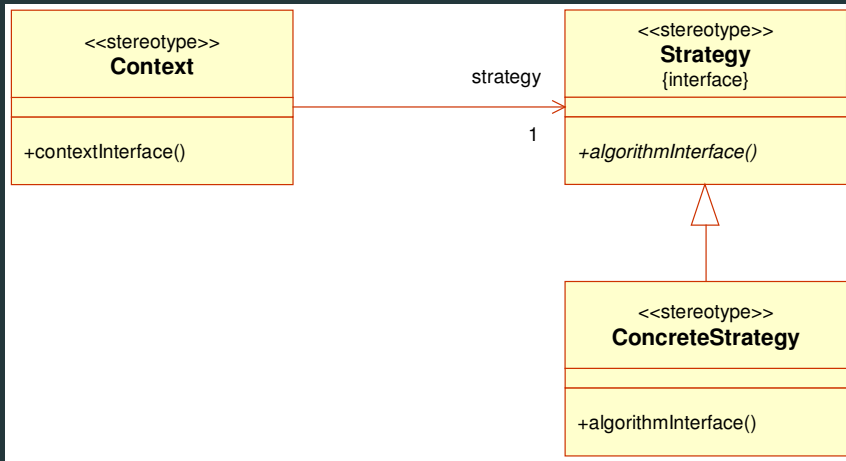
Noto anche come *Policy*.



- L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo
- L'oggetto contesto è associato a un oggetto strategia, che è l'oggetto che implementa un algoritmo

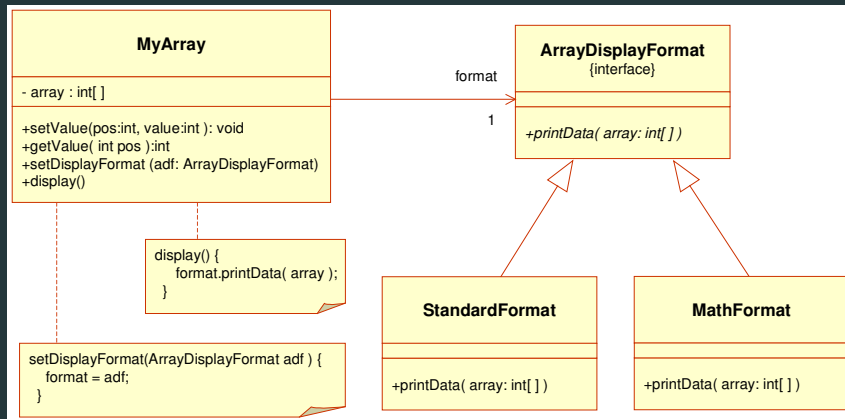
- Consente la definizione di una **famiglia di algoritmi**, i incapsula ognuno e li rende intercambiabili tra di loro
- Permette di modificare gli algoritmi in modo **indipendente dai** clienti che fanno uso di essi
- **Disaccoppia** gli algoritmi dai clienti che vogliono usarli dinamicamente
- Permette che un oggetto client possa usare **indifferentemente** uno o l'altro algoritmo
- È utile dove è necessario **modificare il comportamento a runtime** di una classe
- **Usa la composizione invece dell'ereditarietà**: i comportamenti di una classe non dovrebbero essere ereditati ma piuttosto incapsulati usando la dichiarazione di interfaccia

# Struttura del pattern Strategy



# Applicazione del pattern Strategy

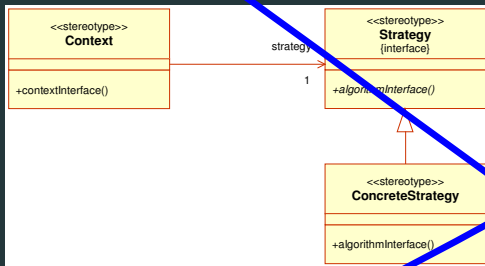
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 142–145.



# Strategy vs State

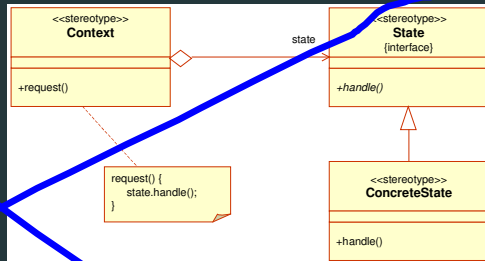
In pratica sono molto simili...

## Pattern Strategy



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

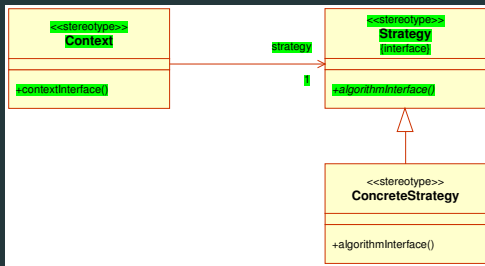
## Pattern State



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

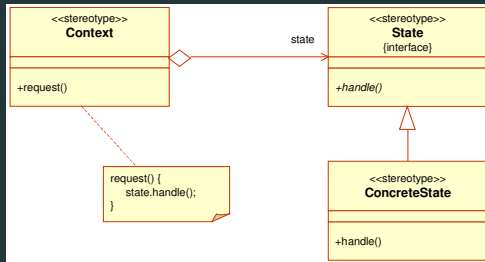
# Strategy vs State

## Pattern Strategy



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

## Pattern State

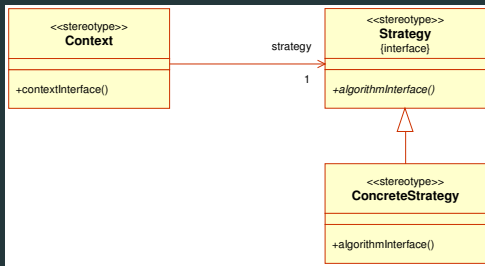


©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

Il **pattern State** si occupa di che cosa (stato o tipo) un oggetto è (al suo interno) e incapsula **un comportamento dipendente dallo stato**. Fare cose diverse in base allo stato, lasciando il chiamante sollevato dall'onere di soddisfare ogni stato possibile.

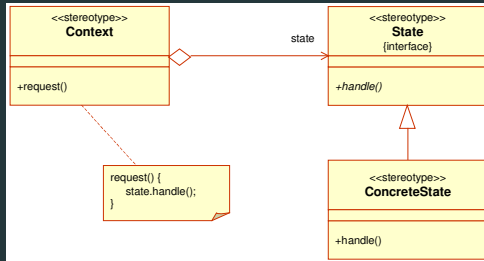
# Strategy vs State

## Pattern Strategy



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

## Pattern State



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

*mentre...*

Il **pattern Strategy** si occupa del modo in cui un oggetto esegue un **determinato compito: incapsula un algoritmo**. Un'implementazione diversa che realizza (fondamentalmente) la stessa cosa, in modo che un'implementazione possa sostituire l'altra a seconda della strategia richiesta.

## Visitor

**Nome:** Visitor

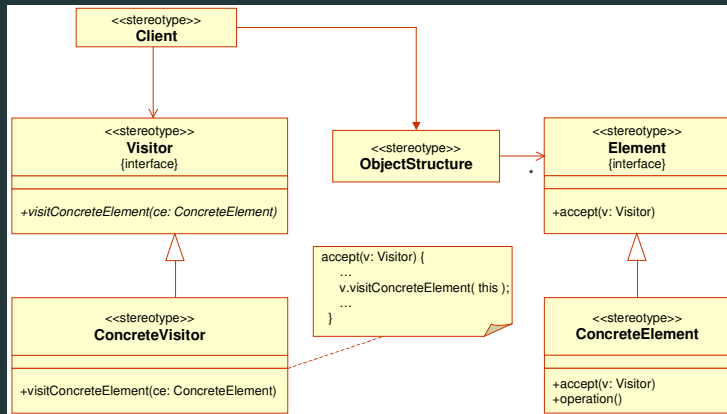
**Problema:** Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni dipendenti dal tipo degli elementi?

**Soluzione:** Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce ad un'interfaccia (Visitable) che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.



- Flessibilità delle operazioni
- Organizzazione logica
- Visita di vari tipi di classe
- Mantenimento di uno stato aggiornabile ad ogni visita
- Le diverse modalità di visita della struttura possono essere definite come sottoclassi del Visitor.

# Struttura del pattern Visitor

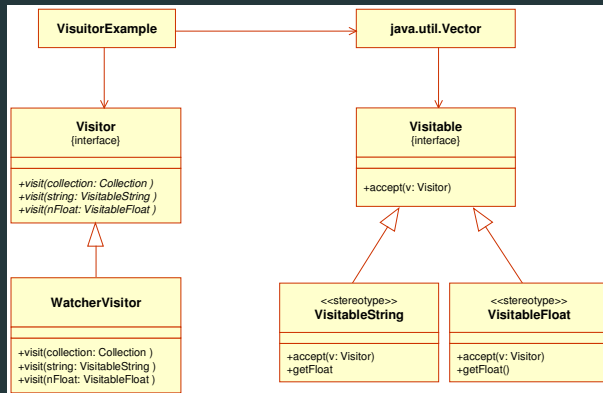


©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

**Errata corrige:** la linea tratteggiata tra la nota e ConcreteVisitor è errata, la nota deve essere collegata con ConcreteElement.

# Applicazione del pattern Visitor

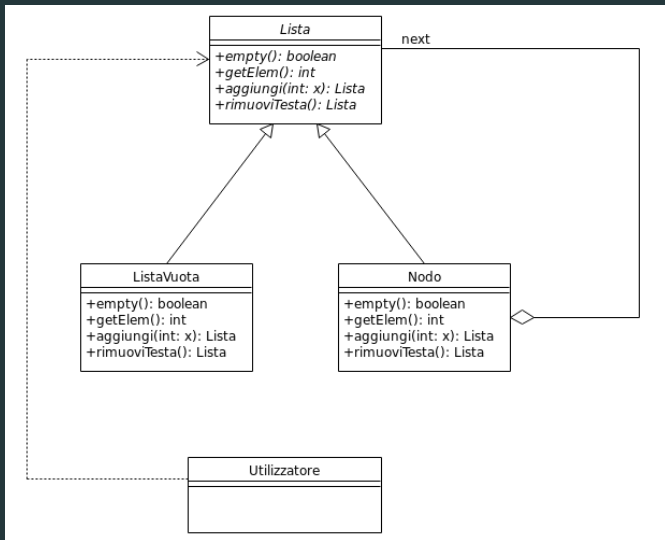
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 152–159.



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

**Errata corrige:** Nella classe **VisitableString** `getFloat` è `getString`.

# Applicazione del pattern Visitor



# Applicazione del pattern Visitor

