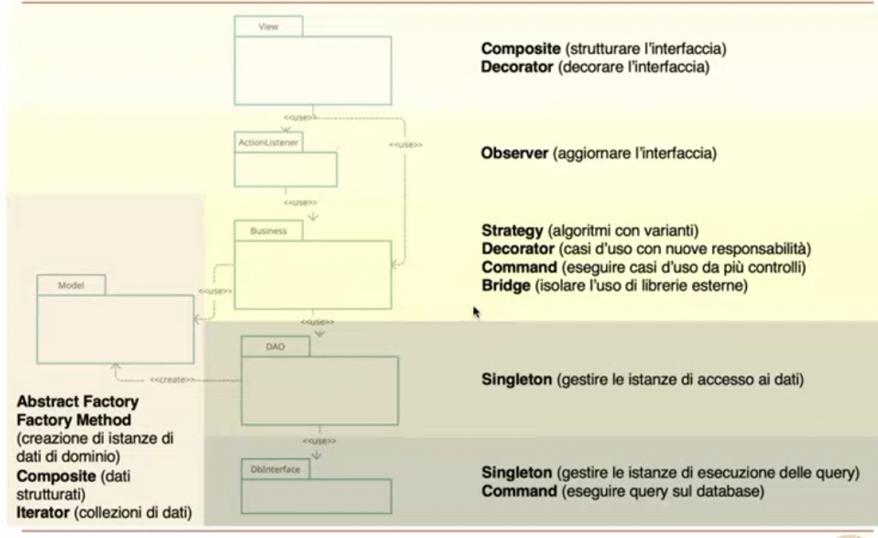


# Design Patterns

rielaborato da Aprile Matteo

3 settembre 2021

## Design pattern suggeriti a ogni livello di logica



Architettura di progetto e design pattern

3

Luca Mainetti



# Indice

<b>I Creational Patterns</b>	<b>6</b>
<b>1 Abstract Factory</b>	<b>7</b>
1.1 Intento . . . . .	7
1.2 Struttura . . . . .	7
1.3 Implementazione . . . . .	7
1.3.1 Factory come Singleton. . . . .	7
1.3.2 Creazione dei Product. . . . .	8
1.4 Esempio Java . . . . .	8
1.4.1 FactoryProducer.java . . . . .	8
1.4.2 AbstractFactory.java . . . . .	9
1.4.3 ShapeFactory.java . . . . .	9
1.4.4 RoundedShapeFactor.java . . . . .	9
1.4.5 Shape.java . . . . .	9
1.4.6 Rectangle.java . . . . .	10
1.4.7 Square.java . . . . .	10
1.4.8 RoundedRectangle.java . . . . .	10
1.4.9 RoundedSquare.java . . . . .	10
1.4.10 main . . . . .	11
<b>2 Factory Method</b>	<b>12</b>
2.1 Intento . . . . .	12
2.2 Struttura . . . . .	12
2.3 Implementazione . . . . .	12
2.3.1 Due varietà principali. . . . .	12
2.3.2 Factory Method parametrizzati. . . . .	13
2.3.3 Utilizzo di modelli per evitare la sottoclasse. . . . .	14
2.4 Esempio Java . . . . .	15
2.4.1 Circle.java . . . . .	15
2.4.2 Rectangle.java . . . . .	15
2.4.3 Shape.java . . . . .	16
2.4.4 ShapeFactory.java . . . . .	16
2.4.5 main . . . . .	16

<b>3 Singleton</b>	<b>17</b>
3.1 Intento . . . . .	17
3.2 Struttura . . . . .	17
3.3 Implementazione . . . . .	17
3.3.1 Garantire un’istanza univoca. . . . .	17
3.3.2 Sottoclasse della classe Singleton. . . . .	18
3.4 Esempio Java . . . . .	20
3.4.1 Singleton.java . . . . .	20
3.4.2 main . . . . .	21
<b>II Structural Patterns</b>	<b>22</b>
<b>4 Bridge</b>	<b>23</b>
4.1 Intento . . . . .	23
4.2 Prestruttura . . . . .	23
4.3 Struttura . . . . .	24
4.4 Implementazione . . . . .	24
4.4.1 Solo un’implementazione. . . . .	24
4.4.2 Creazione dell’oggetto Implementor corretto. . . . .	24
4.5 Esempio Java . . . . .	25
4.5.1 DrawImplementation.java . . . . .	25
4.5.2 Shape.java . . . . .	25
4.5.3 Circle.java . . . . .	25
4.5.4 RedCircle.java . . . . .	26
4.5.5 GreenCircle.java . . . . .	26
4.5.6 main . . . . .	26
<b>5 Composite</b>	<b>27</b>
5.1 Intento . . . . .	27
5.2 Prestruttura . . . . .	27
5.3 Struttura . . . . .	28
5.4 Implementazione . . . . .	28
5.4.1 Condivisione di Component. . . . .	28
5.4.2 Massimizzare l’interfaccia del componente. . . . .	28
5.4.3 Dichiarazione delle operazioni di gestione del figlio. . . . .	28
5.4.4 Il Component dovrebbe implementare un elenco di Component? . . . . .	30
5.4.5 Memorizzazione nella cache per migliorare le prestazioni. . . . .	30
5.4.6 Chi dovrebbe eliminare i Component? . . . . .	30
5.5 Esempio Java . . . . .	31
5.5.1 Employee.java . . . . .	31
5.5.2 main . . . . .	32

<b>6 Decorator</b>	<b>34</b>
6.1 Intento . . . . .	34
6.2 Prestruttura . . . . .	34
6.3 Struttura . . . . .	35
6.4 Implementazione . . . . .	35
6.4.1 Omissione della classe di decoratore astratto. . . . .	35
6.4.2 Mantenere le classi dei componenti leggere. . . . .	35
6.5 Esempio Java . . . . .	36
6.5.1 Shape.java . . . . .	36
6.5.2 Circle.java . . . . .	36
6.5.3 Rectangle.java . . . . .	36
6.5.4 ShapeDecorator.java . . . . .	37
6.5.5 RedShapeDecorator.java . . . . .	37
6.5.6 main . . . . .	37
<b>III Behavioral Patterns</b>	<b>39</b>
<b>7 Command</b>	<b>40</b>
7.1 Intento . . . . .	40
7.2 Prestruttura . . . . .	40
7.3 Struttura . . . . .	41
7.4 Timeline . . . . .	41
7.5 Esempio Java . . . . .	42
7.5.1 Order.java . . . . .	42
7.5.2 BuyStock.java . . . . .	42
7.5.3 SellStock.java . . . . .	43
7.5.4 Stock.java . . . . .	43
7.5.5 Broker.java . . . . .	43
7.5.6 main . . . . .	44
<b>8 Observer</b>	<b>45</b>
8.1 Intento . . . . .	45
8.2 Struttura . . . . .	45
8.3 Timeline . . . . .	46
8.4 Implementazione . . . . .	46
8.4.1 Mappare i soggetti ai loro osservatori. . . . .	46
8.4.2 Osservare più di un soggetto. . . . .	46
8.4.3 Chi chiama Notify e attiva l'aggiornamento? . . . . .	46
8.4.4 Riferimenti penzolanti a soggetti cancellati. . . . .	47
8.4.5 Assicurarsi che lo stato del soggetto sia coerente prima della notifica. . . . .	47
8.4.6 Evitare protocolli di aggiornamento specifici dell'osservatore: i modelli push e pull. . . . .	47
8.4.7 Specificazione esplicita delle modifiche di interesse. . . . .	48
8.4.8 Incapsulamento di semantiche di aggiornamento complesse.	48

8.5 Esempio Java . . . . .	49
8.5.1 ClickListener.java . . . . .	49
8.5.2 main . . . . .	50
<b>9 Strategy . . . . .</b>	<b>51</b>
9.1 Intento . . . . .	51
9.2 Struttura . . . . .	51
9.3 Implementazione . . . . .	51
9.3.1 Definizione delle interfacce Strategy e Context. . . . .	51
9.3.2 Rendere opzionali gli oggetti Strategy. . . . .	52
9.4 Esempio Java . . . . .	52
9.4.1 Strategy.java . . . . .	52
9.4.2 Context.java . . . . .	52
9.4.3 OperationAdd.java . . . . .	53
9.4.4 OperationMult.java . . . . .	53
9.4.5 main . . . . .	53
<b>10 Visitor . . . . .</b>	<b>54</b>
10.1 Intento . . . . .	54
10.2 Struttura . . . . .	55
10.3 Timeline . . . . .	55
10.4 Implementazione . . . . .	56
10.4.1 Doppio invio. . . . .	56
10.4.2 Chi è responsabile dell'attraversamento della struttura dell'oggetto? . . . . .	56
10.5 Esempio Java . . . . .	57
10.5.1 ComputerPart.java . . . . .	57
10.5.2 ComputerPartVisitor.java . . . . .	57
10.5.3 Computer.java . . . . .	57
10.5.4 Keyboard.java . . . . .	58
10.5.5 Monitor.java . . . . .	58
10.5.6 Mouse.java . . . . .	58
10.5.7 ComputerPartDisplayVisitor.java . . . . .	59
10.5.8 main . . . . .	59

# Parte I

## Creational Patterns

# Capitolo 1

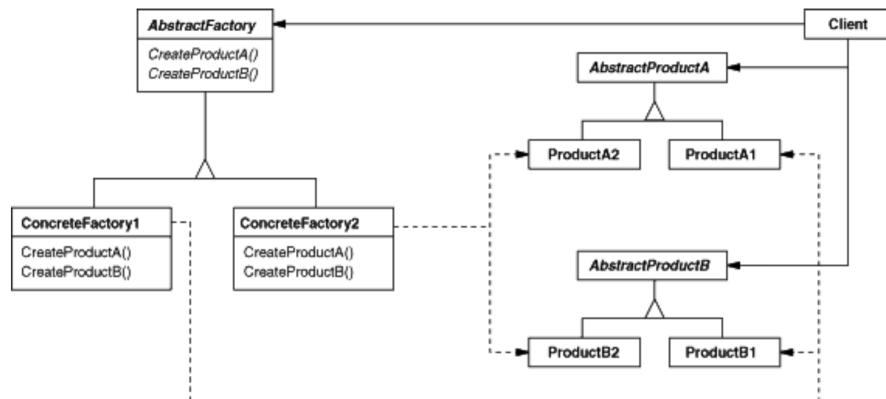
## Abstract Factory

### 1.1 Intento

Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificarne le classi concrete, infatti il costruttore viene chiamato all'interno della factory.

Nel progetto non useremo costruttori ma factory Method o Abstract Factory.

### 1.2 Struttura



### 1.3 Implementazione

#### 1.3.1 Factory come Singleton.

Un'applicazione in genere richiede solo un'istanza di **ConcreteFactory**.

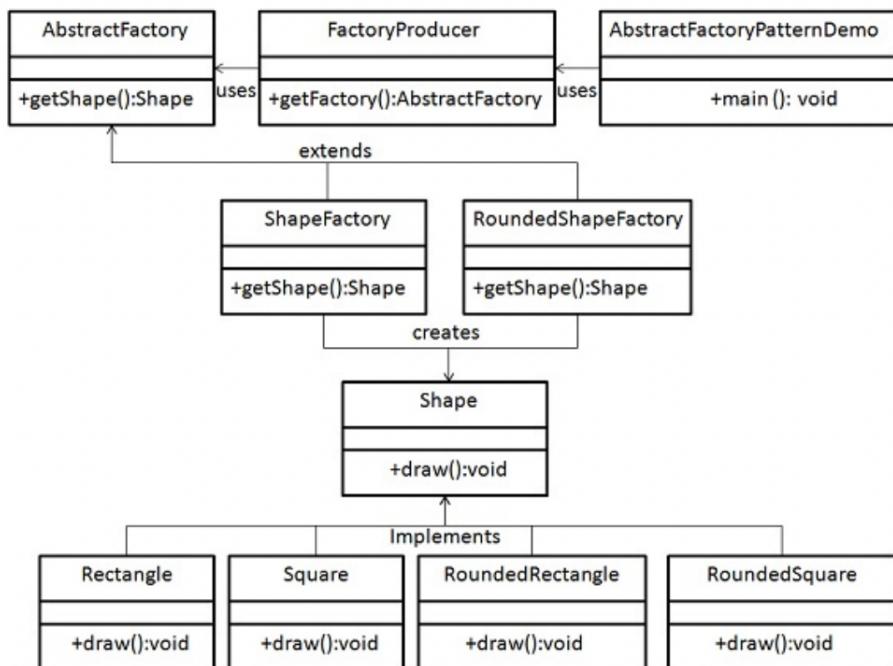
Quindi di solito è meglio implementarlo come Singleton.

### 1.3.2 Creazione dei Product.

AbstractFactory dichiara solo un'interfaccia per la creazione di prodotti. Spetta alle sottoclassi ConcreteProduct crearle effettivamente definendo un Factory Method per ciascun prodotto.

Questa implementazione, quindi, richiede una nuova sottoclasse concreta per ogni famiglia di prodotti, anche se differiscono leggermente (se sono possibili molte famiglie di prodotti utilizzare il modello Prototype).

## 1.4 Esempio Java



### 1.4.1 FactoryProducer.java

```

public class FactoryProducer {
    public static AbstractFactory getFactory(boolean rounded) {
        if (rounded) {
            return new RoundedShapeFactor();
        } else {
            return new ShapeFactory();
        }
    }
}
  
```

#### 1.4.2 AbstractFactory.java

```
public abstract class AbstractFactory {  
    abstract Shape getShape(String shapeType);  
}
```

#### 1.4.3 ShapeFactory.java

```
public class ShapeFactory extends AbstractFactory {  
  
    @Override  
    public Shape getShape(String shapeType) {  
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {  
            return new Rectangle();  
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {  
            return new Square();  
        }  
        return null;  
    }  
}
```

#### 1.4.4 RoundedShapeFactor.java

```
public class RoundedShapeFactor extends AbstractFactory {  
  
    @Override  
    public Shape getShape(String shapeType) {  
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {  
            return new RoundedRectangle();  
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {  
            return new RoundedSquare();  
        }  
        return null;  
    }  
}
```

#### 1.4.5 Shape.java

```
public interface Shape {  
    void draw();  
}
```

#### 1.4.6 Rectangle.java

```
public class Rectangle implements Shape{  
  
    @Override  
    public void draw() {  
        System.out.println("RETTOANGOLO");  
    }  
  
}
```

#### 1.4.7 Square.java

```
public class Square implements Shape{  
  
    @Override  
    public void draw() {  
        System.out.println("QUADRATO");  
    }  
  
}
```

#### 1.4.8 RoundedRectangle.java

```
public class RoundedRectangle implements Shape{  
  
    @Override  
    public void draw() {  
        System.out.println("RETTOANGOLO_ARROTONDATO");  
    }  
  
}
```

#### 1.4.9 RoundedSquare.java

```
public class RoundedSquare implements Shape{  
  
    @Override  
    public void draw() {  
        System.out.println("QUADRATO_ARROTONDATO");  
    }  
  
}
```

#### 1.4.10 main

```
public static void main(String[] args) {
    AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

    Shape s1 = shapeFactory.getShape("rectangle");
    s1.draw();

    Shape s2 = shapeFactory.getShape("square");
    s2.draw();

    shapeFactory = FactoryProducer.getFactory(true);

    Shape s3 = shapeFactory.getShape("rectangle");
    s3.draw();

    Shape s4 = shapeFactory.getShape("square");
    s4.draw();
}
```

# Capitolo 2

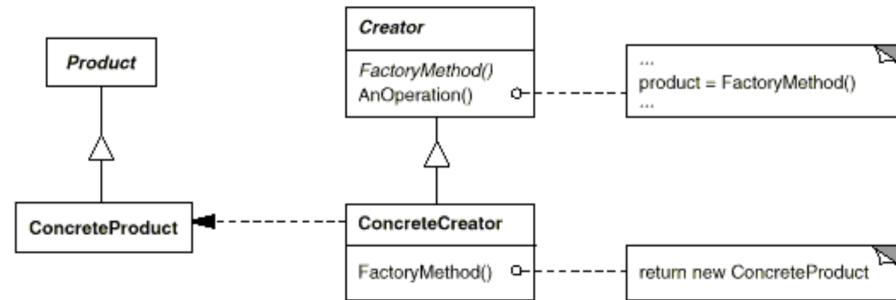
## Factory Method

### 2.1 Intento

Definisce un'interfaccia per la creazione di un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare.

(come l'Abstract Factory ma con un solo Product)

### 2.2 Struttura



### 2.3 Implementazione

#### 2.3.1 Due varietà principali.

Le varianti del modello Factory Method sono

1. il caso in cui la classe Creator è una classe astratta e non fornisce un'implementazione per il Factory Method che dichiara.

Questo caso richiede sottoclassi per definire un'implementazione, perché non esiste un valore predefinito ragionevole. Si aggira il dilemma di dover istanziare classi imprevedibili.

- il caso in cui Creator è una classe concreta e fornisce un'implementazione predefinita per il Factory Method. È anche possibile avere una classe astratta che definisce un'implementazione predefinita, ma è meno comune.

Perciò il Creator concreto utilizza il Factory Method principalmente per la flessibilità. Sta seguendo una regola che dice "Crea oggetti in un'operazione separata in modo che le sottoclassi possano sovrascrivere il modo in cui sono state create". Questa regola garantisce che i progettisti di sottoclassi possano modificare la classe di oggetti istanziati dalla loro classe genitore, se necessario.

### 2.3.2 Factory Method parametrizzati.

Un'altra variazione sul modello consente al Factory Method di creare più tipi di Product. Il Factory Method accetta un parametro che identifica il tipo di oggetto da creare. Tutti gli oggetti creati dal Factory Method condivideranno l'interfaccia del prodotto.

Un metodo di fabbrica parametrizzato ha la seguente forma generale, dove MyProduct e YourProduct sono sottoclassi di Product:

```
class Creator {
    abstract Product Create(ProductId);
};

Product Create (ProductId id) {
    if (id == MINE) {
        return new MyProduct;
    }

    if (id == YOURS) {
        return new YourProduct;
        // repeat for remaining products...
    }

    return 0;
}
```

L'override di un Factory Method parametrizzato consente di estendere o modificare in modo semplice e selettivo i prodotti prodotti da un Creator. Puoi introdurre nuovi identificatori per nuovi tipi di prodotti oppure puoi associare identificatori esistenti a prodotti diversi. Ad esempio, una sottoclasse MyCreator potrebbe scambiare MyProduct e YourProduct e supportare una nuova sottoclasse TheirProduct:

```
Product Create (ProductId id) {
    if (id == YOURS) {
        return new MyProduct;
    }

    return 0;
}
```

```

if (id == MINE) {
    return new YourProduct;
    // N.B.: switched YOURS and MINE
}

if (id == THEIRS) {
    return new TheirProduct;
}

return Create(id); // called if all others fail
}

```

Nota che l'ultima cosa che fa questa operazione è chiamare Create sulla classe genitore. Questo perché Create gestisce solo My, Your e Their in modo diverso rispetto alla classe genitore. Non è interessato ad altre classi. Quindi MyCreator estende i tipi di prodotti creati e rinvia la responsabilità della creazione di tutti i prodotti tranne alcuni al suo genitore.

### 2.3.3 Utilizzo di modelli per evitare la sottoclasse.

Come abbiamo detto, un altro potenziale problema con i Factory Method è che potrebbero costringerti a sottoclassi solo per creare gli oggetti Product appropriati. Un altro modo, in C++, è fornire una sottoclasse di template di Creator parametrizzata dalla classe Product:

```

class Creator {
    abstract Product CreateProduct() = 0;
};

templateclass <class TheProduct>

class StandardCreator: public Creator {
    public:
        virtual Product* CreateProduct();
};

template <class TheProduct>

Product* StandardCreator<TheProduct>::CreateProduct () {
return new TheProduct;
}

```

Con questo modello, il client fornisce solo la classe di prodotto, non è richiesta alcuna sottoclasse di Creator.

```
class MyProduct : public Product {
```

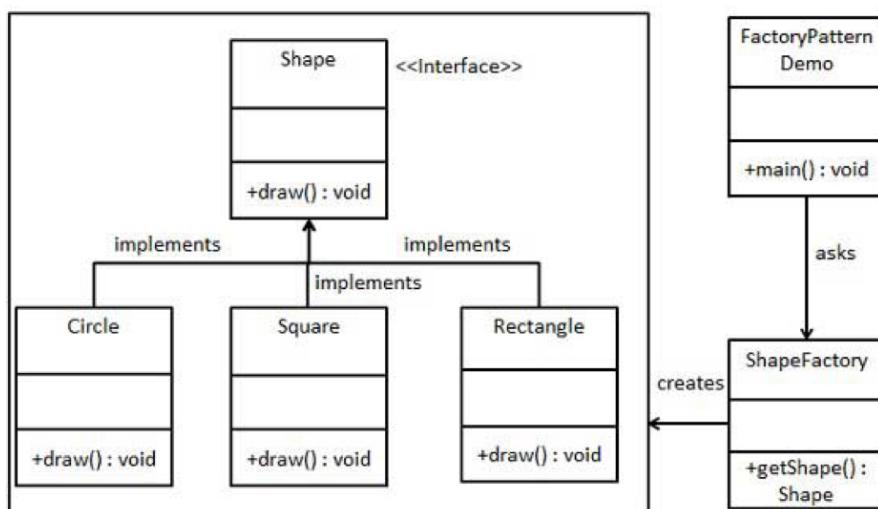
```

public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;

```

## 2.4 Esempio Java



(Il costruttore va nascosto per esempio con un if)

### 2.4.1 Circle.java

```

public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("CERCHIO");
    }
}

```

### 2.4.2 Rectangle.java

```

public class Rectangle implements Shape {
}

```

```

@Override
public void draw() {
    System.out.println("RECTANGLE");
}

}

```

#### 2.4.3 Shape.java

```

public interface Shape {
    void draw();
}

```

#### 2.4.4 ShapeFactory.java

```

public class ShapeFactory {
    public Shape getShape(String shapeType){
        if (shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        }
        return null;
    }
}

```

#### 2.4.5 main

```

public static void main(String[] args) {
    ShapeFactory sf1 = new ShapeFactory();

    Shape s1 = sf1.getShape("circle");
    s1.draw();

    Shape s2 = sf1.getShape("rectangle");
    s2.draw();
}

```

# Capitolo 3

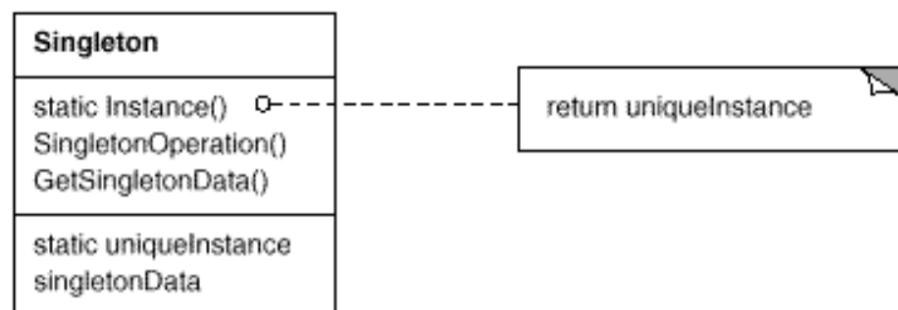
## Singleton

### 3.1 Intento

Utile per assicurarsi che una classe abbia solo un'istanza e fornisci un punto di accesso globale ad essa.

(nel progetto utile per l'interfaccia di accesso al db, cioè nel dbInstance, e nei DAO)

### 3.2 Struttura



### 3.3 Implementazione

#### 3.3.1 Garantire un'istanza univoca.

Il Singleton rende l'unica istanza un'istanza normale di una classe, ma quella classe è scritta in modo che possa essere creata solo un'istanza.

Un modo comune per eseguire questa operazione consiste nel nascondere l'operazione che crea l'istanza dietro una funzione statica, o un metodo di classe,

che garantisce la creazione di una sola istanza. Questo approccio garantisce che un singleton venga creato e inizializzato prima del suo primo utilizzo.

È possibile definire l'operazione di classe con una funzione statica Istanza della classe Singleton. Singleton definisce anche una variabile statica "instance" che contiene un puntatore alla sua istanza univoca. La classe Singleton è dichiarata come:

```
public class SingleObject {
    private static SingleObject instance = new SingleObject();

    private SingleObject (){

    }

    public static SingleObject getInstance() {
        return instance;
    }
}
```

I client accedono al Singleton esclusivamente tramite la funzione dell'istanza.

Notare che il costruttore è privato. Un client che tenta di istanziare direttamente Singleton riceverà un errore in fase di compilazione. Ciò garantisce che solo un'istanza possa essere creata.

### 3.3.2 Sottoclasse della classe Singleton.

Il problema principale non è tanto definire la sottoclasse, ma installare la sua istanza univoca in modo che i client possano usarla. In sostanza, la variabile che fa riferimento all'istanza singleton deve essere inizializzata con un'istanza della sottoclasse.

La tecnica più semplice consiste nel determinare quale Singleton si desidera utilizzare nell'operazione dell'istanza di Singleton.

```
MazeFactory Instance () {
    if (_instance == 0) {
        const char mazeStyle = getenv ("MAZESTYLE");

        if (mazeStyle.equals ("bombed")) {
            _instance = new BombedMazeFactory;
        } else if (mazeStyle.equals ("enchanted")) {
            _instance = new EnchantedMazeFactory;
            // ... other possible subclasses
        } else {
            // default _instance = new MazeFactory;
        }
    }
}
```

```

    return _instance;
}

```

Un altro modo per scegliere la sottoclasse di Singleton consiste nell'estrarrre l'implementazione di Instance dalla classe genitore ed inserirla nella sottoclasse. Ciò consente di decidere la classe di Singleton al momento del collegamento ma la tiene nascosta ai client del singleton. Ma non è un approccio abbastanza flessibile.

Un approccio più flessibile utilizza un registro di Singleton. Invece di fare in modo che Instance definisca l'insieme di possibili classi Singleton, le classi Singleton possono registrare la loro istanza Singleton per nome in un registro noto.

Il registro esegue il mapping tra nomi di stringhe e Singleton. Quando l'istanza ha bisogno di un Singleton, consulta il registro, chiedendo il Singleton per nome. Il registro cerca il Singleton corrispondente, se esiste, e lo restituisce.

Questo approccio libera l'istanza dal conoscere tutte le possibili classi o istanze Singleton. Tutto ciò che richiede è un'interfaccia comune per tutte le classi Singleton che includa operazioni per il registro:

```

class Singleton {
    static void Register(const char name, Singleton);
    static Singleton Instance();

    protected static Singleton Lookup(const char name);

    private static Singleton _instance;
    private static List<NameSingletonPair> _registry;
}

```

Register registra l'istanza Singleton con il nome specificato. Per mantenere il registro semplice, faremo in modo che memorizzi un elenco di oggetti NameSingletonPair. Ogni NameSingletonPair associa un nome a un Singleton e l'operazione di ricerca trova un Singleton dato il suo nome.

```

Singleton Instance () {
    if (_instance == 0) {
        const char singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}

```

Dove si registrano le classi Singleton? Una possibilità è nel loro costruttore. Ad esempio, una sottoclasse MySingleton potrebbe fare quanto segue:

```

MySingleton () {

```

```

    // ...
    Register("MySingleton", this);
}

```

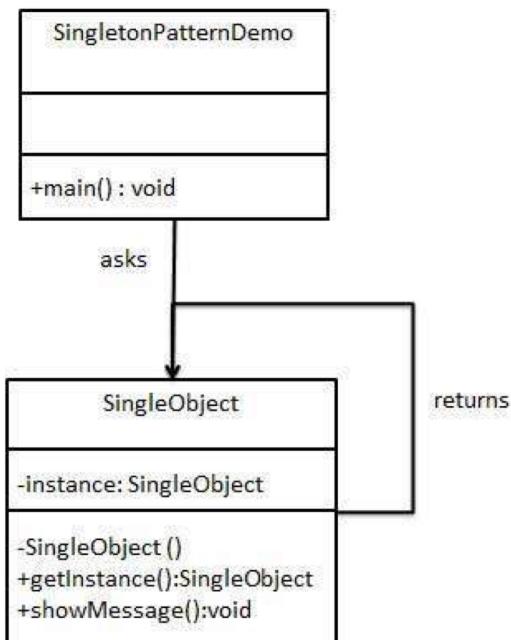
Ovviamente, il costruttore non verrà chiamato a meno che qualcuno non istanzia la classe, il che fa eco al problema che il pattern Singleton sta cercando di risolvere! Possiamo aggirare questo problema definendo un'istanza statica di MySingleton:

```
static MySingleton theSingleton;
```

La classe Singleton non è più responsabile della creazione del Singleton, infatti, la sua responsabilità primaria è rendere accessibile nel sistema l'oggetto di scelta Singleton.

L'approccio dell'oggetto statico ha ancora un potenziale svantaggio, ovvero che devono essere create istanze di tutte le possibili sottoclassi Singleton, altrimenti non verranno registrate.

### 3.4 Esempio Java



#### 3.4.1 Singleton.java

```
public class Singleton {
    private static Singleton instance = new Singleton();
}
```

```
private Singleton (){

}

public static Singleton getInstance() {
    return instance;
}

public void showMessage( String message) {
    System.out.println(message);
}
}
```

### 3.4.2 main

```
public static void main( String [] args) {
    Singleton so = Singleton.getInstance();

    so.showMessage( "puppe");
}
```

## Parte II

# Structural Patterns

# Capitolo 4

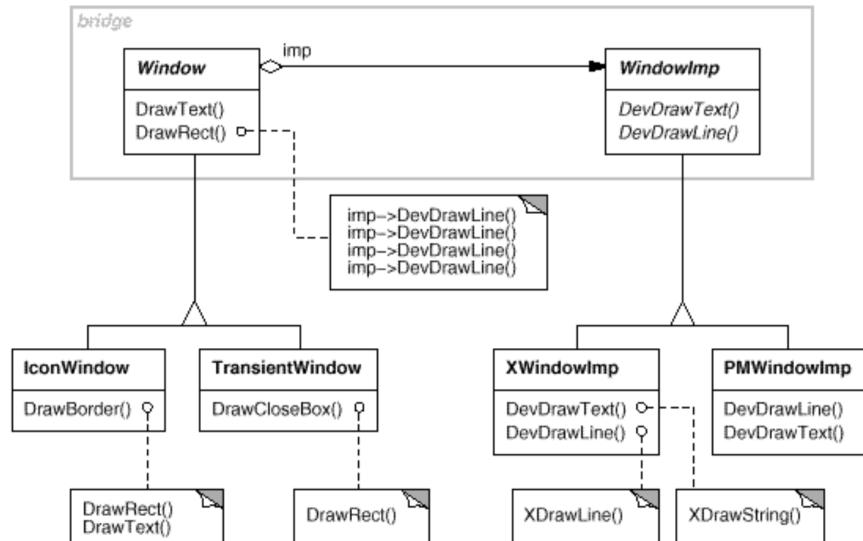
## Bridge

### 4.1 Intento

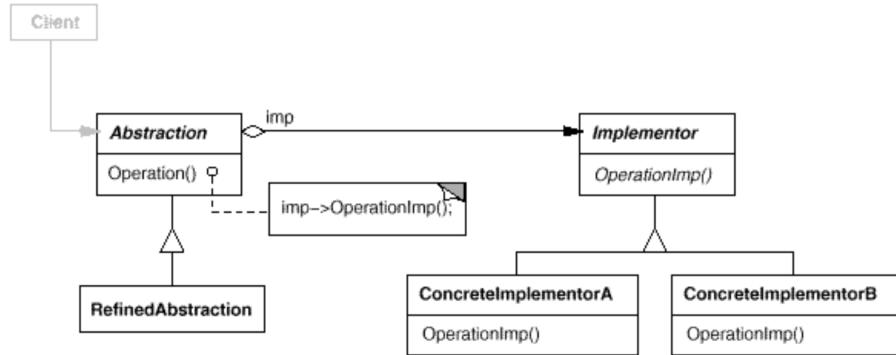
Disaccoppia un'astrazione dalla sua implementazione in modo che i due possano variare indipendentemente. Per esempio nel caso di utilizzo di librerie esterne.

(nel progetto lo usiamo per isolare le librerie per inviare email e creare pdf)

### 4.2 Prestruttura



## 4.3 Struttura



## 4.4 Implementazione

### 4.4.1 Solo un implementazione.

Se abbiamo solo un'implementazione, non è necessario creare una classe Implementor astratta. In tal caso ci sarà una relazione uno a uno tra Abstraction e Implementor.

Questa separazione è ancora utile quando una modifica nell'implementazione di una classe non deve influenzare i suoi client esistenti.

### 4.4.2 Creazione dell'oggetto Implementor corretto.

Come, quando e dove decidi quale classe Implementor istanziare quando ce n'è più di una?

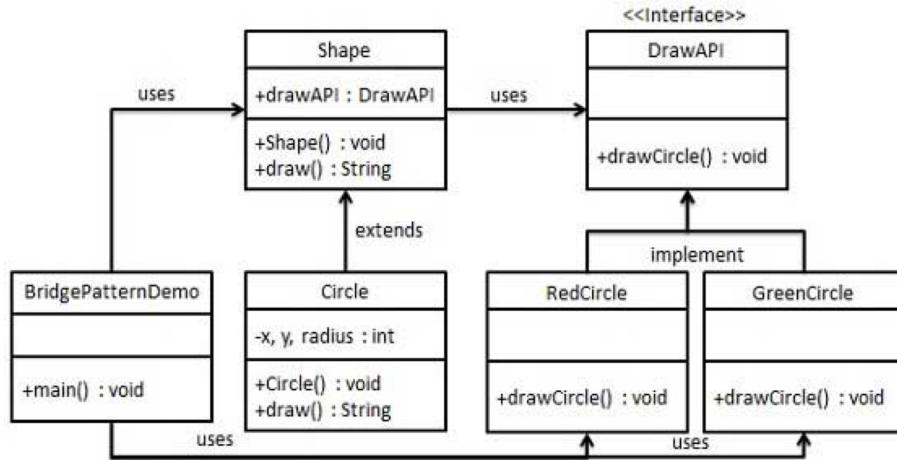
Un'implementazione di elenchi collegati può essere utilizzata per piccole raccolte e una tabella hash per quelle più grandi.

Un altro approccio consiste nello scegliere inizialmente un'implementazione predefinita e modificarla in seguito in base all'utilizzo.

Oppure si può delegare del tutto la decisione ad un altro oggetto. Nell'esempio Window / WindowImp, possiamo introdurre un oggetto Factory il cui unico compito è incapsulare le specifiche della piattaforma.

La factory sa che tipo di oggetto WindowImp creare per la piattaforma in uso; a Window gli chiede semplicemente un WindowImp e restituisce il tipo giusto.

## 4.5 Esempio Java



### 4.5.1 DrawImplementation.java

```
public interface DrawImplementation {
    public void drawCircle(int x, int y, int radius);
}
```

### 4.5.2 Shape.java

```
public abstract class Shape {
    protected DrawImplementation drawImpl;

    protected Shape(DrawImplementation drawAPI) {
        this.drawImpl = drawAPI;
    }

    public abstract void draw();
}
```

### 4.5.3 Circle.java

```
public class Circle extends Shape{
    private int x, y, radius;

    public Circle(DrawImplementation drawImpl, int x, int y, int radius) {
        super(drawImpl);
        this.x = x;
    }
}
```

```

        this.y = y;
        this.radius = radius;
    }

    @Override
    public void draw() {
        // specifico dell'ambiente windowing usato a runtime
        drawImpl.drawCircle(x, y, radius);
    }
}

```

#### 4.5.4 RedCircle.java

```

public class RedCircle implements DrawImplementation{

    @Override
    public void drawCircle(int x, int y, int radius) {
        System.out.println("cerchio_rosso_di:" + radius + x + y);
    }
}

```

#### 4.5.5 GreenCircle.java

```

public class GreenCircle implements DrawImplementation{

    @Override
    public void drawCircle(int x, int y, int radius) {
        System.out.println("cerchio_verde_di:" + radius + x + y);
    }
}

```

#### 4.5.6 main

```

public static void main(String[] args) {
    Shape redC = new Circle(new RedCircle(), 100, 100, 10);
    Shape greC = new Circle(new GreenCircle(), 100, 100, 10);

    redC.draw();
    greC.draw();
}

```

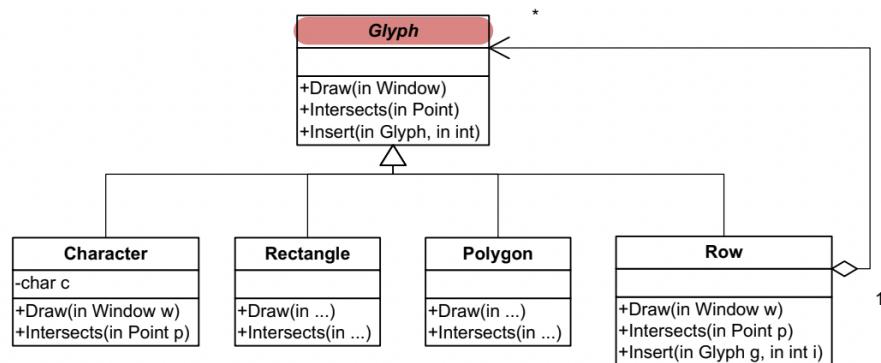
# Capitolo 5

## Composite

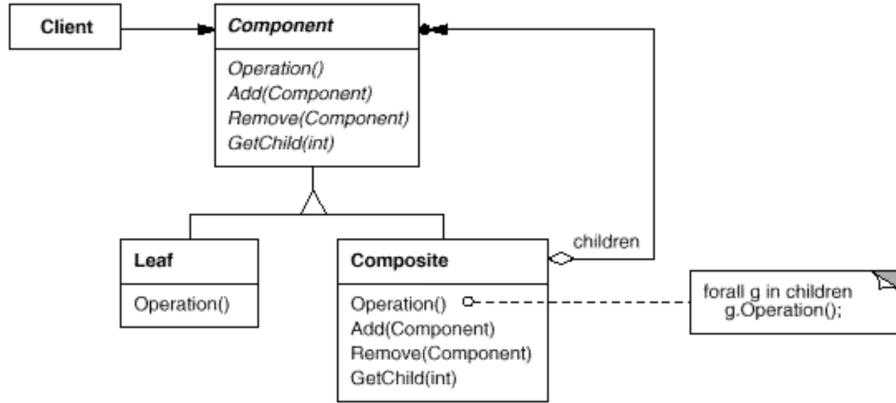
### 5.1 Intento

Comporre oggetti in strutture ad albero per rappresentare gerarchie parziali. Il Composite consente ai clienti di trattare i singoli oggetti e le composizioni di oggetti in modo uniforme.

### 5.2 Prestruttura



### 5.3 Struttura



### 5.4 Implementazione

#### 5.4.1 Condivisione di Component.

È spesso utile condividere i Component, ad esempio, per ridurre i requisiti di archiviazione. Ma quando un componente non può avere più di un genitore, la condivisione di Component diventa difficile.

Una possibile soluzione è che i figli memorizzino più genitori. (Il modello Flyweight (218) mostra come rielaborare un design per evitare di memorizzare i genitori del tutto)

#### 5.4.2 Massimizzare l'interfaccia del componente.

Uno degli obiettivi del pattern Composite è rendere i clienti ignari delle classi che stanno utilizzando.

Per raggiungere questo obiettivo, la classe Component definisce le operazioni più comuni, fornendo quindi implementazioni predefinite per le sottoclassi Leaf e Composite, che le sovrascriveranno.

In che modo Component può fornire un'implementazione predefinita a Leaf? Le classi Leaf possono utilizzare l'implementazione predefinita, ma le classi composite la reimplementeranno per restituire i propri figli.

#### 5.4.3 Dichiarazione delle operazioni di gestione del figlio.

Meglio dichiarare le operazioni Aggiungi e Rimuovi nel Component e renderle significative per le classi Leaf, o dichiararle e definirle solo in Composite e nelle sue sottoclassi?

La decisione prevede un compromesso tra sicurezza e trasparenza:

1. La definizione alla radice della gerarchia offre trasparenza, poiché si trattano i Component in modo uniforme. Ne viene meno la sicurezza, dato che i clienti possono provare ad aggiungere e rimuovere oggetti dalle Leaf. L'unico modo per fornire trasparenza è definire le operazioni di aggiunta e rimozione predefinite in Component.

Questo crea un problema: non c'è modo di implementare Add di Component senza introdurre la possibilità che fallisca o che permetta un tentativo di aggiungere qualcosa a una Leaf creando un bug.

Di solito è meglio far fallire Aggiungi e Rimuovi per impostazione predefinita (magari sollevando un'eccezione) se il Component non può avere figli o se l'argomento di Rimuovi non è figlio di Component.

2. Definire la gestione dei figli nella classe Composite ti dà sicurezza, perché qualsiasi tentativo di aggiungere o rimuovere oggetti dalle Leaf verrà catturato. Se perdi in trasparenza, perché Leaf e Composite hanno interfacce diverse.

Se opti per la sicurezza, a volte potresti perdere le informazioni sul tipo e dover convertire un Componente in un Composite.

Senza ricorrere a un cast non sicuro per i tipi possiamo dichiarare un'operazione nella classe Component:

- in Componente:

```
Composite GetComposite() {
    return 0;
}
```

- in Composite:

```
Composite GetComposite() {
    return this;
}
```

GetComposite ti consente di interrogare un componente per vedere se è un composto.

```
Composite aComposite = new Composite();
Leaf aLeaf = new Leaf();

Component aComponent;
Composite test;

aComponent = aComposite;
if (test = aComponent.GetComposite()) {
    test.Add(new Leaf);
```

```

    }

aComponent = aLeaf;
if (test = aComponent.GetComposite()) {
    test.Add(new Leaf); // not add leaf
}

```

#### **5.4.4 Il Component dovrebbe implementare un elenco di Component?**

Definire l'insieme dei figli come una variabile di istanza nella classe Component comporta una penalità di spazio per ogni Leaf (solo se nella struttura ci sono pochi figli).

#### **5.4.5 Memorizzazione nella cache per migliorare le prestazioni.**

Il Composit può memorizzare nella cache i risultati effettivi o solo le informazioni che gli consentono di cortocircuitare l'attraversamento o la ricerca.

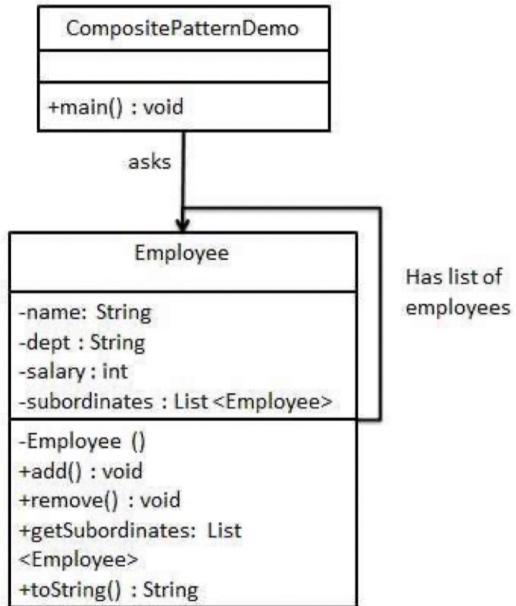
Se stai usando la memorizzazione nella cache, devi definire un'interfaccia per dire ai Composite che le loro cache non sono valide dato che si potrebbe usare un Draw quando un figlio non è visibile, non avendo quindi nessun apparente modifica.

#### **5.4.6 Chi dovrebbe eliminare i Component?**

Nelle lingue senza raccolta di rifiuti, di solito è meglio rendere un Composit responsabile dell'eliminazione dei suoi figli quando viene distrutto.

Un'eccezione è quando gli oggetti Leaf sono immutabili e quindi possono essere condivisi.

## 5.5 Esempio Java



### 5.5.1 Employee.java

```
public class Employee {  
    private String name;  
    private String dept;  
    private int salary;  
    private List<Employee> subordinates; // Design pattern Composite  
  
    public Employee( String name, String dept, int salary){  
        this.name = name;  
        this.dept = dept;  
        this.salary = salary;  
        this.subordinates = new ArrayList<Employee>();  
    }  
  
    public void addSubordinates( Employee employee){  
        subordinates.add(employee);  
    }  
  
    public void removeSubordinates( Employee employee){  
        subordinates.remove(employee);  
    }  
}
```

```

public List<Employee> getSubordinates(){
    return subordinates;
}

@Override
public String toString(){
    return ("Composite.Employee[ Name:" + name +
        ", dept:" + dept +
        ", salary:" + salary + "]");
}
}

```

### 5.5.2 main

```

public class CompositePatternDemo {
    public static void main(String[] args) {
        Employee ceo = new Employee("Adriana", "Main", 30000);

        Employee headMark = new Employee("Roberto", "Marketing", 20000);
        Employee clerk1 = new Employee("Laura", "Marketing", 10000);
        Employee clerk2 = new Employee("Alessandro", "Marketing", 10000);

        Employee headDev = new Employee("Luigi", "Development", 20000);
        Employee dev1 = new Employee("Ilaria", "Development", 10000);
        Employee dev2 = new Employee("Giovanni", "Development", 10000);

        ceo.addSubordinates(headMark);
        ceo.addSubordinates(headDev);

        headMark.addSubordinates(clerk1);
        headMark.addSubordinates(clerk2);

        headDev.addSubordinates(dev1);
        headDev.addSubordinates(dev2);

        System.out.println(ceo);

        List<Employee> ceoSubordinates = ceo.getSubordinates();
        Iterator<Employee> ceoSubIter = ceoSubordinates.iterator();

        while (ceoSubIter.hasNext()) {
            Employee e = ceoSubIter.next();
            System.out.println(e);
        }
    }
}

```

```
List<Employee> headSubordinates = e.getSubordinates();
Iterator<Employee> headSubIter = headSubordinates.iterator();

while (headSubIter.hasNext()) {
    System.out.println(headSubIter.next());
}

}
```

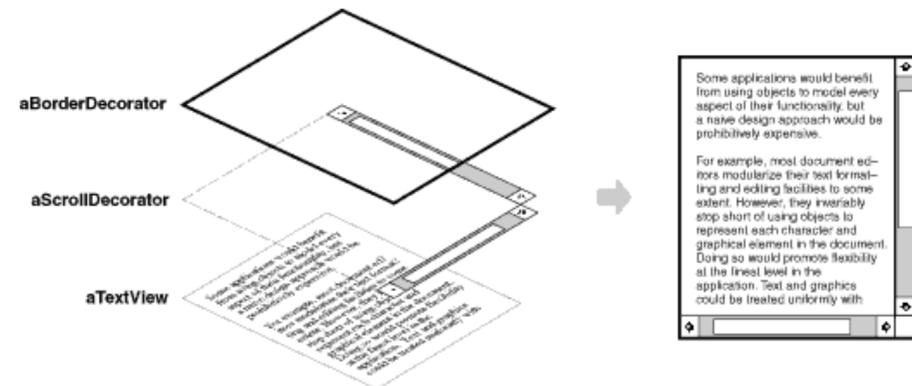
# Capitolo 6

## Decorator

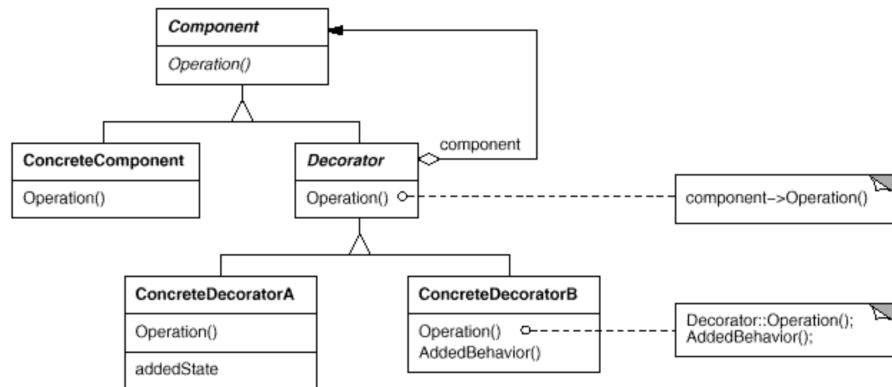
### 6.1 Intento

Associare responsabilità aggiuntive a un oggetto in modo dinamico. I decorator forniscono un'alternativa flessibile all'ereditarietà

### 6.2 Prestruttura



## 6.3 Struttura



## 6.4 Implementazione

### 6.4.1 Omissione della classe di decoratore astratto.

Non c'è bisogno di definire una classe di decoratore astratto quando è necessario aggiungere solo una responsabilità. Questo è spesso il caso quando hai a che fare con una gerarchia di classi esistente piuttosto che progettarne una nuova.

Puoi unire la responsabilità del decoratore per inoltrare le richieste al componente in **ConcreteDecorator**.

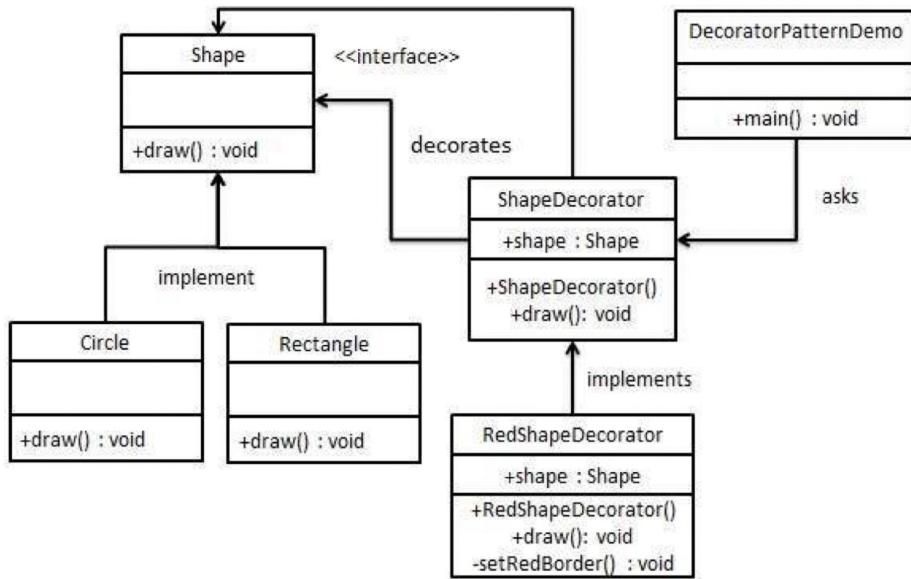
### 6.4.2 Mantenere le classi dei componenti leggere.

Per garantire un'interfaccia conforme, componenti e decoratori devono discendere da una classe **Component** comune.

È importante:

- mantenere leggera questa classe comune, facendola concentrare sull'associare ad un'interfaccia, non a memorizzare dati.
- la rappresentazione dei dati dovrebbe essere rinviata alle sottoclassi; altrimenti la complessità potrebbe rendere i decoratori troppo pesanti da usare in quantità.

## 6.5 Esempio Java



### 6.5.1 Shape.java

```
public interface Shape {  
    void draw();  
}
```

### 6.5.2 Circle.java

```
public class Circle implements Shape{  
  
    @Override  
    public void draw() {  
        System.out.println("CERCHIO");  
    }  
  
}
```

### 6.5.3 Rectangle.java

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
  
    }
```

```

        System.out.println("RETTOANGOLO");
    }
}
```

#### 6.5.4 ShapeDecorator.java

```

public abstract class ShapeDecorator implements Shape{
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    @Override
    public void draw() {
        decoratedShape.draw();
    }
}
```

#### 6.5.5 RedShapeDecorator.java

```

public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape) {
        System.out.println("BORDO_ROSSO");
    }
}
```

#### 6.5.6 main

```

public static void main(String[] args) {
    Shape circle = new Circle();
```

```
Shape redCircle = new RedShapeDecorator(new Circle());
Shape redRectangle = new RedShapeDecorator(new Rectangle());

System.out.println("Cerchio_normale:");
circle.draw();

System.out.println("Cerchio_rosso:");
redCircle.draw();
System.out.println("Rettangolo_rosso:");
redRectangle.draw();
}
```

# Parte III

## Behavioral Patterns

# Capitolo 7

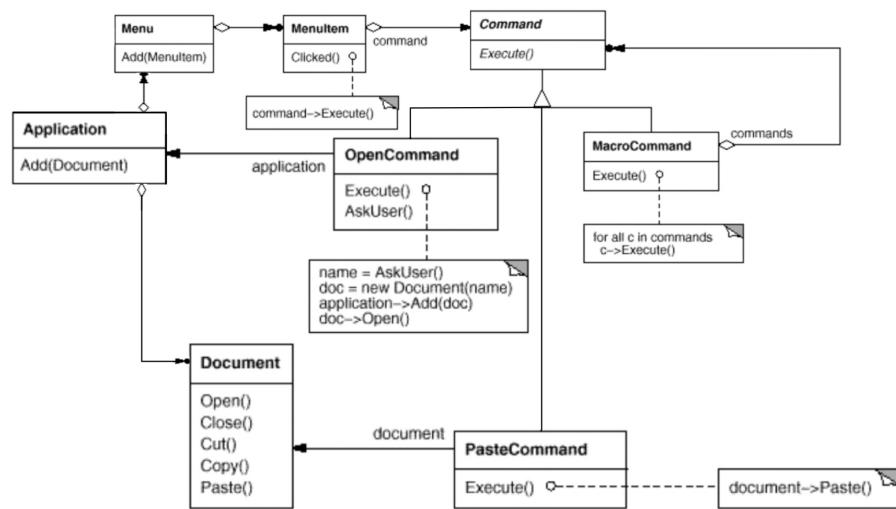
## Command

### 7.1 Intento

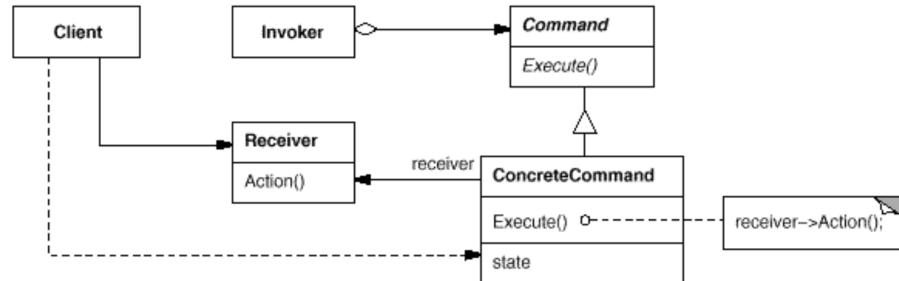
Incapsula una richiesta come oggetto, consentendo in tal modo di parametrizzare i client con richieste diverse, richieste di coda o di registro e operazioni non supportabili.

(nel progetto da usare quando si utilizzano delle query)

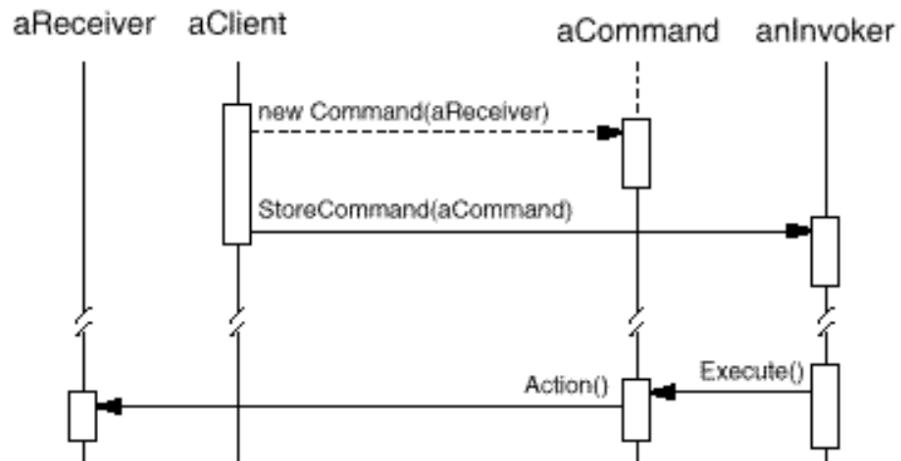
### 7.2 Prestruttura



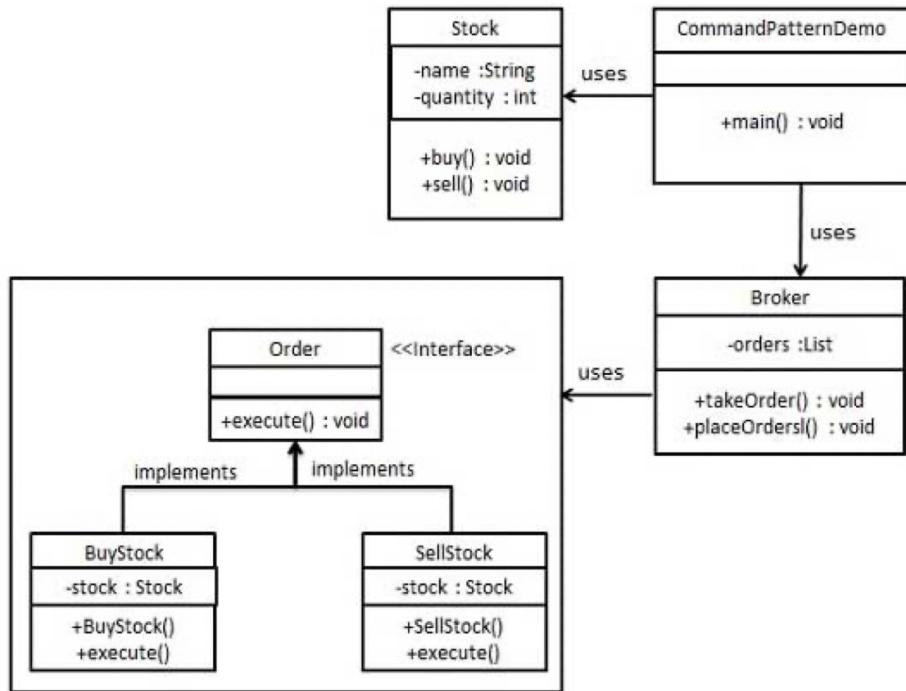
### 7.3 Struttura



### 7.4 Timeline



## 7.5 Esempio Java



### 7.5.1 Order.java

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

### 7.5.2 BuyStock.java

```
public class BuyStock implements Order{  
    private Stock stock;  
  
    public BuyStock(Stock stock) {  
        this.stock = stock;  
    }  
  
    @Override  
    public void execute() {  
        stock.buy();  
    }  
}
```

```
}
```

### 7.5.3 SellStock.java

```
public class SellStock implements Order{
    private Stock stock;

    public SellStock(Stock stock) {
        this.stock = stock;
    }

    @Override
    public void execute() {
        stock.sell();
    }

}
```

### 7.5.4 Stock.java

```
public class Stock {
    private String name;
    private int quantity;

    public Stock(String name, int quantity) {
        this.name = name;
        this.quantity = quantity;
    }

    public void buy(){
        System.out.println("comprati" + name + "da" + quantity);
    }

    public void sell(){
        System.out.println("venduti" + name + "da" + quantity);
    }
}
```

### 7.5.5 Broker.java

```
public class Broker {
    private List<Order> orderList = new ArrayList<>();
```

```

public void takeOrder(Order order) {
    orderList.add(order);
}

public void placeOrder() {
    for (Order order : orderList) {
        order.execute();
    }
    orderList.clear();
}
}

```

#### 7.5.6 main

```

public static void main(String[] args) {
    Stock stock = new Stock("prodotti", 10);

    BuyStock buyStock = new BuyStock(stock);
    SellStock sellStock = new SellStock(stock);

    Broker broker = new Broker();
    broker.takeOrder(buyStock);
    broker.takeOrder(sellStock);

    broker.placeOrder();
}

```

# Capitolo 8

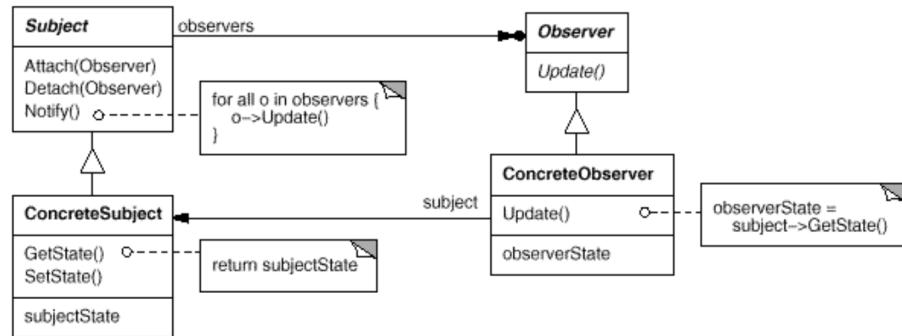
## Observer

### 8.1 Intento

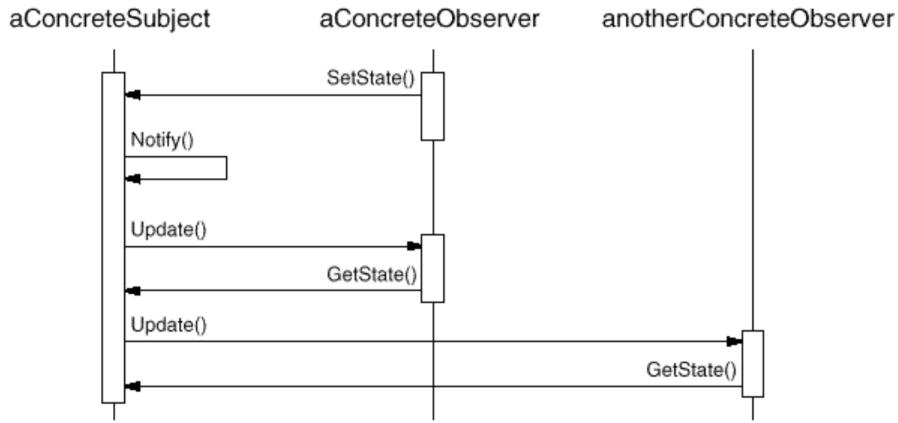
Definisce un'interfaccia per la creazione di un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare.

(come l'Abstract Factory ma con un solo Product)

### 8.2 Struttura



## 8.3 Timeline



## 8.4 Implementazione

### 8.4.1 Mappare i soggetti ai loro osservatori.

Tale archiviazione può essere troppo costosa quando ci sono molti soggetti e pochi osservatori.

Si può quindi utilizzare una ricerca associativa (ad esempio una tabella hash) per mantenere la mappatura soggetto-osservatore. Quindi un soggetto senza osservatori non incorre in spese di archiviazione.

### 8.4.2 Osservare più di un soggetto.

Potrebbe avere senso in alcune situazioni che un osservatore dipenda da più di un soggetto.

In questi casi è necessario estendere l'interfaccia di aggiornamento per far sapere all'osservatore quale soggetto sta inviando la notifica (passando se stesso come parametro nell'operazione di aggiornamento, facendo sapere all'osservatore quale soggetto esaminare)

### 8.4.3 Chi chiama Notify e attiva l'aggiornamento?

- Avere operazioni di impostazione dello stato sulla chiamata del soggetto Notifica dopo aver modificato lo stato del soggetto.
  - **Vantaggio:** i clienti non devono ricordarsi di chiamare Notify sull'argomento.
  - **Svantaggio:** operazioni consecutive causano diversi aggiornamenti consecutivi. Potrebbe essere inefficiente.

- Rendi i clienti responsabili di chiamare Notify al momento giusto.
  - **Vantaggio:** il cliente ha la gestione degli aggiornamenti, evitando così inutili aggiornamenti.
  - **Svantaggio:** il cliente ha la responsabilità aggiuntiva di attivare l'aggiornamento, rendendo più probabili gli errori.

#### 8.4.4 Riferimenti penzolanti a soggetti cancellati.

Un modo per evitare riferimenti penzolanti è fare in modo che il soggetto informi i suoi osservatori quando viene cancellato in modo che possano reimpostare il loro riferimento ad esso.

La semplice eliminazione degli osservatori non è un'opzione, perché altri oggetti potrebbero fare riferimento a loro o potrebbero osservare anche altri soggetti.

#### 8.4.5 Assicurarsi che lo stato del soggetto sia coerente prima della notifica.

È importante perché il soggetto è interrogato per il suo stato attuale nel corso dell'aggiornamento del proprio stato. È facile da violare quando le operazioni della sottoclassificazione Oggetto chiamano operazioni ereditate.

È possibile evitarlo inviando notifiche dai Model in classi di soggetti astratte, facendo in modo che Notifica sia l'ultima operazione nel Model, che assicurerà che l'oggetto sia autoconsistente quando le sottoclassificazioni sovrascrivono le operazioni del soggetto.

#### 8.4.6 Evitare protocolli di aggiornamento specifici dell'osservatore: i modelli push e pull.

Le implementazioni del modello Observer spesso fanno trasmettere al soggetto informazioni aggiuntive sul cambiamento. Il soggetto passa queste informazioni come argomento ad Aggiorna. La quantità di informazioni può variare notevolmente.

- **Modello PUSH:** presuppone che i soggetti sappiano qualcosa sui bisogni dei loro osservatori.

I soggetti forniscono agli osservatori informazioni dettagliate sul cambiamento. (osservatori meno riutilizzabili)

- **Modello PULL:** enfatizza l'ignoranza del soggetto.

Il soggetto invia la minima notifica e gli osservatori chiedono esplicitamente i dettagli in seguito. (può essere inefficiente)

#### **8.4.7 Specificazione esplicita delle modifiche di interesse.**

È possibile migliorare l'efficienza degli aggiornamenti consentendo la registrazione degli osservatori solo per eventi di interesse specifici. Per fare ciò sono attaccati ai loro soggetti usando

```
void Attach( Osservatore , interesse );
```

Al momento della notifica, il soggetto fornisce l'aspetto modificato ai suoi osservatori. Per esempio:

```
void Update( Subject , interesse );
```

#### **8.4.8 Incapsulamento di semantiche di aggiornamento complesse.**

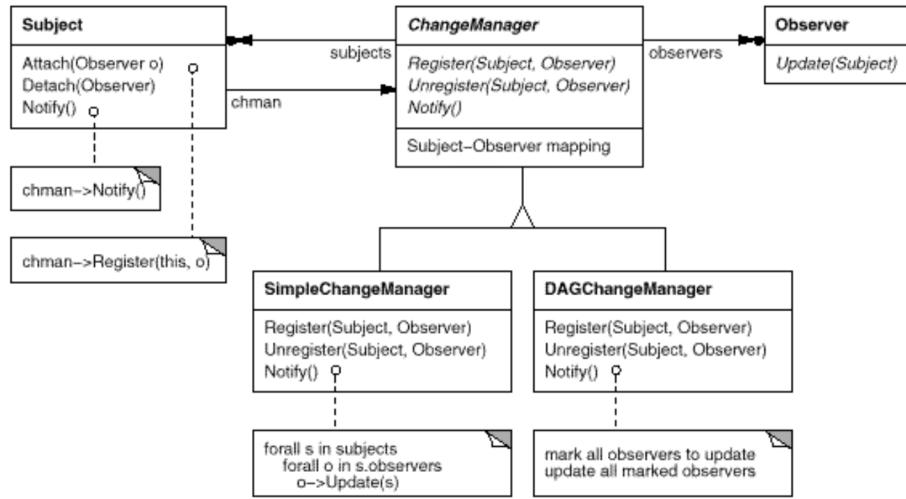
- **ChangeManager:** il suo scopo è ridurre al minimo il lavoro necessario per far riflettere gli osservatori su un cambiamento nel loro soggetto.

Ad esempio, se un'operazione comporta modifiche a più soggetti, potresti dover assicurarti che i loro osservatori vengano avvisati dopo che tutti i soggetti sono stati modificati per evitare di avvisare gli osservatori più di una volta.

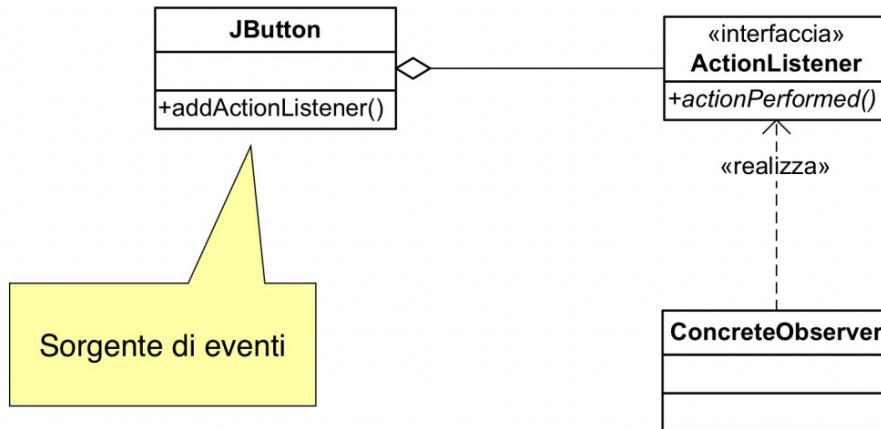
ChangeManager ha tre responsabilità:

1. Mappa un soggetto ai suoi osservatori e fornisce un'interfaccia per mantenere questa mappatura. Ciò elimina la necessità per i soggetti di mantenere riferimenti ai propri osservatori e viceversa.
2. Definisce una particolare strategia di aggiornamento.
3. Aggiorna tutti gli osservatori dipendenti su richiesta di un soggetto.

Il DAGChangeManager assicura che l'osservatore riceva un solo aggiornamento. SimpleChangeManager va bene quando più aggiornamenti non sono un problema (il modello Singleton sarebbe utile qui).



## 8.5 Esempio Java



### 8.5.1 ClickListener.java

```

public class ClickListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        JButton bc = (JButton) e.getSource();
        if (bc.getText().equals("Cliccato")) {
            bc.setText("Decliccato");
        } else {
    }
}

```

```

        bc.setText("Cliccato");
    }
}
}
```

### 8.5.2 main

```

public static void main(String [] args) {
    /**
     * Creazione del frame e del pulsante
     */
    JFrame f = new JFrame();
    JButton b = new JButton("Decliccate");

    /**
     * Si aggiunge al frame il pulsante
     */
    f.add(b);

    /**
     * Creazione di un Ascoltatore di Azioni
     * per andare a notificare i cambiamenti alla view
     */
    ActionListener al = new ClickListener();

    /**
     * !!! PARTE IMPORTANTE PER L'OBSERVER !!!
     * Si aggiunge al pulsante l'ActionListener
     * in modo che i due siano collegati
     */
    b.addActionListener(al);

    /**
     * Setting del frame
     */
    f.setSize(300, 200);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
}
```

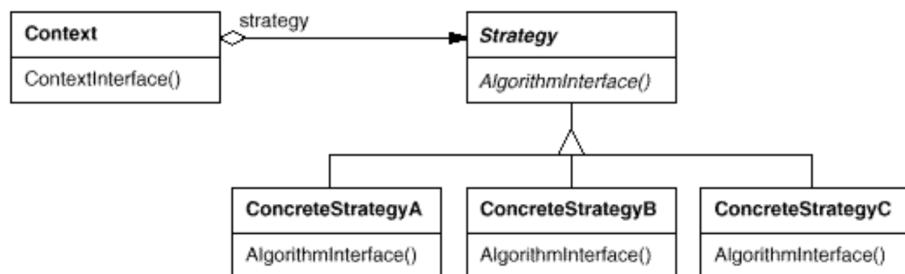
# Capitolo 9

## Strategy

### 9.1 Intento

Ogni volta che si ha un algoritmo che ammette varianti si deve usare una Strategy, in modo da riutilizzare sempre lo stesso metodo (utile per aggiungere algoritmi in runtime).

### 9.2 Struttura



### 9.3 Implementazione

#### 9.3.1 Definizione delle interfacce Strategy e Context.

Le interfacce **Strategy** e **Context** devono fornire a **ConcreteStrategy** l'accesso ai dati di cui ha bisogno.

Un approccio è fare in modo che **Context** passi i dati nei parametri a **StrategyOperations**.

In ogni caso, la **Strategy** può richiedere esattamente ciò di cui ha bisogno. Le esigenze del particolare algoritmo ed i suoi requisiti di dati determineranno la tecnica migliore.

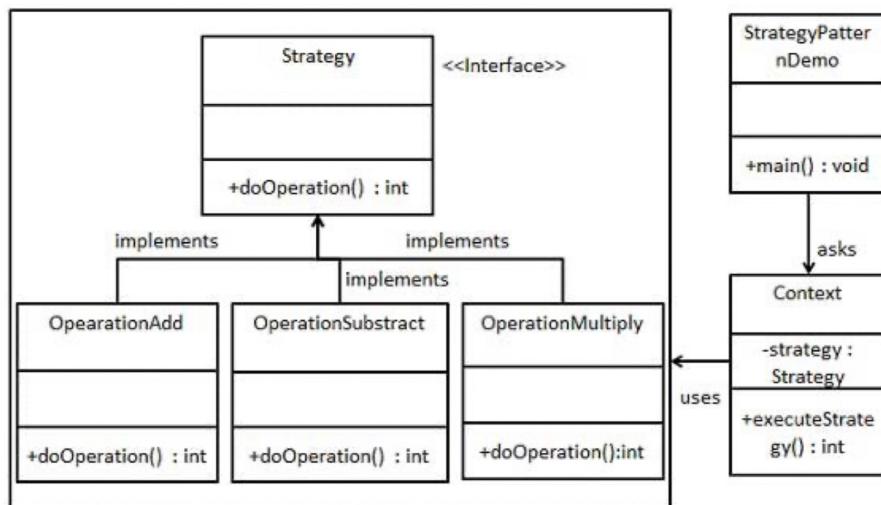
### 9.3.2 Rendere opzionali gli oggetti Strategy.

La classe Context può essere semplificata se è significativo non avere un oggetto Strategy. Il Context controlla se ha uno StrategyObject prima di accedervi.

- Se ce n'è uno, il Context lo usa normalmente.
- Se non c'è una Strategy, il contesto esegue il comportamento predefinito.

Il vantaggio di questo approccio è che i client non devono occuparsi affatto degli oggetti Strategy a meno che non amino il comportamento predefinito.

## 9.4 Esempio Java



### 9.4.1 Strategy.java

```
public interface Strategy {  
    public int doOperation(int n1, int n2);  
}
```

### 9.4.2 Context.java

```
public class Context {  
    private Strategy strategy;  
  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }
```

```
    }

    public int executeStrategy(int n1, int n2) {
        return strategy.doOperation(n1, n2);
    }
}
```

### 9.4.3 OperationAdd.java

```
public class OperationAdd implements Strategy {  
  
    @Override  
    public int doOperation(int n1, int n2) {  
        return n1 + n2;  
    }  
}
```

#### 9.4.4 OperationMult.java

```
public class OperationMult implements Strategy {  
  
    @Override  
    public int doOperation(int n1, int n2) {  
        return n1 * n2;  
    }  
}
```

### 9.4.5 main

```
public static void main(String[] args) {  
    Context c;  
  
    c = new Context(new OperationAdd());  
    System.out.println("10+5=" + c.executeStrategy(10, 5));  
  
    c = new Context(new OperationMult());  
    System.out.println("10*5=" + c.executeStrategy(10, 5));  
}
```

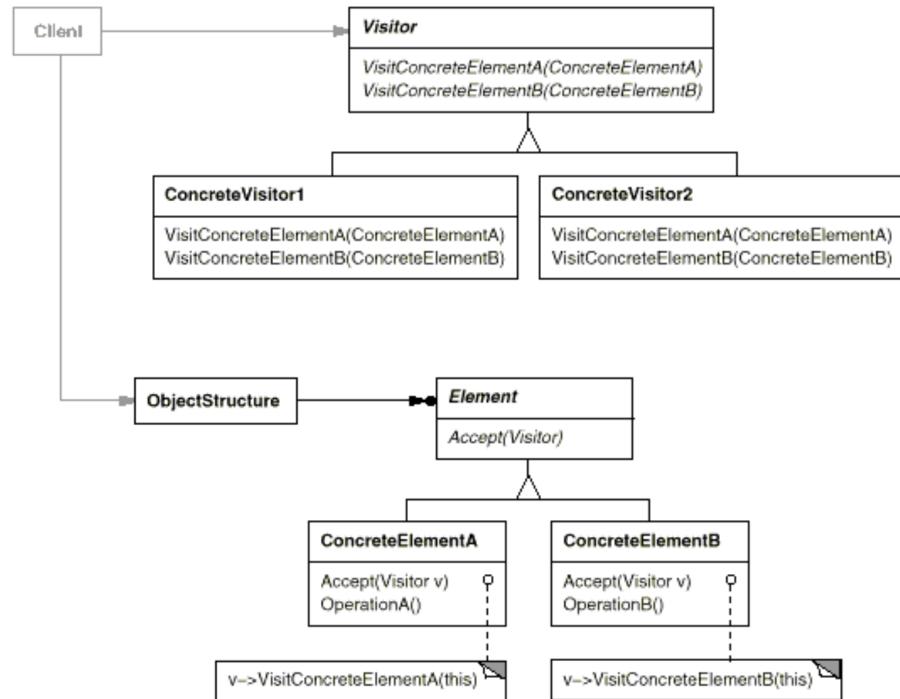
# Capitolo 10

## Visitor

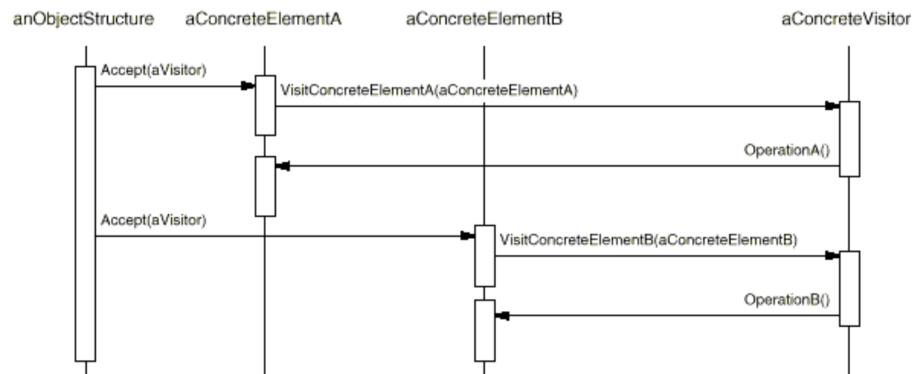
### 10.1 Intento

Rappresentare un'operazione da eseguire sugli elementi di una struttura di oggetti. Il Visitor permette di definire una nuova operazione senza modificare le classi degli elementi su cui opera.

## 10.2 Struttura



## 10.3 Timeline



## 10.4 Implementazione

### 10.4.1 Doppio invio.

Il pattern Visitor consente di aggiungere operazioni alle classi senza modificarle. Il visitatore ottiene ciò utilizzando una tecnica chiamata double-dispatch.

Nei linguaggi a single-dispatch, due criteri determinano quale operazione soddisferà una richiesta:

- nome della richiesta
- tipo di destinatario

"Doppio invio" significa semplicemente che l'operazione che viene eseguita dipende dal tipo di richiesta e dai tipi di due destinatari.

Accept è un'operazione a doppio invio. Il suo significato dipende da due tipi:

- quello del Visitator
- quello di Element

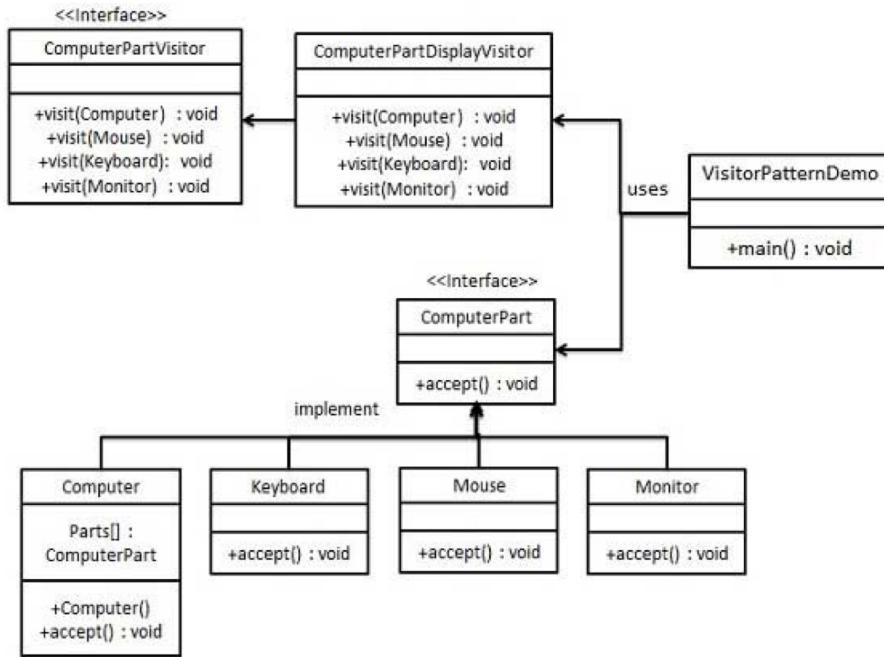
### 10.4.2 Chi è responsabile dell'attraversamento della struttura dell'oggetto?

Spesso la struttura dell'oggetto è responsabile dell'iterazione.

Una raccolta itererà semplicemente sui suoi elementi, chiamando l'operazione Accept uno per ciascuno. Un composto normalmente attraverserà se stesso facendo in modo che ogni operazione Accept attraversi i figli dell'elemento e chiami Accept su ciascuno di essi in modo ricorsivo.

Un'altra soluzione consiste nell'utilizzare un iteratore per visitare gli elementi.

## 10.5 Esempio Java



### 10.5.1 ComputerPart.java

```
public interface ComputerPart {  
    void accept(ComputerPartVisitor cpv);  
}
```

### 10.5.2 ComputerPartVisitor.java

```
public interface ComputerPartVisitor {  
    void visit(Computer computer);  
    void visit(Monitor monitor);  
    void visit(Mouse mouse);  
    void visit(Keyboard keyboard);  
}
```

### 10.5.3 Computer.java

```
public class Computer implements ComputerPart {  
    ComputerPart[] parts;
```

```

public Computer() {
    parts = new ComputerPart[] {new Keyboard(),
                                new Mouse(),
                                new Monitor()};
}

@Override
public void accept(ComputerPartVisitor cpv) {
    for (int i = 0; i < parts.length; i++) {
        parts[i].accept(cpv);
    }
    cpv.visit(this);
}

```

#### 10.5.4 Keyboard.java

```

public class Keyboard implements ComputerPart{

    @Override
    public void accept(ComputerPartVisitor cpv) {
        cpv.visit(this);
    }

}

```

#### 10.5.5 Monitor.java

```

public class Monitor implements ComputerPart{

    @Override
    public void accept(ComputerPartVisitor cpv) {
        cpv.visit(this);
    }

}

```

#### 10.5.6 Mouse.java

```

public class Mouse implements ComputerPart{
    @Override

```

```

public void accept(ComputerPartVisitor cpv) {
    cpv.visit(this);
}

}

```

#### 10.5.7 ComputerPartDisplayVisitor.java

```

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

    @Override
    public void visit(Computer computer) {
        System.out.println("Visito_computer");
    }

    @Override
    public void visit(Monitor monitor) {
        System.out.println("Visito_monitor");
    }

    @Override
    public void visit(Mouse mouse) {
        System.out.println("Visito_mouse");
    }

    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Visito_keyboard");
    }

}

```

#### 10.5.8 main

```

public static void main(String[] args) {
    ComputerPart computer = new Computer();
    computer.accept(new ComputerPartDisplayVisitor());
}

```