

Design Pattern Comportamentali

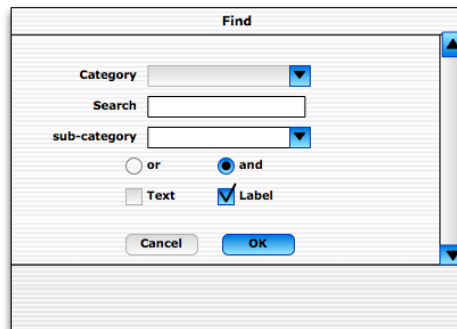
- Focalizzano sul controllo del flusso tra oggetti
- Descrivono le comunicazioni tra oggetti
- Aiutano a valutare le responsabilità assegnate agli oggetti
- Suggestiscono modi per incapsulare algoritmi dentro classi

Mediator

- Intento
 - Definisce un oggetto che incapsula come un gruppo di oggetti interagisce
 - Promuove l'ascolto accoppiamento tra oggetti poiché essi non interagiscono direttamente
- Motivazione
 - La distribuzione delle responsabilità tra gli oggetti può risultare in molte connessioni tra oggetti
 - Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si comporta come se fosse monolitico
 - Diminuire le dipendenze di una classe e renderla più generale

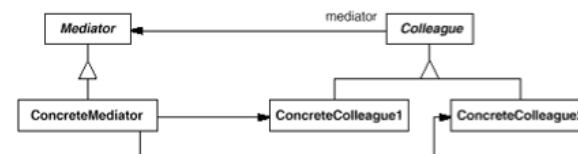
Mediator

- Per la finestra di ricerca mostrata
 - Ogni elemento visualizzato (testo, lista, bottone) è controllato da una corrispondente classe
 - Ciascuna classe deve comunicare il suo stato alle altre per far aggiornare la visualizzazione
 - Senza Mediator ciascuna classe dovrà invocare i metodi di tutte le altre classi



Mediator

- Soluzione
 - Isolare le comunicazioni (complesse) tra oggetti dipendenti creando una classe separata per esse
 - Mediator
 - Definisce un'interfaccia tra oggetti che comunicano
 - ConcreteMediator
 - Implementa il comportamento cooperativo e coordina oggetti Colleague
 - Colleague
 - Ognuno conosce il Mediator e comunica con il Mediator quando avrebbe comunicato con gli altri Colleague



Mediator

- Conseguenze

- La maggior parte della complessità che risulta nella gestione di dipendenze è spostata dagli oggetti cooperanti al Mediator. Questo rende gli oggetti più facili da implementare e mantenere
- Le classi Colleague sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con codice che gestisce le dipendenze
- Il codice del Mediator non è in genere riusabile poiché la gestione delle dipendenze implementata è solo per una specifica applicazione

Mediator

```
// classe che implementa un ConcreteMediator
public class WindowFind implements Mediator {
    // istanze di vari Colleague
    private TextButton fine = new TextButton("OK");
    private TextButton canc = new TextButton("Cancel");
    private TextBox searcher = new TextBox();
    private ListBox categ = new ListBox(categories, 2);
    // metodo che attiva i controlli
    public void selectCateg(String s) {
        if (s.compareTo(categories[0]) == 0) {
            subcateg.activate();
            fine.deactivate();
            categ.show();
        }
        ...
    }
    public void init() {
        fine.deactivate();
        canc.activate();
        searcher.show();
        categ.deactivate();
    }
}

public interface Mediator {
    // metodo che i Colleague possono invocare
    public void selectCateg(String s);
}

// classe che implementa un ConcreteColleague
public class TextButton extends Colleague {
    private String nome;
    private boolean active = false;
    public void activate() {
        active = true;
        show();
    }
    public void deactivate() {
        active = false;
        show();
    }
    public void show() {
        if (active) console.bold();
        else console.normal();
        console.print(x, y, nome);
    }
}

// classe che implementa un ConcreteColleague
public class TextBox extends Colleague {
    private String content;
    private boolean active = false;
    public void activate() {
        active = true;
        show();
    }
    public void deactivate() {
        active = false;
        show();
    }
    public void show() {
        console.normal();
        console.print(4, 1, content);
    }
    public String takeInput() {
        mediator.selectCateg(
            console.takeInput());
        // TextBox avvisa ConcreteMediator
    }
}
```

Mediator

```
// classe che implementa un ConcreteColleague
public class TextButton extends Colleague {
    private String nome;
    private boolean active = false;
    public void activate() {
        active = true;
        show();
    }
    public void deactivate() {
        active = false;
        show();
    }
    public void show() {
        console.normal();
        console.print(4, 1, content);
    }
    public String takeInput() {
        mediator.selectCateg(
            console.takeInput());
        // TextBox avvisa ConcreteMediator
    }
}

public class Colleague {
    protected Mediator mediator;
    public Mediator getMediator() {
        return mediator;
    }
    public void setMediator(Mediator m) {
        mediator = m;
    }
}
```

State

- Intento

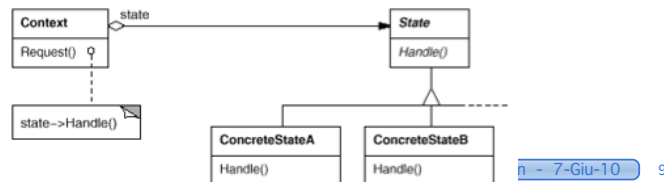
- Permette ad un oggetto di alterare il suo comportamento quando il suo stato cambia. L'oggetto appare come aver cambiato la sua classe

- Applicabilità

- Il comportamento di un oggetto dipende dal suo stato e il comportamento cambia a run-time in dipendenza del suo stato
- Le operazioni hanno condizioni che dipendono dallo stato

State

- Soluzione
 - Inserire ogni ramo condizionale in una classe separata
- Context
 - Definisce l'interfaccia di interesse per i client
 - Mantiene un'istanza di un ConcreteState che definisce lo stato corrente
- State
 - Definisce una interfaccia che incapsula il comportamento associato con un particolare stato
- ConcreteState
 - Implementa il comportamento associato con uno stato



9 - 7-Giu-10 9

State

- Conseguenze
 - Inserisce il comportamento associato ad uno stato in una sola classe (ConcreteState)
 - Permette di incorporare la logica che gestisce il cambiamento di stato separatamente ed in una sola classe (Context), anziché (con istruzioni if o switch) sulla classe che implementa i comportamenti
 - Aiuta ad evitare stati inconsistenti poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante
 - Incrementa il numero di classi

Prof. E. Tramontana - Design Pattern - 7-Giu-10 10

State

```
// TestOrologio e' un Client
public class TestOrologio {
    public static void main(String[] args) {
        Orario o = new Orario();
        o.intoDigitale();
        o.oraAttuale();
    }
}

// Orario ha il ruolo di Context
public class Orario {
    private Orologio o;
    public void oraAttuale() {
        o.displayTime();
    }
    public void intoDigitale() {
        o = new Digitale()
    }
    public void intoAnalog() {
        o = new Analogico()
    }
}

// Orologio definisce l'interfaccia State
public interface Orologio {
    public void displayTime();
}

// Digitale ha il ruolo di un ConcreteState
public class Digitale implements Orologio {
    public void displayTime() {
        Date d = Calendar.getInstance().getTime();
        System.out.print(
            DateFormat.getDateTimeInstance().format(d));
    }
}

// Analogico ha il ruolo di un ConcreteState
public class Analogico implements Orologio {
    public void displayTime() {
        // implementa un comportamento
    }
}
```

Prof. E. Tramontana - Design Pattern - 7-Giu-10 11

Decorator

- Intento
 - Attaccare responsabilità aggiuntive ad un oggetto dinamicamente. Fornire un'alternativa flessibile alla creazione di sottoclassi per estendere funzionalità
- Motivazione
 - A volte si vuole aggiungere una responsabilità ad un singolo oggetto, non all'intera classe
 - Le responsabilità possono essere sottratte dinamicamente
 - Ad es. per un componente grafico si vorrebbe poter aggiungere la proprietà bordo, se si eredita da una classe Bordo, tutti gli oggetti della sottoclasse avranno questa proprietà, quindi niente flessibilità
 - Alternativa flessibile, inserire il componente grafico dentro un oggetto che aggiunge il bordo. L'oggetto che racchiude, chiamato Decorator, manda la richiesta al componente e aggiunge altre attività prima o dopo l'invio della richiesta
 - I Decorator possono essere annidati ricorsivamente, per aggiungere più di una responsabilità
 - A volte la creazione di sottoclassi non è praticabile. Un numero grande di estensioni produrrebbe un numero enorme di sottoclassi per gestire tutte le combinazioni

Prof. E. Tramontana - Design Pattern - 7-Giu-10 12

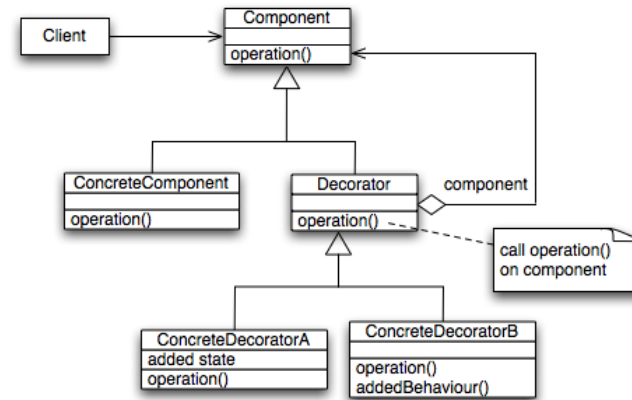
Decorator

- Soluzione

- Component definisce l'interfaccia per gli oggetti che possono avere aggiunte delle responsabilità dinamicamente
- ConcreteComponent definisce un oggetto si cui poter aggiungere responsabilità
- Decorator mantiene un riferimento all'oggetto Component e definisce una interfaccia conforme a quella di Component
 - Decorator inoltra le richieste al suo oggetto Component e può effettuare altre operazioni prima e dopo l'inoltro della richiesta
- ConcreteDecorator aggiunge responsabilità al componente

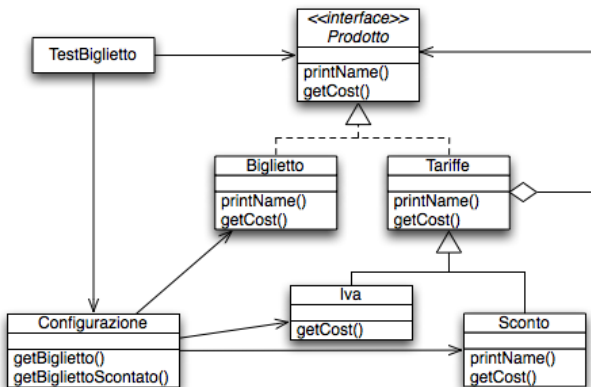
Decorator

- Struttura



Esempio Decorator

- Ho alcuni prodotti, ad es. Biglietto, ognuno con un nome ed un costo; ho diversi modi di calcolare il costo di questi prodotti, in base a come applico sconti e percentuali IVA
- Voglio poter combinare a runtime sconti diversi sui diversi prodotti



Esempio Decorator

```
// Biglietto e' un ConcreteComponent
public class Biglietto implements Prodotto {
    public void printName() {
        System.out.print("Biglietto n. 1231231");
    }
    public double getCost() {
        return 100.0;
    }
}
```

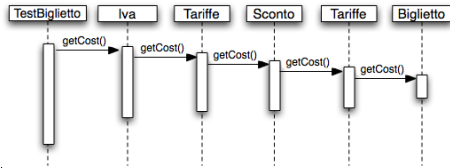
```
// Prodotto e' un Component
public interface Prodotto {
    public void printName();
    public double getCost();
}
```

```
// Tariffe e' un Decorator,
// contiene l'istanza dell'oggetto annidato ed
// invoca operazioni sull'oggetto annidato
public abstract class Tariffe implements Prodotto {
    private Prodotto myComp;
    public Tariffe(Prodotto myC) {
        myComp = myC;
    }
    public void printName() {
        myComp.printName();
    }
    public double getCost() {
        return myComp.getCost();
    }
}
```

```
// Sconto e' un ConcreteDecorator,
// aggiunge funzionalita' ad un prodotto
public class Sconto extends Tariffe {
    public Sconto(Prodotto myC) {
        super(myC);
    }
    public void printName() {
        //non chiama l'oggetto annidato
        System.out.print("Biglietto ridotto");
    }
    public double getCost() {
        //chiama getCost sull'oggetto annidato
        return super.getCost()*0.95;
    }
}
```

Esempio Decorator

```
// Configurazione definisce alcuni metodi factory
public class Configurazione {
    // getBiglietto restituisce un tipo Prodotto
    public static Prodotto getBiglietto() {
        // Iva e' un Decorator per Biglietto
        return new Iva(new Biglietto());
    }
    public static Prodotto getBigliettoScontato() {
        // qui ho due annidamenti
        return new Iva(new Sconto(new Biglietto()));
    }
}
```



```
// TestBiglietto e' un client
public class TestBiglietto {
    public static void main(String[] args) {
        Prodotto prod;
        prod = Configurazione.getBigliettoScontato();
        prod.printName();
        System.out.println(" costo: " + prod.getCost());
        prod = Configurazione.getBiglietto();
        prod.printName();
        System.out.println(" costo: " + prod.getCost());
    }
}
```

```
// Iva e' un ConcreteDecorator
public class Iva extends Tariffe {
    public Iva(Prodotto myC) {
        super(myC);
    }
    public double getCost() {
        return super.getCost()*1.2;
    }
}
```