

DESIGN PATTERN

Cos'è un Design Pattern?

un DP è una struttura (un certo numero di classi) fatti in modo per risolvere dei problemi nei programmi abbastanza noti. (problemi ricorrenti)

Cosa ci serve?

LA STRUTTURA, IL CODICE è SECONDARIO. (il codice è una conseguenza della struttura)

La struttura è per un piccolo numero di classi (in paragone ai vari software enormi (che hanno anche migliaia di classi) questo gestisce nell'ordine delle decine).

- I DP hanno delle documentazioni che spiegano le conseguenze, vantaggi e svantaggi delle singole classi che vengono implementate
- Esistono DP per sistemi diversi: **Centralizzati, concorrenti, distribuiti, real-time, etc.**

SONO ORGANIZZATI IN CATALOGHI IN BASE ALLO SCOPO

- **Creazionali:** Riguardano la creazione di istanze
Singleton, Factory Method, Abstract Factory, Buuilder, Prototype
- **Strutturali:** riguardano la scelta della struttura
Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy(Surrogato)
- **Comportamentali:** riguardano la scelta dell'incapsulamento degli algoritmi
Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visitor

Descrizione dei Design Pattern

- **Nome:** Identifica il DP con un nome e di lavorare con un alto livello di astrazione. Indica lo scopo
- **Intento:** Descrive brevemente le funzionalità e lo scopo
- **Problema:** (motivazione + applicabilità) descrive a quali problemi in DP viene applicato e quali sono le condizioni necessarie per applicarlo
- **Soluzione:** descrive gli elementi(classi) che costituiscono il DP, le loro responsabilità e le loro relazioni
- **Conseguenze:** Indicano i risultati, i vantaggi e svantaggi, compromessi dell'uso del DP

Singleton

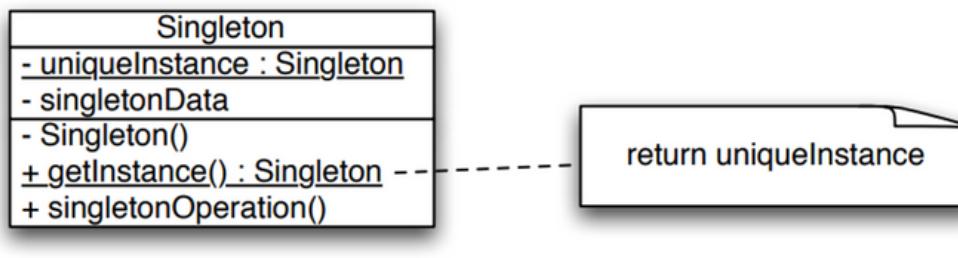
INTENTO: Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale all'istanza

MOTIVAZIONE:

- Alcune classi dovrebbero avere esattamente una istanza in tutta l'applicazione, es. uno spooler di stampa, un file system, un window manager, una lista clienti, etc.
- Una variabile globale rende un oggetto accessibile ma non proibisce di avere più oggetti per una classe
- La classe stessa dovrebbe essere responsabile di tener traccia del suo unico punto di accesso

SOLUZIONE

- La classe che deve essere un Singleton dovrà implementare un'operazione `getInstance()` sulla classe (ovvero, in Java è un metodo static) che ritorna l'unica istanza creata
- La classe Singleton è responsabile per la creazione dell'istanza
- Il costruttore della classe Singleton è privato, così da non permettere la creazione tramite new ad altre classi



ESEMPI CODICE

```

public class Fib {      // classe che implementa un Singleton
    private static Fib obj = new Fib(); // istanza di Fib
    private int[] x = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233};
    private int i;

    private Fib() {           // costruttore privato
        i = 3;
    }

    public static Fib getInstance() { // metodo della classe
        return obj;               // restituisce l'unica istanza
    }

    public int getValue() {
        if (i<11) i++;
        return x[i-1];
    }

    public void revert() {
        i = 0;
    }
}
  
```

```

public class TestFib {

    public static void main(String[] args) {
        // richiede una istanza di Fib
        Fib f = Fib.getInstance();

        System.out.print("f "+f.getValue());
        System.out.println(" "+f.getValue());

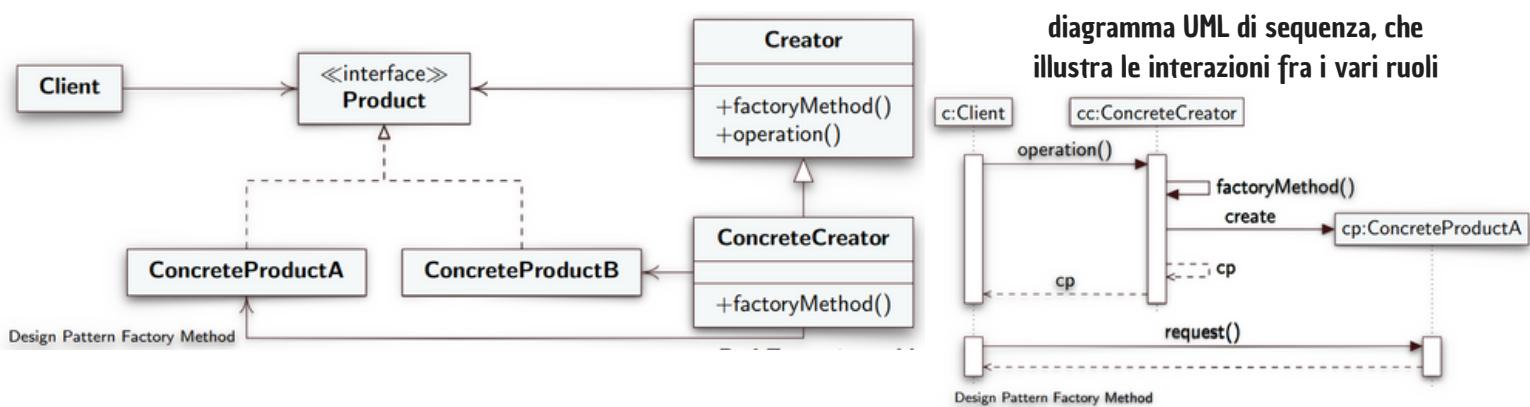
        // richiede una nuova istanza
        Fib f2 = Fib.getInstance();

        System.out.print("f2 "+f2.getValue());
        System.out.println(" "+f2.getValue());

        // Si ha un errore a compile-time con:
        // Fib f3 = (Fib) f2.clone();
        // Fib f4 = new Fib();
    }
}
  
```

Factory Method

- INTENTO:** Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi
- PROBLEMA:**
 - Un framework usa classi astratte per definire e mantenere relazioni tra oggetti. Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare.
 - Un metodo responsabile per l'istanziazione (detto factory, ovvero fabbricatore) incapsula la conoscenza su quale classe creare
- SOLUZIONE:**
 - Product è l'interfaccia comune degli oggetti creati da factoryMethod()
 - ConcreteProduct è un'implementazione di Product
 - Creator dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Creator può avere un'implementazione si default del factoryMethod() che ritorna un certo ConcreteProduct
 - ConcreteCreator implementa il factoryMethod(), o ne fa override, sceglie quale ConcreteProduct istanziare e ritorna tale istanza



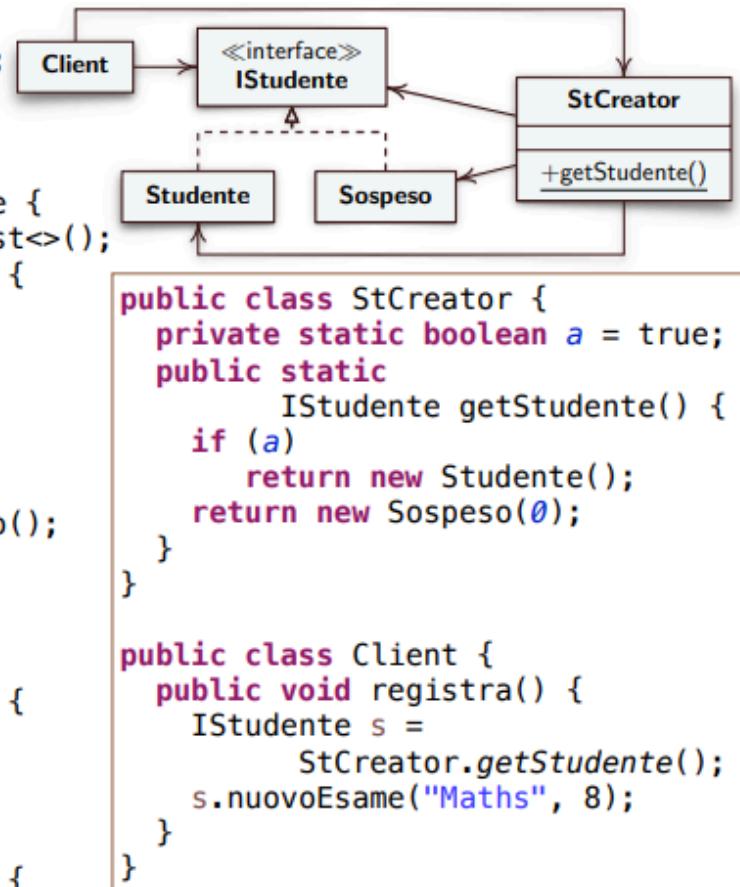
```

public interface IStudente {
    public void nuovoEsame(String m, int v);
    public float getMedia();
}

public class Studente implements IStudente {
    private List<Esame> esami = new ArrayList<>();
    public void nuovoEsame(String m, int v) {
        Esame e = new Esame(m, v);
        esami.add(e);
    }
    public float getMedia() {
        if (esami.isEmpty()) return 0;
        float sum = 0;
        for (Esame e : esami) sum += e.getVoto();
        return sum / esami.size();
    }
}

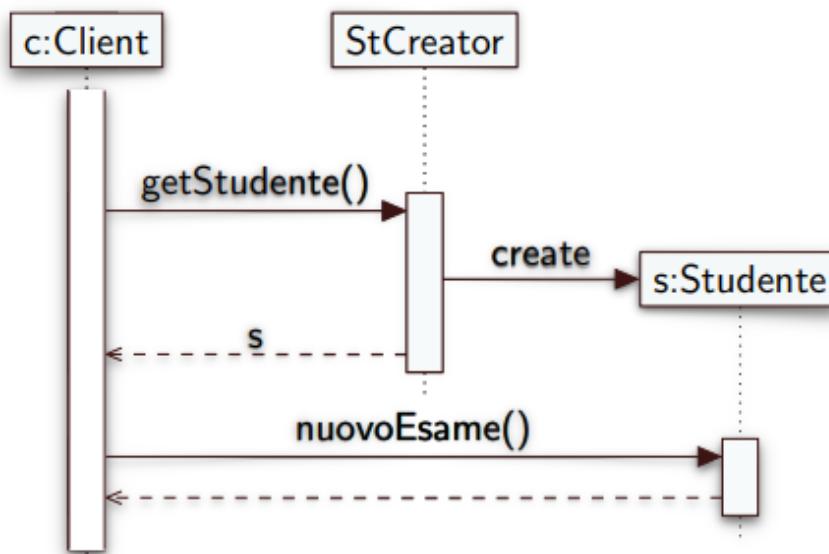
public class Sospeso implements IStudente {
    private float media;
    public Sospeso(float m) {
        media = m;
    }
    public void nuovoEsame(String m, int v) {
        System.out.println("Non e' possibile sostenere esami");
    }
    public float getMedia() {
        return media;
    }
}

```



Factory Method

Nel precedente esempio di codice, l'interfaccia IStudente svolge il ruolo Product, le classi Studente e Sospeso svolgono il ruolo ConcreteProduct, e la classe StCreator svolge il ruolo ConcreteCreator



- **Varianti**

- Il ruolo Creator e ConcreteCreator sono svolti dalla stessa classe
- Il factoryMethod() è un metodo static
- Il factoryMethod() ha un parametro che permette al client di suggerire la classe da usare per creare l'istanza
- Il factoryMethod() usa la Riflessione Computazionale, quindi Class.forName() e newInstance(), per eliminare le dipendenze dai ConcreteProduct, la classe istanziata sarà nota a runtime

```

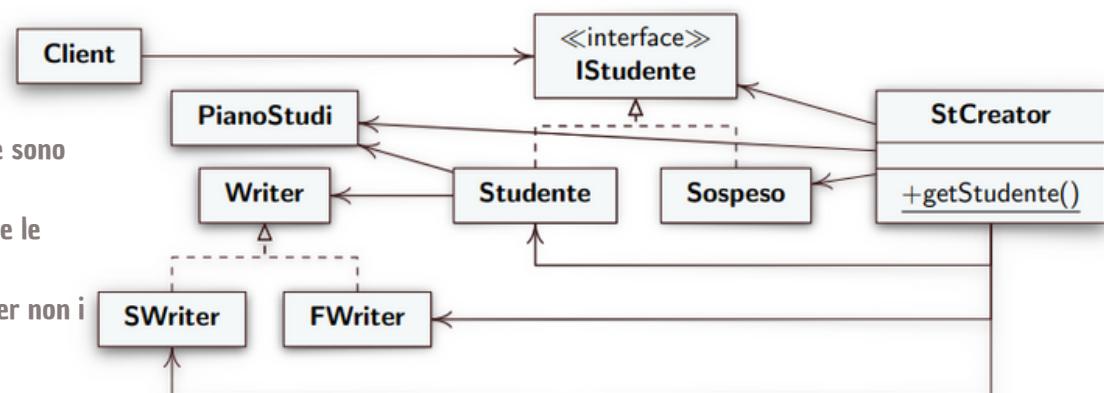
try {
    Class<?> cls = Class.forName("Studente"); // Il nome della classe è una stringa
    Constructor<?> cnstr = cls.getConstructor(new Class[] {});
    return (IStudente) cnstr.newInstance();
}
catch (InstantiationException | IllegalAccessException | IllegalArgumentException | InvocationTargetException | NoSuchMethodException | SecurityException |
ClassNotFoundException e) {
    e.printStackTrace();
}
  
```

Dependency Injection

- Il design pattern Factory Method può essere usato per inserire le dipendenze (dependency injection) necessarie alle istanze di ConcreteProduct
- Tramite la Dependency Injection un oggetto (client) riceve altri oggetti da cui dipende, questi altri oggetti sono detti dipendenze
- La tecnica di Dependency Injection permette di separare la costruzione delle istanze dal loro uso
- Il client non crea l'istanza di cui ha bisogno
- Le dipendenze sono iniettate al client per mezzo di parametri nel suo costruttore. Questo permette di evitare complicazioni derivanti da metodi setter e da controlli per verificare che le dipendenze non siano null, di conseguenza il codice è più semplice
- L'oggetto che fa Dependency Injection si occupa di connettere (fa wiring di) varie istanze. In un unico posto vediamo le connessioni fra gli oggetti

CONSEGUENZE

- Il codice delle classi dell'applicazione conosce solo l'interfaccia Product e può lavorare con qualsiasi ConcreteProduct. I ConcreteProduct sono facilmente intercambiabili
- Se si implementa una sottoclasse di Creator per ciascun ConcreteProduct da istanziare si ha una proliferazione di classi



ESEMPIO

- Si abbiano Writer e PianoStudi che sono dipendenze per Studente
- Studente riceve nel suo costruttore le istanze di Writer e PianoStudi
- Studente conosce solo il tipo Writer non i suoi sottotipi

Factory Method

Object Pool

- Un **object pool** è un deposito di istanze già create, una istanza sarà estratta dal pool quando una classe client ne fa richiesta
 - Il pool può crescere o può avere **dimensioni fisse**. Dimensioni fisse: se non ci sono oggetti disponibili al momento della richiesta, non ne creo di nuovi
 - Il client restituisce al pool l'istanza usata quando non più utile
- Il design pattern Factory Method può implementare un object pool
 - I client richiedono istanze, come visto per il Factory Method
 - I client dovranno indicare quando l'istanza non è più in uso, quindi riusabile
 - Lo stato dell'istanza da riusare potrebbe dover essere riscritto
 - L'object pool dovrebbe essere unico: potrei usare un Singleton

Esempio Di Object Pool

```
import java.util.ArrayList;
import java.util.List;

// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool {
    private List<Shape> pool = new ArrayList<>();

    // metodo factory che ritorna un oggetto prelevato dal pool
    public Shape getShape() {
        if (pool.size() > 0)
            return pool.remove(0);
        return new Circle();
    }

    // inserisce un oggetto nel pool
    public void releaseShape(Shape s) {
        pool.add(s);
    }
}
```

Abstract Factory

INTENTO

Fornire un'interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete

PROBLEMA

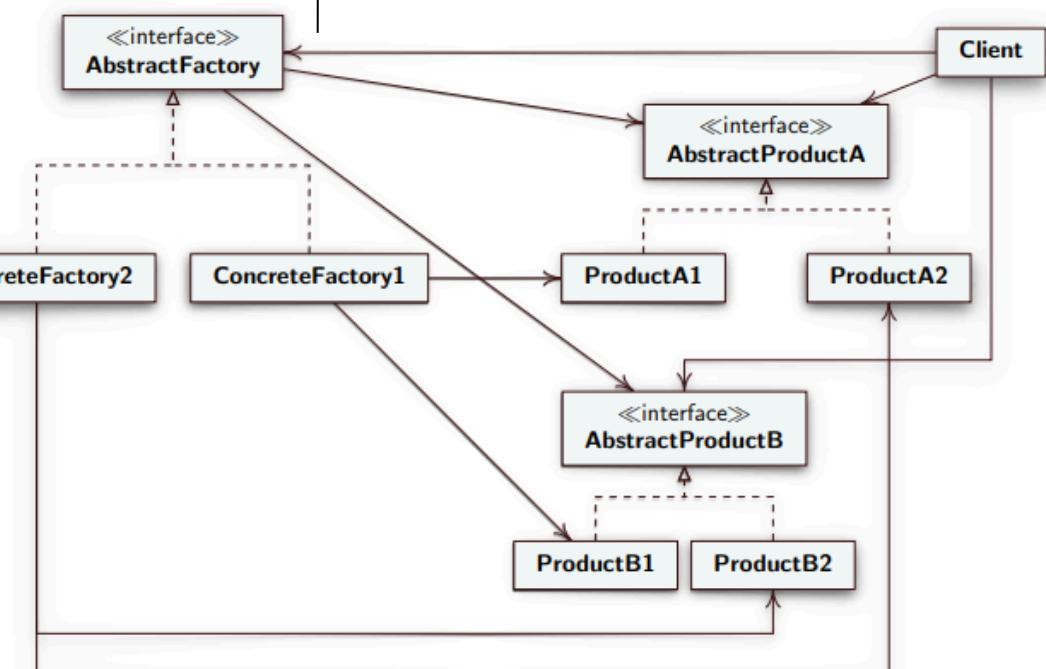
- Il sistema complessivo dovrebbe essere **indipendente dalle classi usate**, così da essere configurabile con una di varie famiglie di classi. Le classi di una famiglia dovrebbero essere usate in modo consistente
- Es. lo strato di interfaccia utente permette vari tipi di look-andfeel, così differenti comportamenti per accessori (widget) dell'interfaccia utente sono possibili

CONSEGUENZE

- Permette di usare classi consistentemente (per famiglie)
- Le famiglie di classi sono facilmente intercambiabili
- Non è immediato supportare nuove classi Product, poiché bisogna aggiungere un metodo su AbstractFactory e su ogni ConcreteFactory

SOLUZIONE

- AbstractFactory**: interfaccia astratta per creare famiglie di oggetti
- ConcreteFactory**: classi che implementano operazioni per creare ciascuna famiglia di oggetti
- AbstractProduct** è l'interfaccia per una famiglia di oggetti
- Product** definisce un oggetto, creato da un ConcreteFactory, che implementa l'interfaccia AbstractProduct
- Il **client** usa solo interfacce dichiarate da AbstractFactory e AbstractProduct



```

interface Icon { // AbstractProductA
    public void draw();
    public void fill();
}

interface Text { // AbstractProductB
    public void tell();
    public void shout();
}

interface Creator { // AbstractFactory
    public Icon getIcon(); // create method
    public Text getText();
}

// ConcreteFactory
class Creator1 implements Creator {
    public Icon getIcon() {
        return new Circle();
    }
    public Text getText() {
        return new Japanese();
    }
}

class Creator2 implements Creator {
    public Icon getIcon() {
        return new Box();
    }
    public Text getText() {
        return new English();
    }
}

class Circle implements Icon { // ProductA1
    public void draw() {
        System.out.print("C ");
    }
    public void fill() {
        System.out.print("o ");
    }
}

class Box implements Icon { // ProductA2
    public void draw() {
        System.out.print("[ ] ");
    }
    public void fill() {
        System.out.print("[X] ");
    }
}

class Japanese implements Text { // ProductB1
    public void tell() {
        System.out.println(" Youkoso. Konnichiwa! Hajimemashite ");
    }
    public void shout() {
        System.out.println(" Shizuka ni shite kudasai ");
    }
}

class English implements Text { // ProductB2
    public void tell() {
        System.out.println("::::: Welcome. Nice to meet you :::::");
    }
    public void shout() {
        System.out.println("::::: Be quiet please! :::::");
    }
}

public class AbsFactorTest {
    public static void main(String args[]) {
        Creator c = new Creator1(); // istanzio un Creator
        Icon ic = c.getIcon();
        Text t = c.getText();
        ic.draw();
        t.tell();
    }
}
  
```

Prototype

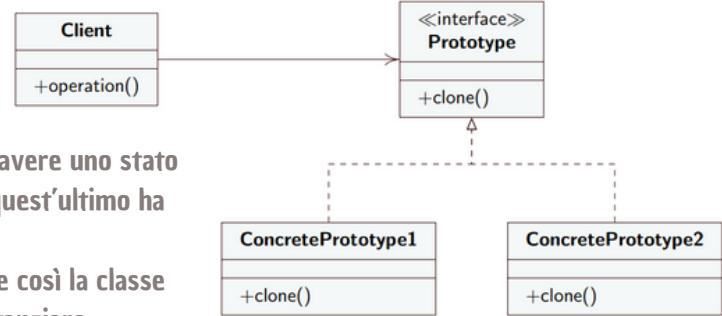
INTENTO: specificare i tipi di oggetti da creare usando una istanza di un prototipo, e creare nuovi oggetti copiando questo prototipo

MOTIVAZIONE

- Durante la creazione di un nuovo oggetto può essere necessario avere uno stato iniziale che è simile a quello di un altro oggetto esistente, e che quest'ultimo ha ottenuto per mezzo di operazioni eseguite su esso
- Quando si crea un oggetto bisogna indicare una specifica classe, e così la classe che chiede l'istanza è strettamente accoppiata con la classe da istanziare
- Lo stretto accoppiamento fra classi da istanziare e chi istanzia rende difficile l'aggiunta di nuovi tipi, per es. nel Factory Method bisogna creare una gerarchia di Creator per aggiungere nuovi tipi

CONSEGUENZE

- Il client non conosce la classe usata, poiché il client usa l'interfaccia comune, quindi il numero di dipendenze del client è ridotto
- Il client può usare classi specifiche (sottoclassi) pur non conoscendone il nome
- Si può registrare un nuovo prototipo (o rimuovere un prototipo) a runtime
- Si possono specificare nuovi oggetti che tengono valori diversi, senza essere obbligati a creare nuove classi
- Un nuovo prototipo può essere usato per tenere una struttura diversa di oggetti (composizione). Gli oggetti della composizione sono attributi del prototipo, come in un Composite
- Per avendo tante classi diverse da poter usare (i ConcretePrototype), non si ha necessità di implementare una gerarchia di Creator (come invece avviene per il Factory Method), in quanto un nuovo oggetto si ottiene tramite clonazione
- È possibile creare una configurazione (un prototipo) dinamicamente



STRUTTURA

Prototype definisce l'interfaccia per le operazioni comuni e dichiara un metodo per clonare se stessa. **ConcretePrototype** è una classe che implementa l'operazione di clonazione. **Client** è una classe che crea nuovi oggetti tramite clonazione dell'istanza prototipo, quindi senza indicare la classe esplicitamente. Client conosce solo l'interfaccia **Prototype**.

IMPLEMENTAZIONE

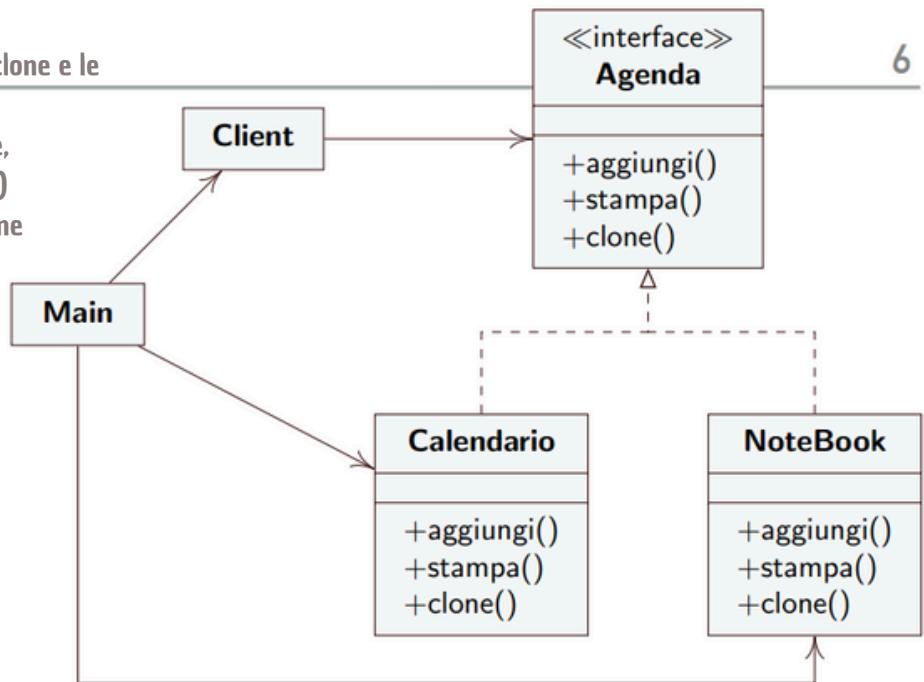
- L'uso di un **manager di prototipi** (che è un registro di istanze) permette ai client di immagazzinare e recuperare i prototipi, prima di clonarli
- L'implementazione dell'operazione di clonazione può essere la parte più difficile quando le strutture contengono riferimenti circolari
 - Sebbene un'operazione di clonazione potrebbe essere fornita dal linguaggio di programmazione, l'operazione di clonazione potrebbe effettuare una **copia superficiale (shallow copy)** anziché una **copia profonda (deep copy)**
 - In una shallow copy i riferimenti tenuti negli attributi dell'oggetto da clonare sarebbero condivisi fra originale e clone, e questo spesso non è sufficiente, quindi occorre una deep copy
- I cloni potrebbero necessitare di essere inizializzati: la classe prototipo potrebbe definire operazioni per impostare dati

Prototype

ESEMPIO

- Si vuol tenere un'agenda di impegni, varie voci possono essere aggiunte all'agenda
- Lo stato dell'agenda è l'insieme di impegni presi
- Una nuova istanza dell'agenda deve poter avere accesso agli impegni precedentemente inseriti
- In modo simile, si vogliono inserire note in un quaderno e accedere alle note precedenti quando si ha una nuova istanza di quaderno

- Agenda è un Prototype: definisce l'operazione clone e le operazioni per le classi concrete
- Calendario e NoteBook sono ConcretePrototype, implementano le operazioni (compresa la clone)
- La classe Client chiama l'operazione di clonazione



```

public interface Agenda {
    public void aggiungi(String evento, LocalDateTime data);
    public void stampa();
    public Agenda clone();
}

```

```

public class NoteBook implements Agenda {
    private List<String> note = new ArrayList<>();
    @Override public void aggiungi(String evento,
                                   LocalDateTime t) {
        note.add(evento + ", " + t.getDayOfWeek() + " "
                 + t.getHour() + ":" + t.getMinute());
    }
    @Override public void stampa() {
        note.forEach(e -> System.out.println(e));
    }
    @Override public Agenda clone() {
        // deep copy
        List<String> n = new ArrayList<>();
        n.addAll(note);
        NoteBook notenew = new NoteBook();
        notenew.note = n;
        return notenew;
    }
}

```

State

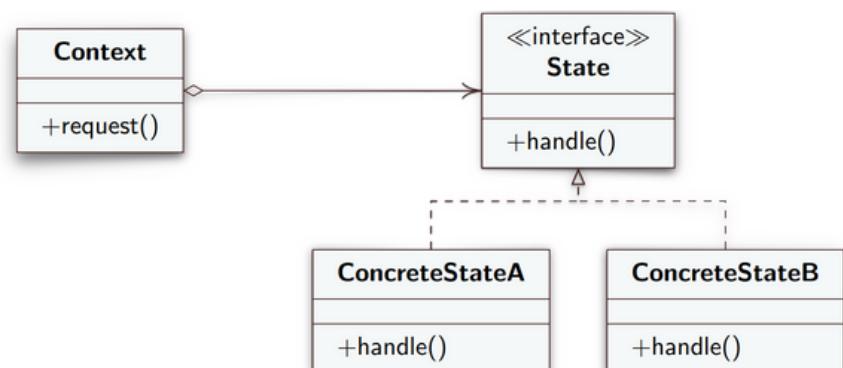
INTENTO: Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Far sembrare che l'oggetto abbia cambiato la sua classe

PROBLEMA:

- Il comportamento di un oggetto dipende dal suo stato e il comportamento deve cambiare a run-time in base al suo stato
- Le operazioni da svolgere hanno alcuni grandi rami condizionali che dipendono dallo stato
- Lo stato è spesso rappresentato dal valore di una o più variabili numerative costanti
- Spesso varie operazioni contengono la stessa struttura condizionale

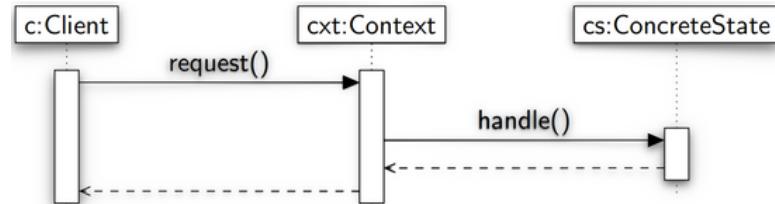
SOLUZIONE: Inserire ogni ramo condizionale in una classe separata

- **Context** definisce l'interfaccia che interessa ai client, e mantiene un'istanza di una classe **ConcreteState** che definisce lo stato corrente
- **State** definisce un'interfaccia che incapsula il comportamento associato ad un particolare stato del Context
- **ConcreteState** sono le sottoclassi che implementano ciascuna il comportamento associato ad uno stato del Context



COLLABORAZIONI:

- Il **Context** passa le richieste dipendenti da un certo stato all'oggetto **ConcreteState** corrente
- Un **Context** può passare se stesso come argomento all'oggetto **ConcreteState** per farlo accedere al contesto se necessario
- Il **Context** è l'interfaccia per le classi client
- Il **Context** o i **ConcreteState** decidono quale stato è il successivo ed in quali circostanze

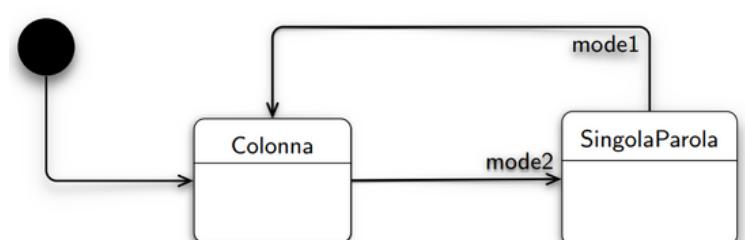
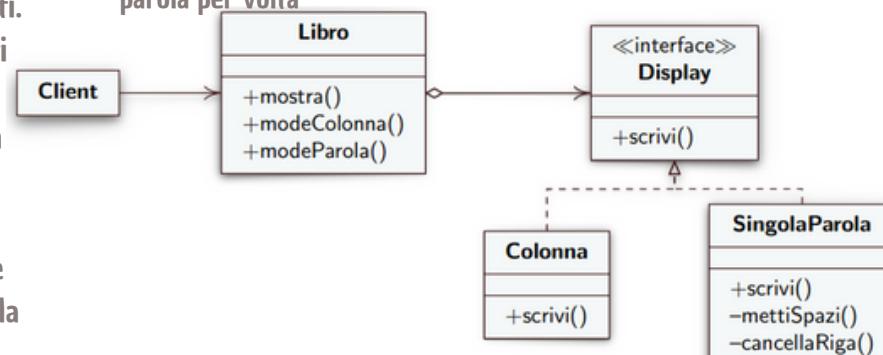


CONSEGUENZE

- Il comportamento associato a uno stato è localizzato in una sola classe (**ConcreteState**) e si partiziona il comportamento di stati differenti. Per tale motivo, si posso aggiungere nuovi stati etransizioni facilmente, creando nuove sottoclassi. Incapsulare le azioni di uno stato in una classe impone una struttura e rende più chiaro lo scopo del codice
- La logica che gestisce il cambiamento di stato è separata dai vari comportamenti ed è in una sola classe (**Context**), anziché (con istruzioni if o switch) sulla classe che implementa i comportamenti. Tale separazione aiuta a evitare stati inconsistenti, poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante
- Il numero di classi totale è maggiore, le classi sono più semplici

ESEMPIO

Si vogliono avere vari modi per scrivere il testo di un libro su un display: in modalità una colonna, due colonne, o una singola parola per volta



State

```

public class Libro { // Context
    private String testo = "Darwin's _Origin of Species_ persuaded the world that the "
        + "difference between different species of animals and plants is not the fixed "
        + "immutable difference that it appears to be.";
    private List<String> lista = Arrays.asList(testo.split("[\\s]+"));
    private Display mode = new Colonna();
    public void mostra() {
        mode.scrivi(lista);
    }
    public void modeColonna() {
        mode = new Colonna();
    }
    public void modeParola() {
        mode = new SingolaParola();
    }
}

public class Colonna implements Display { // ConcreteState
    private final int numCar = 38;
    private final int numRighe = 12;
    public void scrivi(List<String> testo) {
        int riga = 0;
        int col = 0;
        for (String p : testo) {
            if (col + p.length() > numCar) {
                System.out.println();
                riga++;
                col = 0;
            }
            if (riga == numRighe) break;
            System.out.print(p + " ");
            col += p.length() + 1;
        }
    }
}

```

1

```

public interface Display { // State
    public void scrivi(List<String> testo);
}

```

```

public class Client {
    public static void main(String[] args) {
        Libro l = new Libro();
        l.mostra();
        l.modeParola();
        l.mostra();
    }
}

```

```

public class SingolaParola implements Display { // ConcreteState
    private int maxLung;
    public void scrivi(List<String> testo) {
        System.out.println();
        trovaMaxLung(testo);
        for (String p : testo) {
            int numSpazi = (maxLung - p.length()) / 2;
            mettiSpazi(numSpazi);
            System.out.print(p);
            if (p.length() % 2 == 1) numSpazi++;
            mettiSpazi(numSpazi);
            aspetta();
            cancellaRiga();
        }
        System.out.println();
    }
    private void mettiSpazi(int n) {
        System.out.print(" ".repeat(n));
    }
    private void cancellaRiga() {
        System.out.print("\b".repeat(maxLung));
    }
    private void trovaMaxLung(List<String> testo) {
        for (String p : testo) if (maxLung < p.length()) maxLung = p.length();
    }
    private static void aspetta() {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) { }
    }
}

```

2

```

public class LibroPrimaDiState {
    private String testo = "...";
    private List<String> lista = Arrays.asList(testo.split("[\\s]+"));
    private int mode = 2;

    public void mostra() {
        switch (mode) {
            case 1:
                // vedi metodo scrivi della classe SingolaParola
                break;
            case 2:
                // vedi metodo scrivi della classe Colonna
                break;
        }
    }

    public void setMode(int x) {
        mode = x;
    }
}

```

3

Observer

INTENTO:

Definire una dipendenza uno a molti fra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e aggiornati automaticamente
 (Conosciuto anche come Publish-Subscribe)

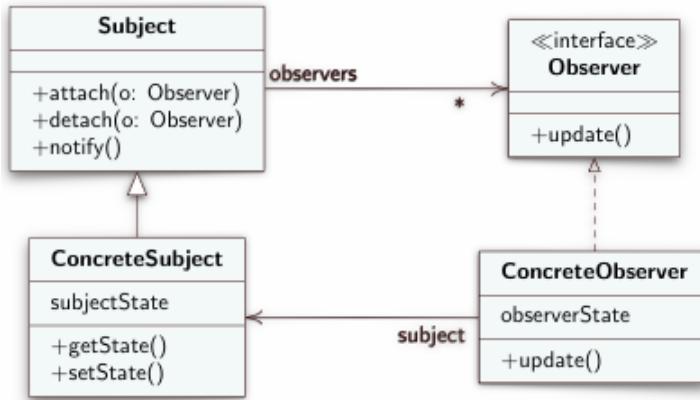
Motivazioni

- Un sistema che è stato partizionato in un insieme di classi che cooperano deve mantenere la consistenza fra oggetti che hanno relazioni
- Es. uno strumento di presentazione è separato dai dati dell'applicazione. Le classi che tengono i dati e quelle di presentazione sono separate e riusabili. Uno spreadsheet e un diagramma sono dipendenti dall'oggetto che contiene i dati e dovrebbero essere notificati dei cambiamenti dei dati

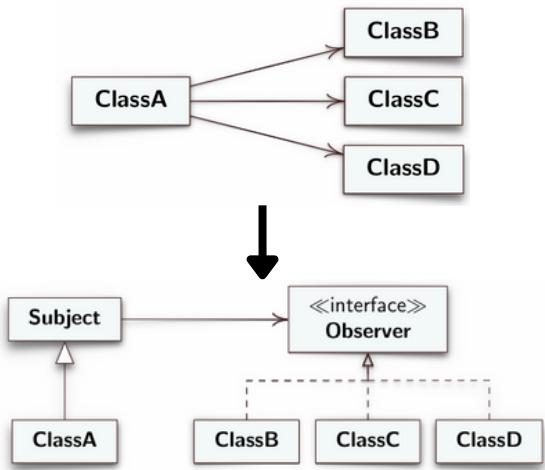
Il design pattern Observer descrive come stabilire relazioni fra oggetti dipendenti

- Gli oggetti chiave sono **Subject** e **Observer**
- Un Subject può avere tanti Observer che dipendono da esso e gli Observer sono notificati quando lo stato del Subject cambia

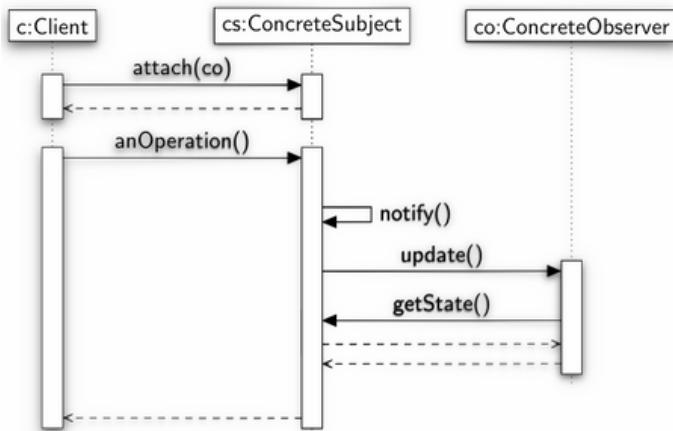
Diagramma delle classi:



Prima e Dopo l'Uso di Observer



Quando lo stato dell'oggetto **ConcreteSubject** cambia, l'oggetto **ConcreteSubject** chiama `notify()` che chiamerà `update()` sugli oggetti **ConcreteObserver** per aggiornarli



Partecipanti:

- **Subject** conosce i suoi osservatori, un qualsiasi numero di oggetti **Observer** può osservare un **Subject**. Implementa le operazioni per aggiungere e togliere oggetti **Observer** e per notificarli
- **Observer** definisce una interfaccia (`operazione update()`) comune a tutti gli oggetti che necessitano la notifica
- **ConcreteSubject** tiene lo stato che interessa agli oggetti **ConcreteObserver**. Notifica i suoi osservatori quando il suo stato cambia. Eredita da **Subject**
- **ConcreteObserver** tiene un riferimento all'oggetto **ConcreteSubject**, e tiene lo stato che deve rimanere consistente con quello del **Subject**. Implementa **Observer** per ricevere notifiche dei cambiamenti del **Subject**. Dopo la notifica può interrogare il **subject** per ottenere il nuovo dato

Conseguenze

- Il **Subject** conosce solo la classe **Observer** e non ha bisogno di conoscere le classi **ConcreteObserver**. **ConcreteSubject** e **ConcreteObserver** non sono accoppiati quindi più facili da riusare e modificare
- La notifica inviata da un **ConcreteSubject** è mandata a tutti gli oggetti che si sono iscritti, il **ConcreteSubject** non sa quanti sono. Gli osservatori possono essere rimossi in qualunque momento. L'osservatore sceglie se gestire o ignorare la notifica
- L'aggiornamento da parte del **Subject** può far avviare tante operazioni sugli **Observer** e altri oggetti per gli aggiornamenti. La notifica non dice agli **Observer** cosa è cambiato nel **ConcreteSubject**, è un evento che indica il completamento di un'operazione del **ConcreteSubject**

Observer

```

public class Subject {
    private List<Observer> obs = new ArrayList<>();
    private boolean changed = false;
    public void notify(Object state) {
        if (!changed) return;
        for (Observer o : obs) o.update(this, state);
        changed = false;
    }
    public void setChanged() {
        changed = true;
    }
    public void attach(Observer o) {
        obs.add(o);
    }
    public void detach(Observer o) {
        obs.remove(o);
    }
}

public class AddrBook extends Subject {
    private List<Persona> nomi = new ArrayList<>();

    public void insert(Persona p) {
        if (nomi.contains(p)) return;
        nomi.add(p);
        setChanged(); // la prossima notifica avverrà
        notify(nomi); // notifica i ConcreteObserver
    }
    public Persona find(String cognome) {
        for (Persona p : nomi)
            if (p.getCognome().equals(cognome)) return p;
        System.out.println("AddrBook.find: NOT found");
        return null;
    }
}

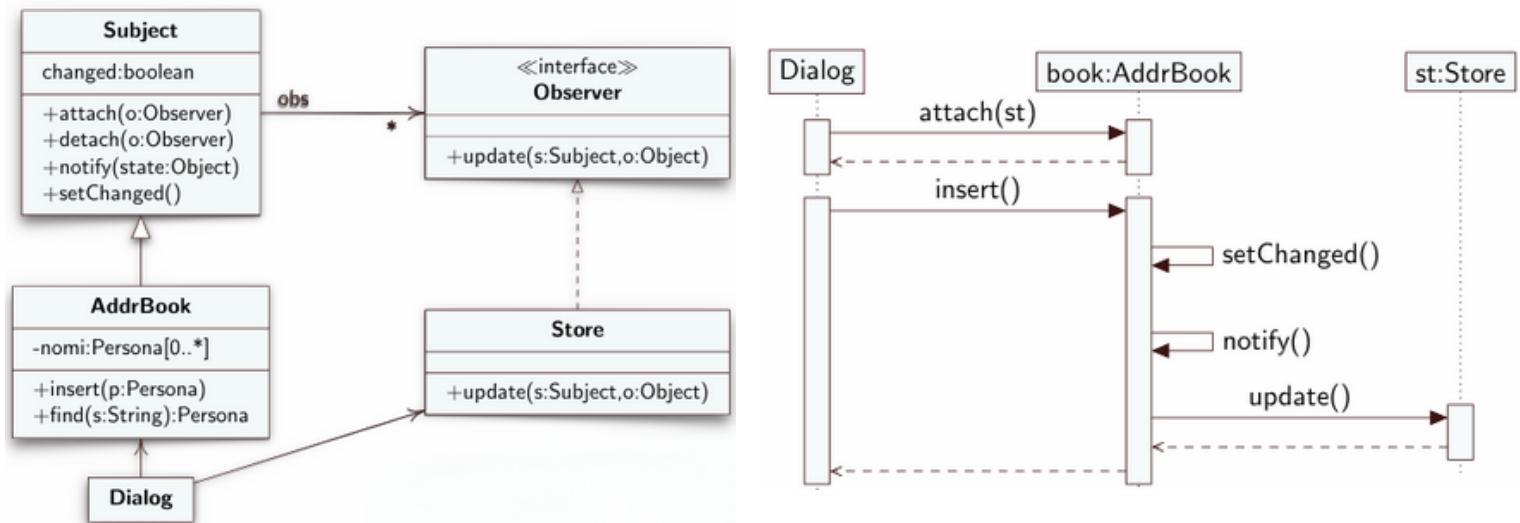
public interface Observer {
    public void update(Subject s, Object o);
}

public class Store implements Observer {
    @Override
    public void update(Subject s, Object o) {
        List<Persona> l = (List<Persona>) o;
        String nom;
        try (FileWriter f = new FileWriter("nomi.txt")) {
            for (Persona p : l) {
                nom = p.getNome() + "\t" + p.getCognome() +
                    "\t" + p.getTelefono();
                f.write(nom + "\n");
            }
        } catch (IOException e) { }
    }
}

public class Dialog {
    private static final AddrBook book =
        new AddrBook();
    private static final Store st = new Store();
    private static final Persona p1 =
        new Persona("Oliver", "Stone", "012345", "NY");

    public static void main(String[] args) {
        book.attach(st);
        book.insert(p1);
    }
}

```



Avvertenze

- Con tanti Subject e pochi Observer, anziché tenere riferimenti a Observer nei Subject, si può usare una sola tabella associativa (incomune fra i Subject) per ridurre lo spazio impegnato
- Un Observer potrebbe osservare più oggetti, per sapere chi ha mandato la notifica, come parametro di update(), si manda il riferimento al Subject
- Subject chiama notify() dopo un cambiamento, oppure aspetta un certo numero di cambiamenti, in modo da evitare continue notifiche agli Observer
- Per mantenere la consistenza fra Observer e Subject, un cambiamento dell'Observer deve essere comunicato al Subject (con setState())
- Quando si vuol eliminare un Subject, gli Observer dovranno essere avvisati per cancellare il loro riferimento al Subject
- Il Subject può passare ulteriori informazioni quando chiama update(), modello push; anziché aspettare che l'Observer legga lo stato, modello pull
- Gli Observer che si registrano possono specificare gli eventi di interesse, in modo che con l'update() si manda solo ciò che è cambiato

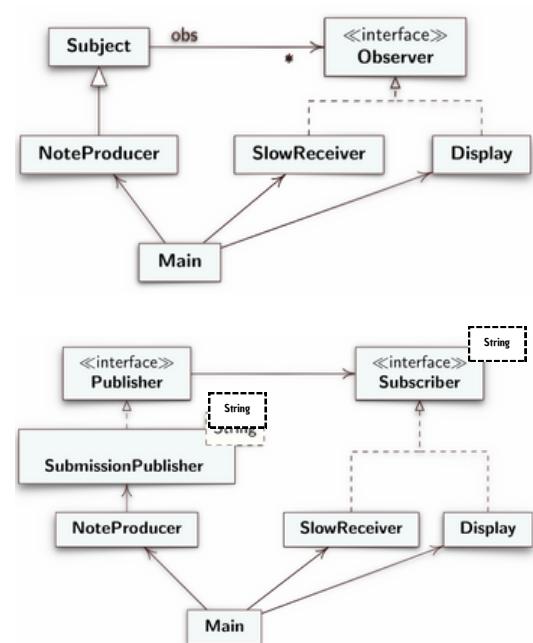
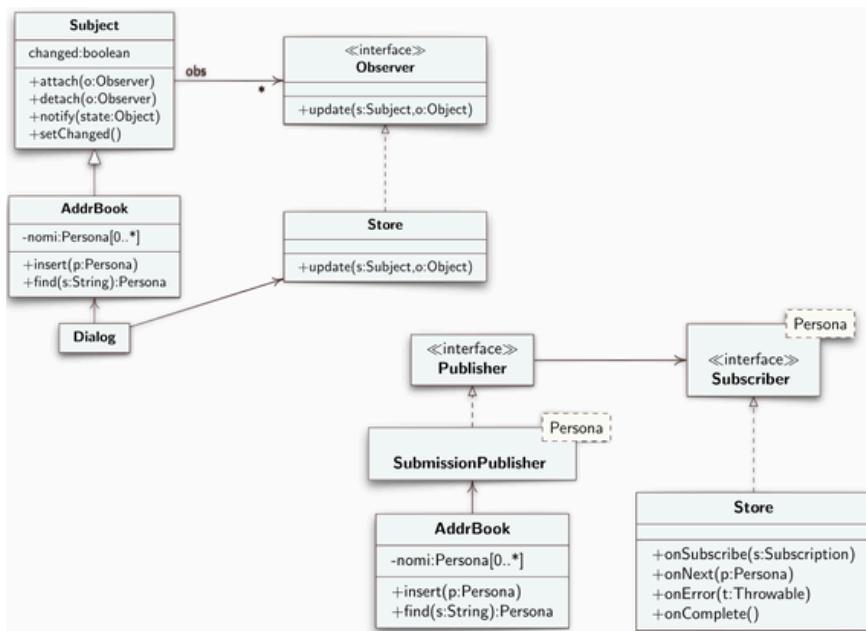
Observer

Observer In Java

- Il problema affrontato dal design pattern Observer è così comune che la sua soluzione è fornita nella libreria java.util, con i tipi Observable e Observer
- Observable svolge il ruolo di Subject quindi tiene traccia di tutti gli oggetti ConcreteObserver che vogliono essere informati di un cambiamento. Observable notifica il cambiamento di stato quando il metodo notifyObservers() viene chiamato
- La classe Observable ha una variabile (flag) che indica se lo stato è cambiato dalla precedente notifica, è impostato dal metodo setChanged(). La chiamata a setChanged() è da effettuare all'interno dei metodi della classe ConcreteSubject secondo la logica desiderata
- Observer è una interfaccia che ha solo il metodo update() e può avere un argomento che indica quale oggetto ha causato l'aggiornamento

Reactive Streams

- Quando il ConcreteSubject chiama notify(), questo chiama update() che esegue sul thread del chiamante, costringendo il chiamante ad aspettare l'esecuzione di update() di ciascun ConcreteObserver
- Più recentemente, con i Reactive Streams, si è cercato di risolvere il problema del passaggio di un insieme di item da un publisher a un subscriber senza bloccare il publisher e senza inondare il subscriber
- Con la versione 9 di Java (settembre 2017), Observable e Observer sono disponibili per compatibilità con versioni precedenti, ma sconsigliati
- Java 9 fornisce nella libreria java.util.concurrent le interfacce Publisher<T>, Subscriber<T>, Subscription, e la classe SubmissionPublisher. Quest'ultima implementa un Publisher dedicando un thread per mandare ciascun messaggio ai Subscriber



Publisher Subscriber Java 9

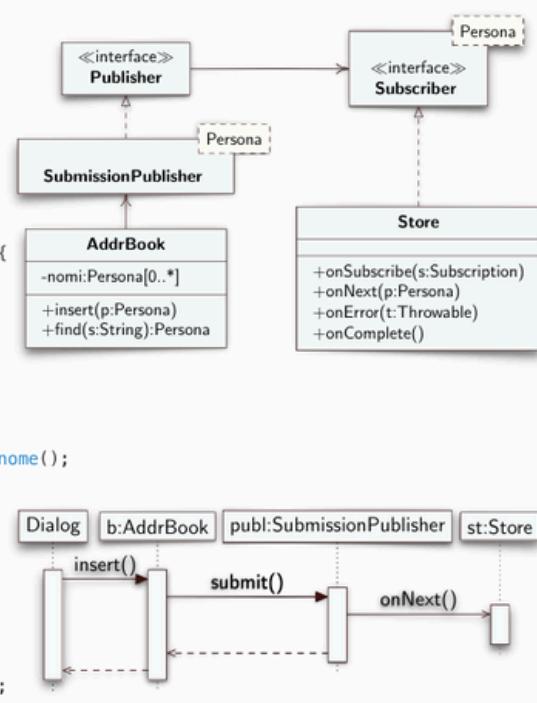
- Publisher** è l'astrazione per fornire item, definisce il metodo subscribe() che prende in input Subscriber<T>
- SubmissionPublisher** implementa Publisher e invia item ai Subscriber in maniera asincrona. Quando sul SubmissionPublisher è chiamato il metodo submit(), esso esegue in un thread dedicato (asincrono) la chiamata al metodo onNext() dei Subscriber, e si blocca se il subscriber non può ricevere l'item
- Subscriber** è usato per ricevere item, definisce i metodi onComplete(), onError(), onNext(), onSubscribe()
- Subscription** definisce il collegamento fra Publisher e Subscriber; i metodi cancel() e request(n) permettono al Subscriber di fermare l'invio di messaggi e di richiedere l'invio dei prossimi n messaggi

L'annotazione '@Override' in Java indica che un metodo sovrscrive un metodo della superclasse. Aiuta a prevenire errori, migliorare la leggibilità e garantire che il metodo esista nella superclasse.

```

public class AddrBook {
    private List<Persona> nomi = new ArrayList<>();
    private SubmissionPublisher<Persona> publ = new SubmissionPublisher<>();
    public void attach(Subscriber<Person> s) {
        publ.subscribe(s);
    }
    public boolean insert(Persona p) {
        if (nomi.contains(p)) return false;
        nomi.add(p);
        publ.submit(p);
        return true;
    }
}

public class Store implements Subscriber<Person> {
    private Subscription sub;
    @Override
    public void onSubscribe(Subscription s) {
        sub = s;
        sub.request(1);
    }
    @Override
    public void onNext(Persona p) {
        String nom = p.getNome() + "\t" + p.getCognome();
        System.out.println("Store onNext: " + nom);
        sub.request(1);
    }
    @Override
    public void onError(Throwable throwable) {
        System.out.println("In Store: errore");
    }
    @Override
    public void onComplete() {
        System.out.println("In Store: completato");
    }
}
  
```



Observer

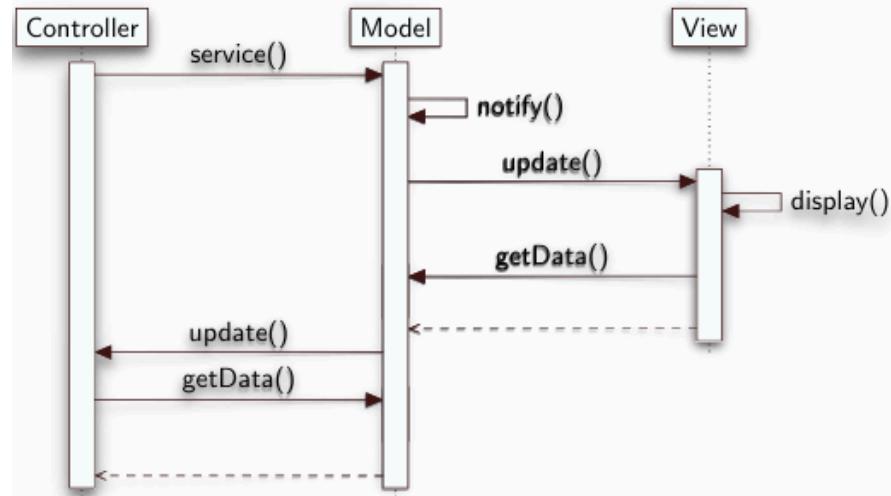
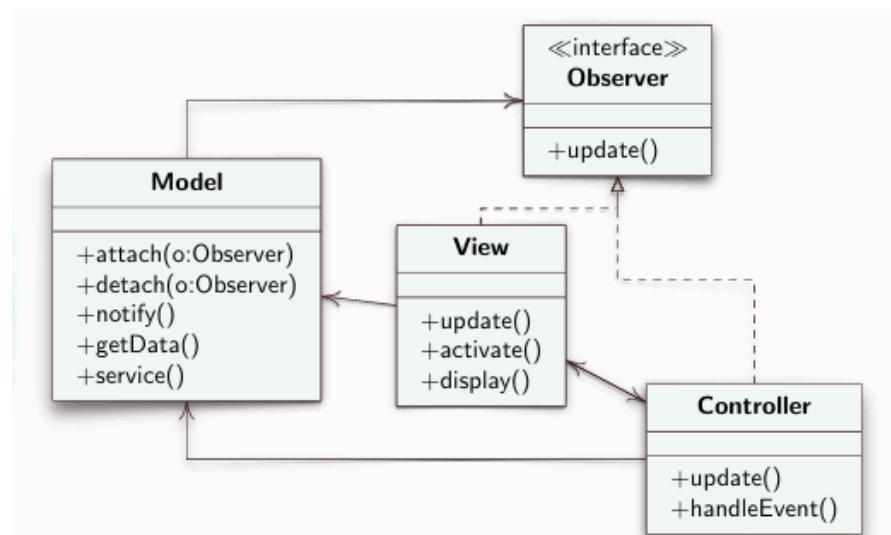
Model View Controller

- E' considerato un **pattern architetturale** per le applicazioni interattive, che individua tre componenti: **Model** per funzionalità principali e dati; **View** per mostrare i dati; **Controller** per prendere gli input dell'utente
- **Motivazioni**
- Le interfacce utente possono cambiare, poiché funzionalità, dispositivi o piattaforme cambiano
- Le stesse informazioni sono presentate in finestre differenti (per es. sotto forma di grafici diversi)
- Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati
- I cambiamenti all'interfaccia utente dovrebbero essere facili
- Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali

Model incapsula le funzionalità principali e i dati dell'applicazione. E' indipendente dalla rappresentazione degli output e dagli input. Registra View e Controller. Avvisa View e Controller registrati dei cambiamenti di dati(tiene dei dati importanti)

View mostra i dati all'utente. Generalmente ci sono tante View, ogni View è associata a un Controller. View inizializza il proprio Controller, e mostra i dati che legge da Model (riesce a visualizzare i dati che gli faccio passare

Controller riceve gli input dell'utente (da mouse e tastiera) sotto forma di eventi. Traduce gli eventi in richieste di servizio per Model o avvisa View (si mette in ascolto dell'input dell'utente e permette di convertire queste interazione in modifiche verso il Model, che potrà prendere quei dati e aggiornare i suoi o lo stato, etc..



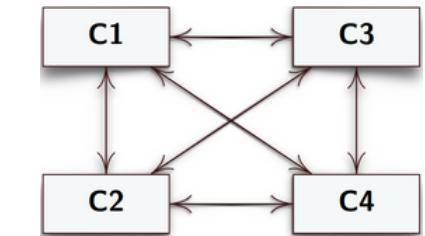
MVC è un "design pattern" (o più comunemente chiamato "pattern architetturale") utilizzatissimo, è simile ed allo stesso tempo un po' diverso rispetto Observer.

Mediator

INTENTO: Definire un oggetto che incapsula come un gruppo di oggetti interagisce. Il Mediator promuove il lasco accoppiamento fra oggetti poiché evita che essi interagiscano direttamente, e permette di modificare le loro interazioni indipendentemente da essi

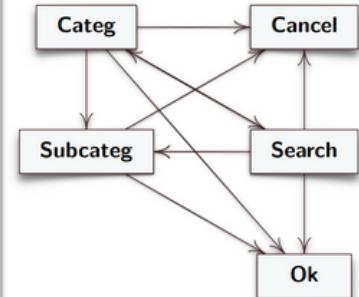
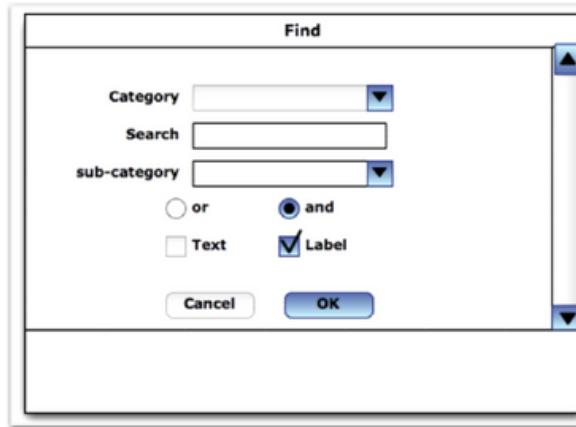
MOTIVAZIONE

- La distribuzione di responsabilità fra vari oggetti può risultare in molte connessioni fra oggetti, nel caso peggiore un oggetto conosce tutti gli altri
- Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si comporta come se fosse monolitico. Inoltre potrebbe essere difficile cambiare il comportamento del sistema poiché il comportamento è distribuito fra oggetti
- Si possono evitare questi problemi incapsulando il comportamento collettivo in un oggetto mediatore separato. Il mediatore serve da intermediario ed evita che gli oggetti dipendano fra loro



ESEMPIO - Finestra mostrata

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

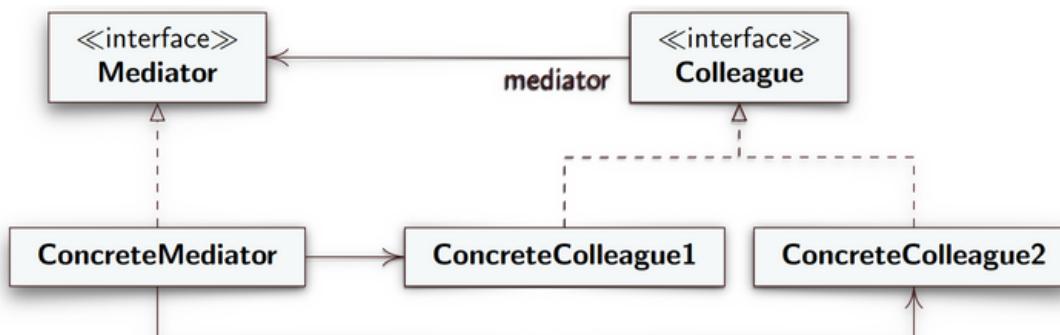


SOLUZIONE

- Isolare le comunicazioni (complesse) tra oggetti dipendenti creando una classe separata per esse
- **Mediator** definisce una interfaccia (punto di incontro) per gli oggetti connessi **Colleague**
- **ConcreteMediator** implementa il comportamento cooperativo e coordina oggetti **Colleague**
- Ogni **Colleague** conosce il **Mediator**, e comunica con il Mediator quando avrebbe comunicato con un altro **Colleague**
- I **ConcreteColleague** mandano richieste, e ricevono richieste, a un oggetto Mediator. Il Mediator implementa il comportamento cooperativo inoltrando le richieste a opportuni **ConcreteColleague**

Quando usare il Mediator?

- Un insieme di oggetti comunicano in modo ben definito ma complesso. Le interdipendenze che ne risultano sono non strutturate e difficili da comprendere
- Riusare un oggetto è difficile poiché esso comunica con tanti altri oggetti
- Un comportamento che è distribuito fra tante classi dovrebbe essere modificabile senza dover ricorrere a sottoclassi



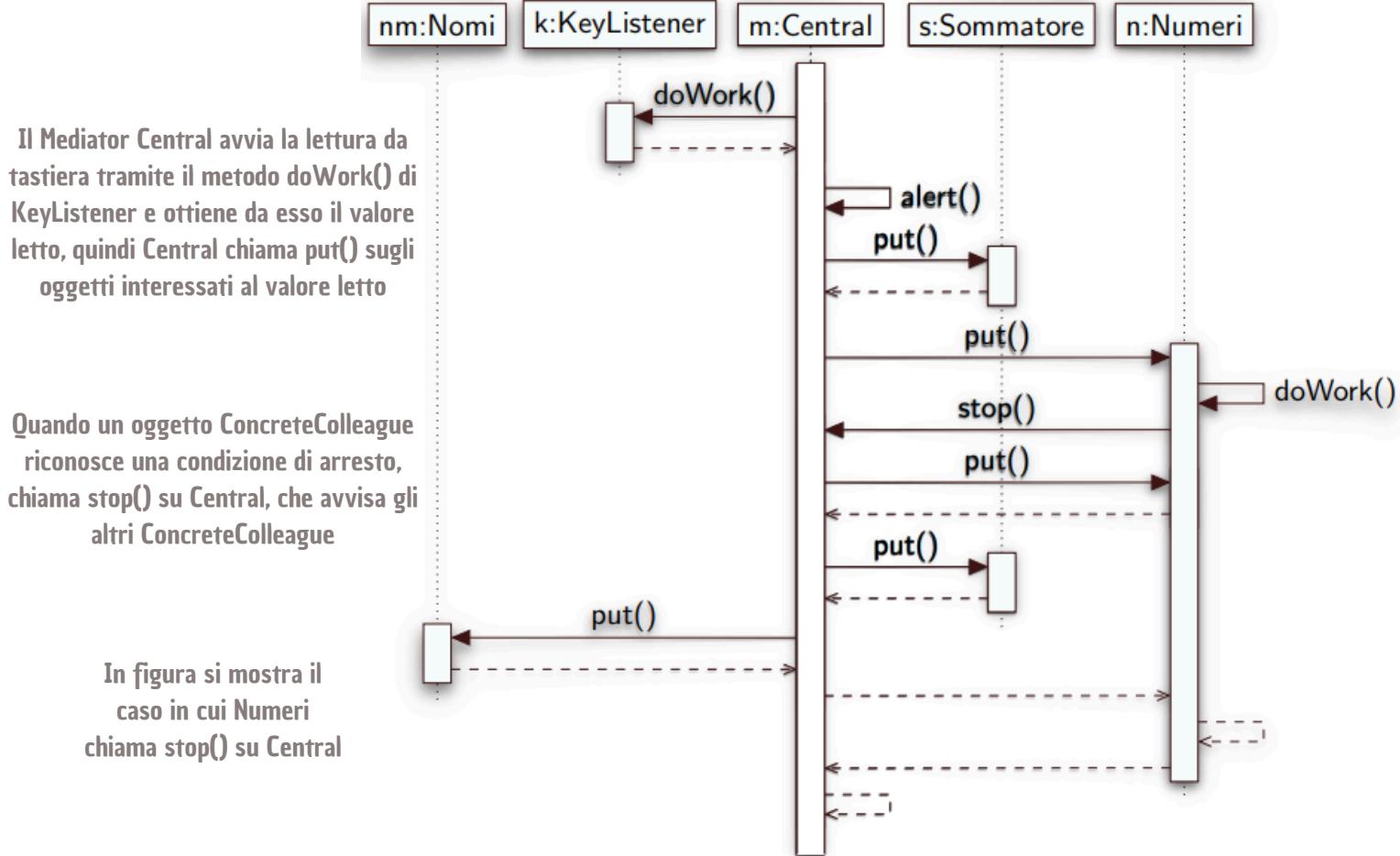
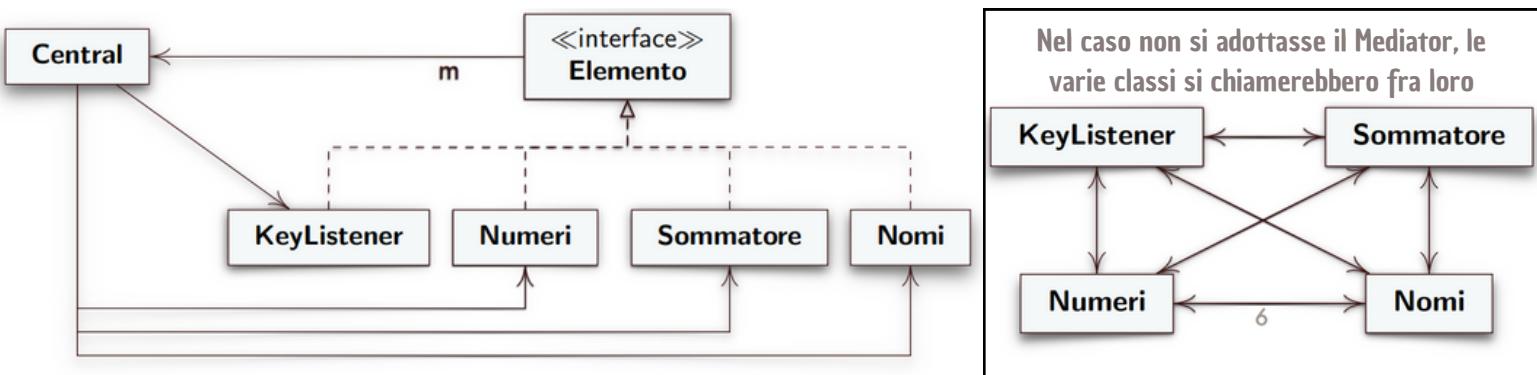
Mediator

CONSEGUENZE

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

ESEMPIO

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



Command

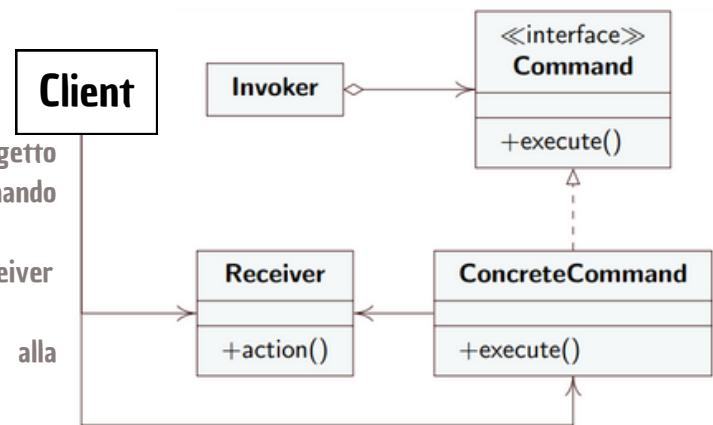
INTENTO: incapsulare una richiesta in un oggetto, permettendo quindi di parametrizzare i client con differenti richieste, code o log di richieste, e supportare l'annullamento di operazioni

MOTIVAZIONE

- Qualche volta può essere necessario mandare una richiesta a oggetti senza conoscere alcunché dell'operazione richiesta o del ricevente della richiesta
- La richiesta viene trasformata in oggetto e quest'oggetto può essere immagazzinato o passato come altri oggetti
- La chiave di questo pattern è una classe astratta Command che dichiara un'interfaccia per eseguire operazioni. Le sottoclassi specificano la coppia ricevente e azione, immagazzinando il ricevente come attributo e implementando il metodo che effettua le richieste. Il ricevente conosce ciò che è necessario per servire la richiesta

Partecipanti

- **Command** definisce un'interfaccia per eseguire un'operazione
- **ConcreteCommand** definisce il collegamento fra un oggetto Receiver e un'azione; implementa l'operazione execute chiamando le corrispondenti operazioni sul Receiver
- **Client** crea un oggetto ConcreteCommand e imposta il suo Receiver
- **Invoker** chiede al comando di effettuare la richiesta
- **Receiver** conosce come effettuare le operazioni associate alla richiesta. Qualunque classe può essere un Receiver



Collaborazioni

- Il client crea un oggetto ConcreteCommand e specifica il suo oggetto Receiver
- Un oggetto Invoker tiene l'oggetto ConcreteCommand
- L'oggetto Invoker effettua una richiesta chiamando execute() sull'oggetto Command. Quando i comandi sono reversibili, il ConcreteCommand conserva lo stato per cancellare il comando e ritornare a prima della chiamata a execute()
- L'oggetto ConcreteCommand chiama le operazioni sul suo Receiver per portare a termine la richiesta

Conseguenze

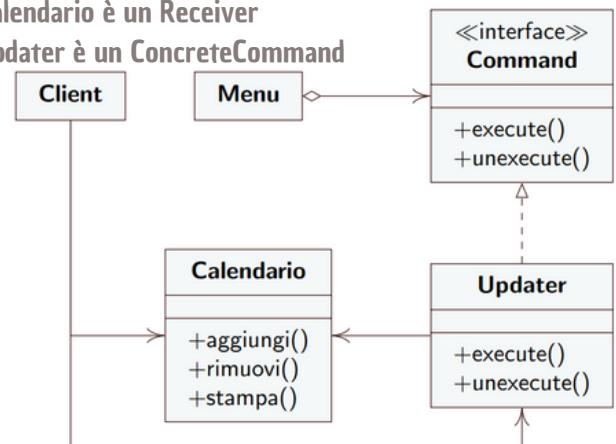
- Il pattern Command disaccoppia l'oggetto che invoca l'operazione da quello che conosce come effettuarla
- I comandi diventano oggetti e possono essere manipolati ed estesi come altri oggetti
- Si possono assemblare comandi in un comando composto (MacroCommand). In generale, i comandi composti sono un'istanza del design pattern Composite
- E' facile aggiungere nuovi comandi, poiché non bisogna cambiare classi esistenti

Implementazione

- Un Command può avere un'ampia gamma di abilità. Potrebbe solo definire il legame con un ricevente e l'azione da eseguire. Oppure, implementare tutto facendo così a meno del ricevente
- I Command possono supportare undo se hanno la capacità di rendere reversibile la loro esecuzione. Per questo la classe ConcreteCommand potrebbe aver bisogno di conservare uno stato aggiuntivo. Il ricevente deve fornire le operazioni che permettono al command di far tornare il ricevente allo stato precedente
- Per un livello di undo il command deve conservare l'ultima attività eseguita, per più livelli di undo (e redo), sarà necessario avere una lista di comandi eseguiti

Esempio

- Si vuol avere un calendario e un menù per aggiungere o togliere note sul calendario
- Menu è un Invoker
- Calendario è un Receiver
- Updater è un ConcreteCommand



Chain Of Responsibility

INTENTO: evitare di accoppiare il mandante di una richiesta col suo ricevente, dando così a più di un oggetto la possibilità di gestire la richiesta. Concatena gli oggetti riceventi e passa la richiesta lungo la catena fino a che un oggetto la gestisce

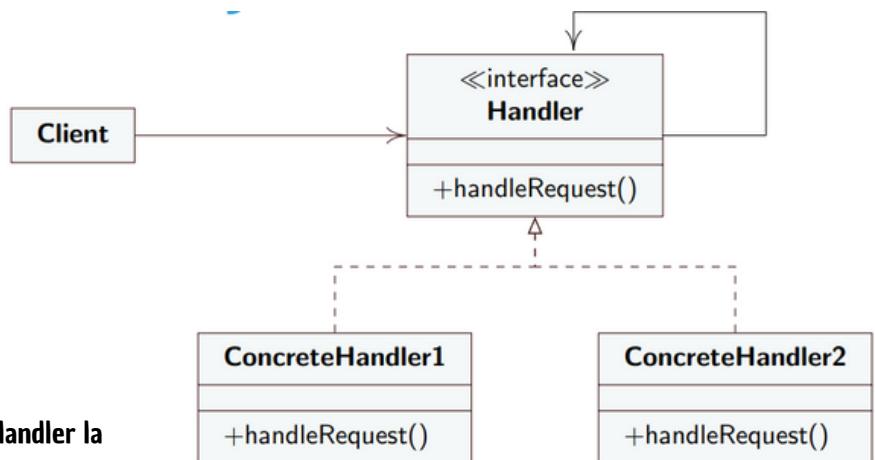
MOTIVAZIONE

- Consideriamo un'interfaccia utente in cui l'utente può chiedere aiuto su parti dell'interfaccia. Per es. un bottone può fornire informazioni di aiuto. Se non esiste un'informazione specifica allora si dovrebbe fornire il messaggio di aiuto del contesto più vicino (per es. la finestra)
- Problema: l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto che effettua la richiesta iniziale
- Disaccoppiare mandante e ricevente

PARTECIPANTI

- **Handler** definisce l'interfaccia per gestire le richieste
- **ConcreteHandler** gestisce la richiesta di cui è responsabile, può accedere al suo successore, inoltra la richiesta se non può gestirla
- **Client** inizia effettuando la richiesta a un **ConcreteHandler**

La richiesta si propaga lungo la catena finché un **ConcreteHandler** la gestisce



CONSEGUENZE

- Riduce l'accoppiamento: chi effettua una richiesta non conosce il ricevente
- Un oggetto della catena non deve conoscere la struttura della catena
- Gli oggetti handler mantengono solo il riferimento al successore (non a una lista di oggetti)
- Si aggiunge flessibilità nella distribuzione di responsabilità agli oggetti. Si può cambiare o aggiungere responsabilità nella gestione di una richiesta cambiando la catena a runtime
- Non vi è garanzia che una richiesta sia gestita, poiché non c'è un ricevente esplicito, la richiesta potrebbe arrivare alla fine della catena senza essere gestita

IMPLEMENTAZIONE

- Handler o ConcreteHandler possono definire i link per avere il successore della catena
- Una gerarchia già esistente può essere usata, anziché ridefinire i link (il design pattern Composite è usato insieme a Chain)
- Ciascun tipo di richiesta può corrispondere a un'operazione, in questo caso il set di richieste è definito dall'Handler, oppure
- Ciascun tipo di richiesta corrisponde a un codice e solo un'operazione è definita dall'Handler. Il set di richieste non è fissato: richiedenti e riceventi prendono accordi sulla codifica delle richieste; si hanno più controlli a runtime

```

public interface Handler {
    public void handleRequest();
    public void activate();
    public void setNext(Handler next);
}

public class Text implements Handler {
    private Handler next;
    public Text(Handler h) {
        next = h;
    }
    @Override public void handleRequest() {
        if (next != null) next.handleRequest();
        else System.out.println("Text: assistenza");
    }
    @Override public void activate() {
        System.out.println("sono un frammento di testo");
    }
    @Override public void setNext(Handler next) {
        this.next = next;
    }
}

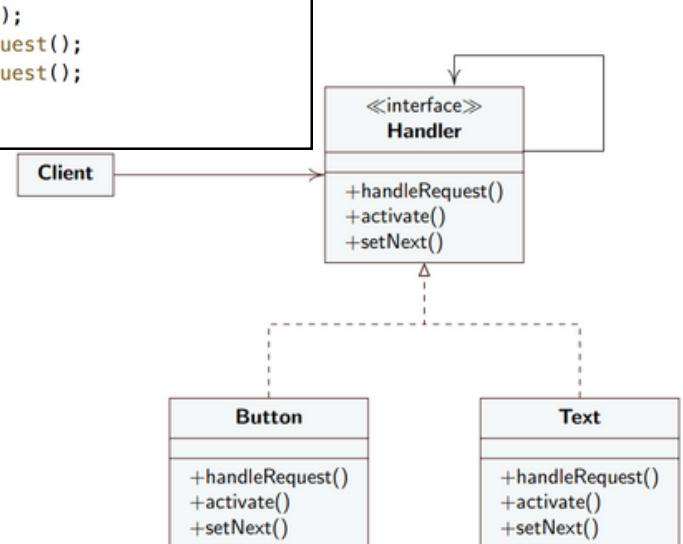
```

```

public class Client {
    public static void main(String[] args) {
        Text t = new Text(null);
        Button b = new Button(t);
        t.activate();
        b.activate();
        b.handleRequest();
        b.handleRequest();
    }
}

```

Button e Text sono **ConcreteHandler**



Bridge

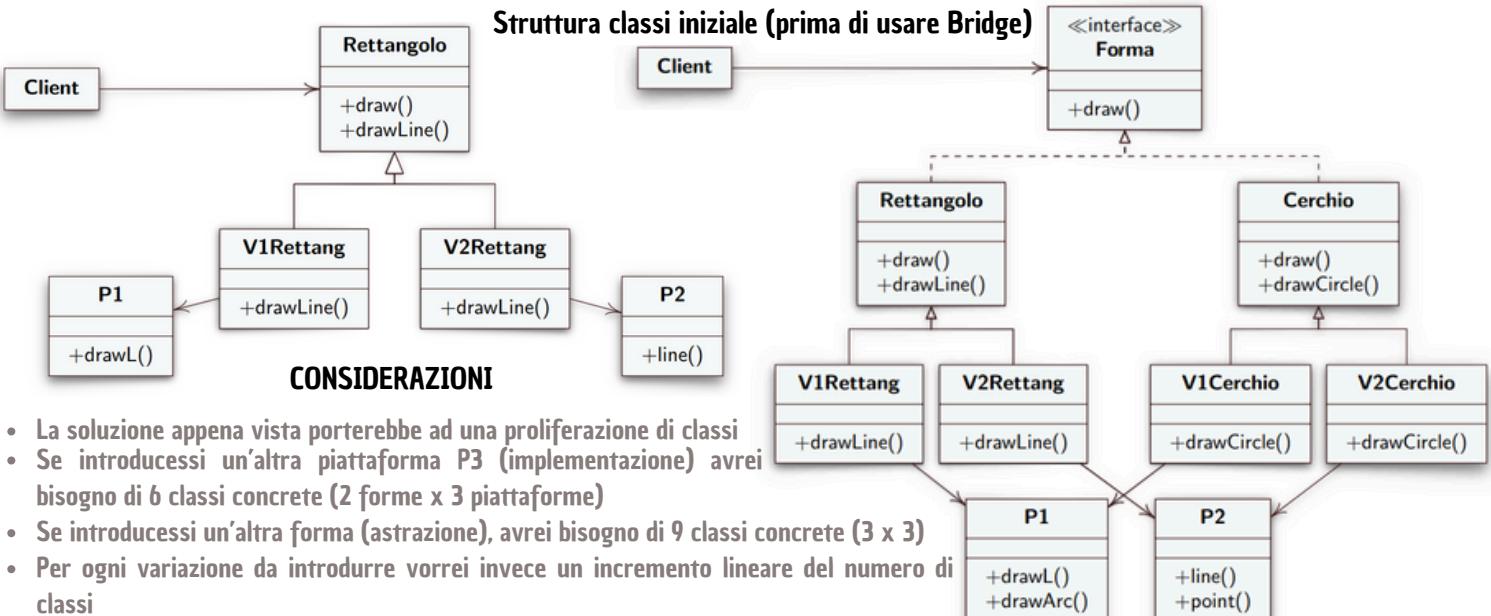
INTENTO: Disaccoppiare una astrazione dalla sua implementazione così che le due possano variare indipendentemente

MOTIVAZIONE

- Quando una astrazione può avere varie implementazioni, di solito si usa l'ereditarietà. Una classe astratta definisce l'interfaccia dell'astrazione, le sottoclassi concrete la implementano in modi diversi
- Questo approccio non è flessibile poiché collega l'astrazione all'implementazione permanentemente, e rende difficile modificare, estendere e usare astrazioni e implementazioni indipendentemente

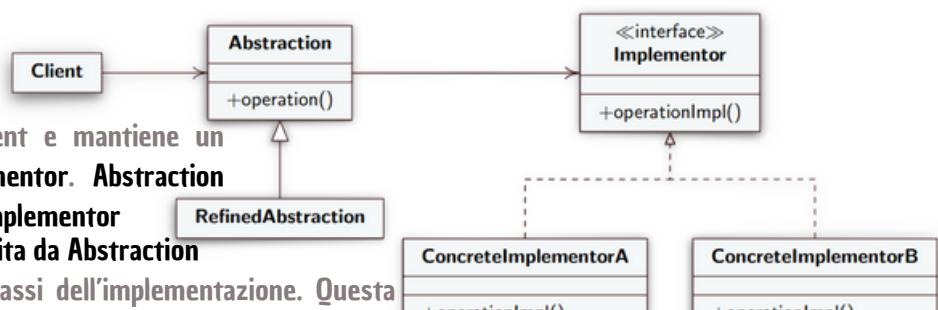
ESEMPIO

- In un sistema, occorre avere Rettangoli e Cerchi (astrazioni). Inoltre, occorre che Rettangoli e Cerchi siano disponibili per due piattaforme P1 e P2 (implementazioni)
- La piattaforma P1 mette a disposizione drawL() per disegnare linee, mentre la piattaforma P2 fornisce line()



SOLUZIONE

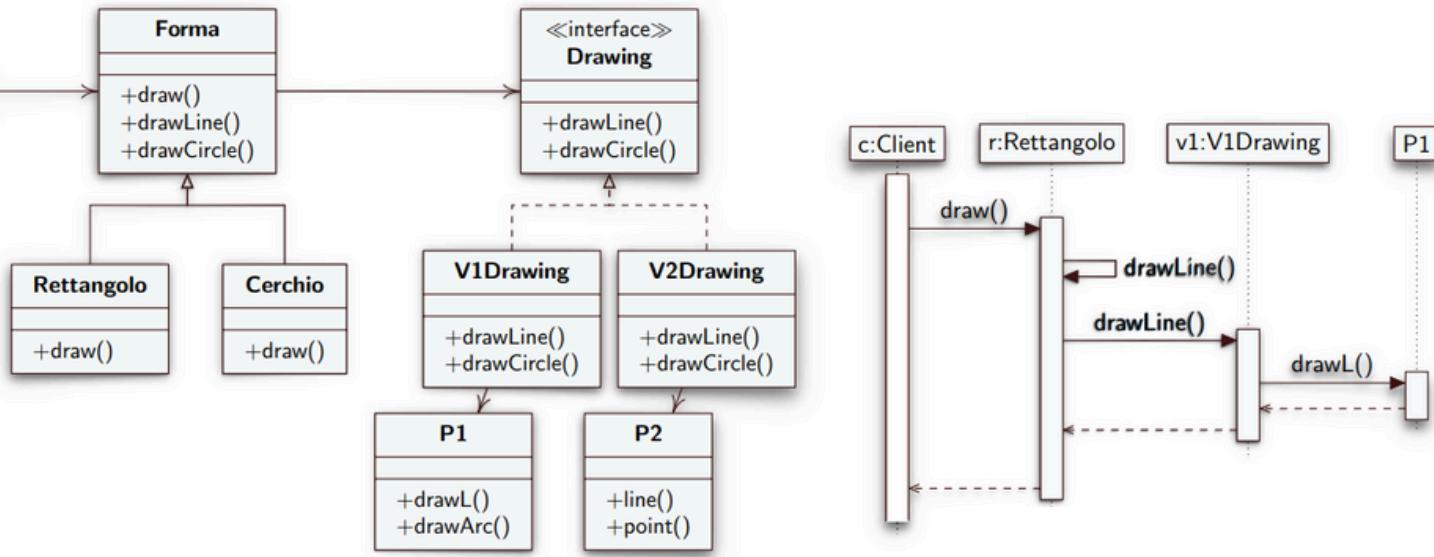
STRUTTURA



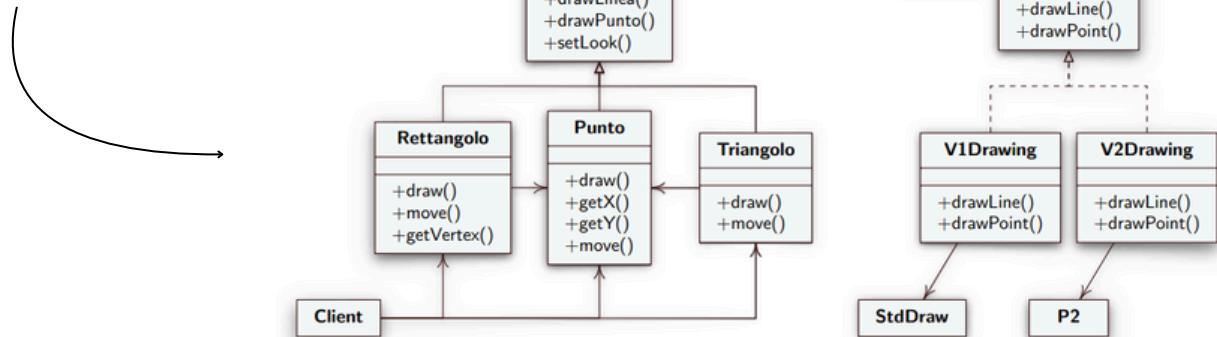
- **Abstraction** definisce l'interfaccia per i client e mantiene un riferimento ad un oggetto di tipo **Implementor**. **Abstraction** inoltra le richieste del client al suo oggetto **Implementor**
- **RefinedAbstraction** estende l'interfaccia definita da **Abstraction**
- **Implementor** definisce l'interfaccia per le classi dell'implementazione. Questa interfaccia non deve corrispondere esattamente ad **Abstraction**, di solito **Implementor** fornisce operazioni primitive, mentre **Abstraction** definisce operazioni di più alto livello basate su tali primitive
- **ConcreteImplementor** implementa l'interfaccia di **Implementor** e fornisce le operazioni concret

Bridge

- La soluzione suggerita dal design pattern Bridge, per il suddetto esempio, avrà il diagramma delle classi disegnato sotto
- Rettangolo.draw() chiama drawLine() della superclasse Forma, quest'ultima chiama drawLine() su un'istanza di una sottoclasse di Drawing. Analoghe chiamate avvengono per Cerchio
- Una nuova classe, es. Rombo, sarà implementata come sottoclasse di Forma. La classe Rombo chiamerà drawLine() su Forma. Non occorre nessuna nuova classe della gerarchia Implementor



Esempio Minimale Del Bridge



Conseguenze

- Bridge permette a una implementazione di non essere connessa permanentemente a una interfaccia, l'implementazione può essere configurata e anche cambiata a runtime
- Il disaccoppiamento permette di cambiare l'implementazione senza dover ricompilare Abstraction ed i Client
- Solo certi strati del software devono conoscere Abstraction e Implementor
- I Client non devono conoscere Implementor
- Le gerarchie di Abstraction e Implementor possono evolvere in modo indipendente

```

// Forma è una Abstraction
public class Forma {
    private Drawing impl;
    public void setImplementor(Drawing imp) { this.impl = imp; }
    public void drawLine(int x, int y, int z, int t) {
        impl.drawLine(x, y, z, t);
    }
}

// Drawing è un Implementor
public interface Drawing {
    public void drawLine(int x1, int y1, int x2, int y2);
}

// Rettangolo è una RefinedAbstraction
public class Rettangolo extends Forma {
    private int a, b, c, d;
    public Rettangolo(int xi, int yi, int xf, int yf) {
        a = xi; b = yi; c = xf; d = yf;
    }
    public void draw() {
        drawLine(a, b, c, b); drawLine(a, b, a, d);
        drawLine(c, b, c, d); drawLine(a, d, c, d);
    }
}

```

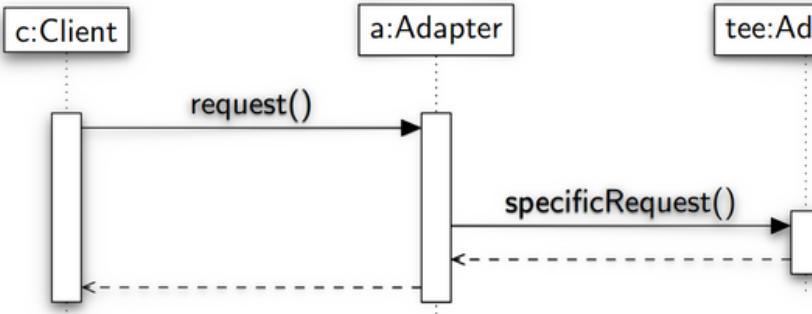
Adapter

INTENTO: Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano. Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili.

PROBLEMA

- Alcune volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia che si aspetta l'applicazione. Ovvero nome metodo, parametri, tipo parametri di chiamate all'interno dell'applicazione non sono corrispondenti a quelli offerti da una classe di libreria
- Non è possibile cambiare l'interfaccia della libreria, poiché non si ha il sorgente (comunque non conviene cambiarla)
- Non è possibile cambiare l'applicazione, e si può voler cambiare quale metodo invocare, senza renderlo noto al chiamante

Diagramma di sequenza della soluzione Object Adapter



```

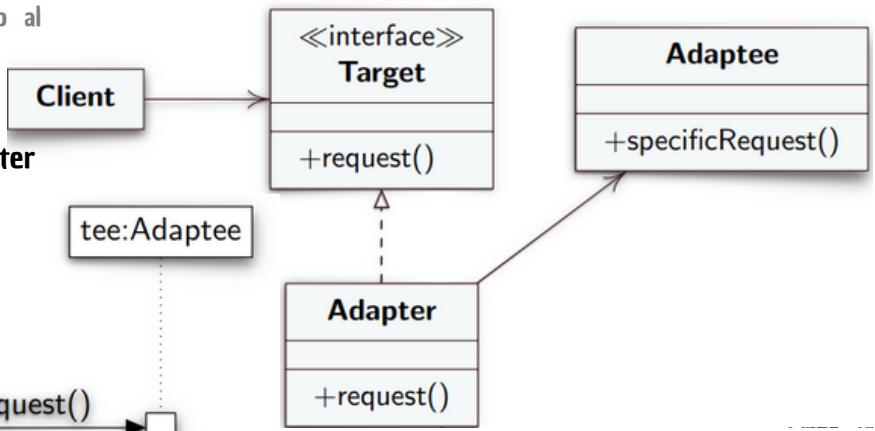
public interface ILabel { // Target
    public String getNextLabel();
}

// Adapter
public class Label implements ILabel {
    private LabelServer ls;
    private String p;
    public Label(String prefix) {
        p = prefix;
    }
    public String getNextLabel() {
        if (ls == null)
            ls = new LabelServer(p);
        return ls.serveNextLabel();
    }
}

public class Client {
    public static void main(String args[]) {
        ILabel s = new Label("LAB");
        String l = s.getNextLabel();
        if (l.equals("LAB1"))
            System.out.println("Test 1:Passed");
        else
            System.out.println("Test1:Failed");
    }
}
  
```

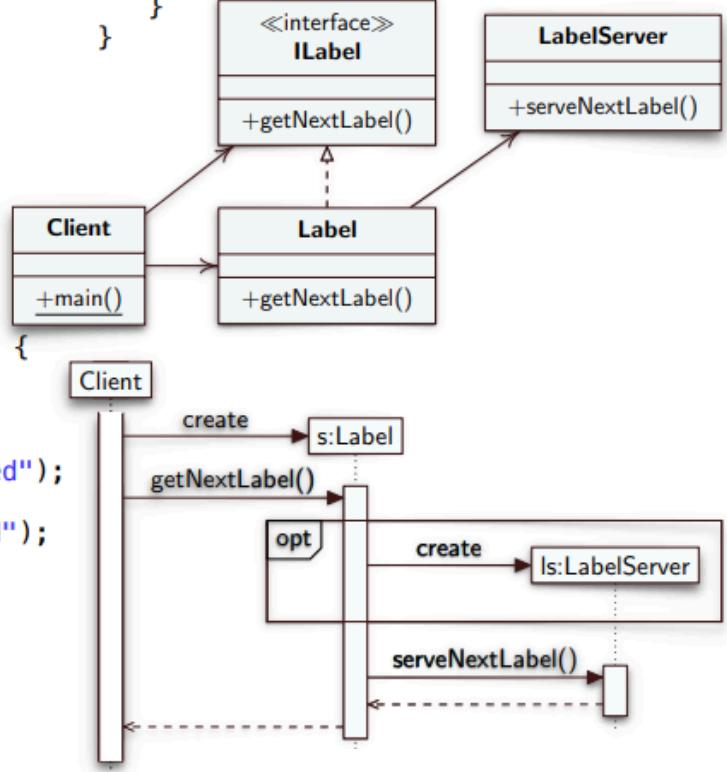
SOLUZIONE OBJECT ADAPTER

- **Target** è l'interfaccia che il chiamante si aspetta
- **Adaptee** è l'oggetto di libreria
- **Adapter** converte, ovvero adatta, la chiamata che fa una classe client all'interfaccia della classe di libreria. Il chiamante usa l'Adapter come se fosse l'oggetto di libreria. Adapter tiene il riferimento all'oggetto di libreria (Adaptee) e sa come invocarlo, ovvero implementa le chiamate verso i metodi di Adaptee



```

public class LabelServer { // Adaptee
    private int labelNum = 1;
    private String labelPrefix;
    public LabelServer(String prefix) {
        labelPrefix = prefix;
    }
    public String serveNextLabel() {
        return labelPrefix + labelNum++;
    }
}
  
```



Adapter

- Variante Adapter a due vie

- Definizione: la classe Adapter fornisce l'interfaccia di Target e l'interfaccia di Adaptee. Realizzazione: la soluzione Class Adapter è un Adapter a due vie
- Conseguenze del design pattern Adapter
 - Client e classe di libreria Adaptee rimangono indipendenti. L'Adapter può cambiare il comportamento dell'Adaptee
 - Può aggiungere test di precondizioni e postcondizioni [Precondizioni: cosa si deve soddisfare prima di eseguire. Postcondizioni: cosa è verificato se tutto è andato bene]
 - L'Object Adapter può implementare la tecnica di Lazy Initialization
 - Il design pattern Adapter aggiunge un livello di indirezione. Ogni invocazione del client ne scatena un'altra fatta dall'Adapter. Possibile rallentamento (trascurabile), e codice più difficile da comprendere

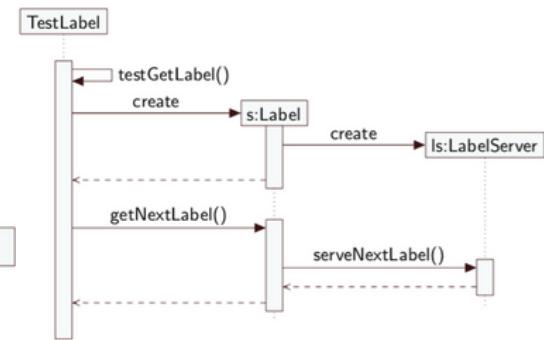
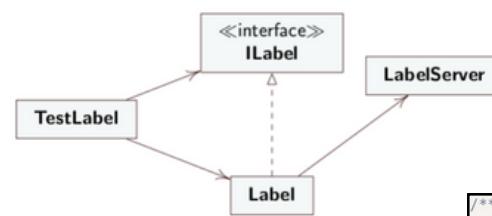
Soluzione Class Adapter

- Adapter è sottoclasse di Adaptee

```
public class Label extends LabelServer implements ILabel { // Adapter

    public Label(String prefix) {
        super(prefix);
    }

    public String getNextLabel() {
        return serveNextLabel();
    }
}
```



Codice Java che implementa il design pattern Adapter

```
/** ILabel e' un Target */
public interface ILabel {
    public String getNextLabel();
    public boolean checkUsed(int k);
    public void insertTag(String s);
}
```

```
/* LabelServer e' un Adaptee */
public class LabelServer {
    private int num = 1;
    private String prefix;

    public LabelServer(String p) {
        prefix = p;
    }

    public String serveNextLabel() {
        return prefix + num++;
    }

    public int getCount() {
        return num;
    }

    public void change(String s) {
        prefix = s;
    }
}
```

```
import java.util.Arrays;
import java.util.List;

/** Label e' un Adapter */
public class Label implements ILabel {
    private List<String> l = Arrays.asList("cat", "dog", "sheep");
    private LabelServer ls;
    private String p;

    // si instanzia subito un Adaptee
    public Label(String prefix) {
        p = prefix;
        ls = new LabelServer(p);
    }

    // l'adattamento consiste nel chiamare un metodo con nome diverso sull'Adaptee
    @Override
    public String getNextLabel() {
        return ls.serveNextLabel();
    }

    // l'adattamento consiste nel fornire una funzionalita' diversa rispetto a quella del
    // metodo sull'Adaptee, che implementa solo parzialmente quanto richiesto dal client
    @Override
    public boolean checkUsed(int k) {
        return (ls.getCount() >= k);
    }

    // qui, oltre a chiamare il corrispondente metodo dell'Adaptee, si verifica, tramite
    // la prima condizione sul corpo del metodo, che la precondizione sia soddisfatta,
    // ovvero non cambiare l'etichetta se non si usa un valore fra quelli permessi
    @Override
    public void insertTag(String t) {
        if (l.contains(t)) ls.change(t);
    }
}
```

```
public class TestLabel {
    static public void main(String args[]) {
        testGetLabel();
        testChangeLabel();
        testNoChangeLabel();
        testUsed();
    }

    private static void testGetLabel() {
        ILabel s = new Label("LAB");
        if (s.getNextLabel().equals("LAB1")) System.out.println("OK Test get label");
        else System.out.println("FAILED Test get label");
    }

    private static void testChangeLabel() {
        ILabel s = new Label("LAB");
        s.insertTag("cat");
        if (s.getNextLabel().equals("cat1")) System.out.println("OK Test change label");
        else System.out.println("FAILED Test change label");
    }

    private static void testNoChangeLabel() {
        ILabel s = new Label("LAB");
        s.insertTag("zebra");
        if (s.getNextLabel().equals("LAB1")) System.out.println("OK Test no-change label");
        else System.out.println("FAILED Test no-change label");
    }

    private static void testUsed() {
        ILabel s = new Label("LAB");
        if (s.checkUsed(1) && !s.checkUsed(2)) System.out.println("OK Test used 1");
        else System.out.println("FAILED Test used 1");
        s.getNextLabel();
        if (s.checkUsed(2) && !s.checkUsed(3)) System.out.println("OK Test used 2");
        else System.out.println("FAILED Test used 2");
    }
}
```

Decorator

INTENTO: Aggiungere ulteriori responsabilità ad un oggetto dinamicamente. I decorator forniscono un'alternativa flessibile all'implementazione di sottoclassi per estendere funzionalità

MOTIVAZIONE

- Alcune volte si vogliono aggiungere responsabilità a singoli oggetti, e non all'intera classe. Per aggiungere responsabilità si potrebbe far uso dell'ereditarietà, tuttavia non si avrebbe flessibilità: un client non può controllare come e quando decorare un componente
- Un approccio più flessibile è racchiudere (**wrap**) il componente in un altro oggetto che aggiunge una responsabilità. L'oggetto wrapper è un decorator. Il **decorator è conforme all'interfaccia che decora**, quindi la sua presenza è trasparente agli oggetti che usano il componente
- Le responsabilità possono essere sottratte dinamicamente. I decorator possono essere annidati ricorsivamente, per aggiungere più 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 1

ESEMPIO

- Si vuol aggiungere la proprietà Bordo ad un componente Testo
- Se si eredita Testo dalla classe Bordo, gli oggetti della sottoclasse avranno questa proprietà, ma non si avrà flessibilità: non si possono avere oggetti senza tale proprietà
- Alternativa flessibile, inserire il componente Testo dentro un oggetto che aggiunge Bordo
- L'oggetto che racchiude, chiamato DecoratorBordo, manda la richiesta al componente Testo e aggiunge altre attività prima o dopo l'invio della richiesta



SOLUZIONE

- **Component** definisce l'interfaccia per gli oggetti che possono avere aggiunte le responsabilità dinamicamente
- **ConcreteComponent** definisce un oggetto su cui poter aggiungere responsabilità
- **Decorator** mantiene un riferimento a un oggetto Component e definisce un'interfaccia conforme a quella di Component. Decorator inoltra le richieste al suo oggetto Component e può fare altre operazioni prima e dopo l'inoltro della richiesta
- **ConcreteDecorator** implementa la responsabilità aggiunta al Component

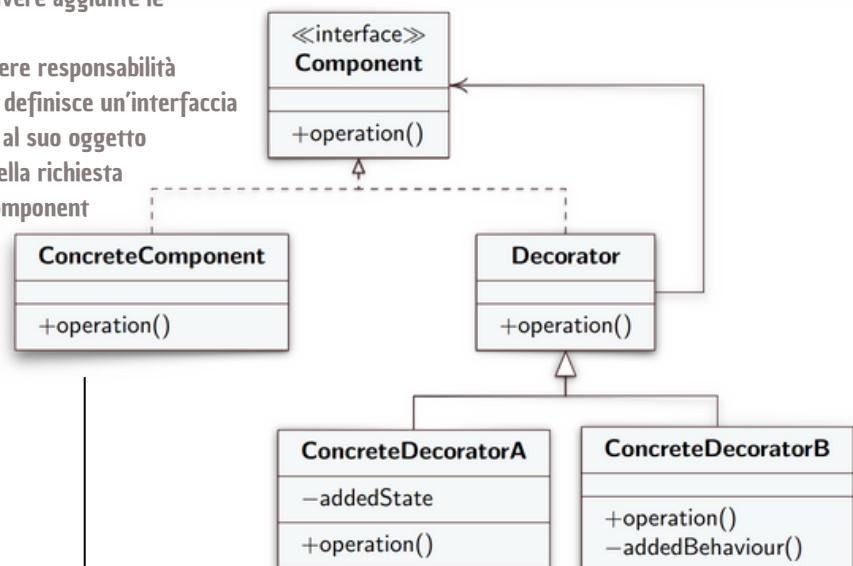
```

interface Component {
    public void operation();
}

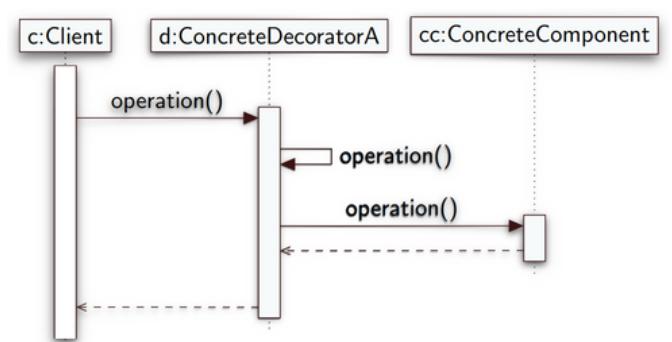
class ConcreteComponent implements Component {
    @Override public void operation() {
        // ...
    }
}

class Decorator implements Component {
    private final Component innerC;
    public Decorator(final Component c) {
        innerC = c;
    }
    @Override public void operation() {
        innerC.operation();
    }
}

class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component c) {
        super(c);
    }
    @Override public void operation() {
        super.operation();
        // ...
    }
}
  
```



operation() in decorator serve solo per "propagare" la chiamata

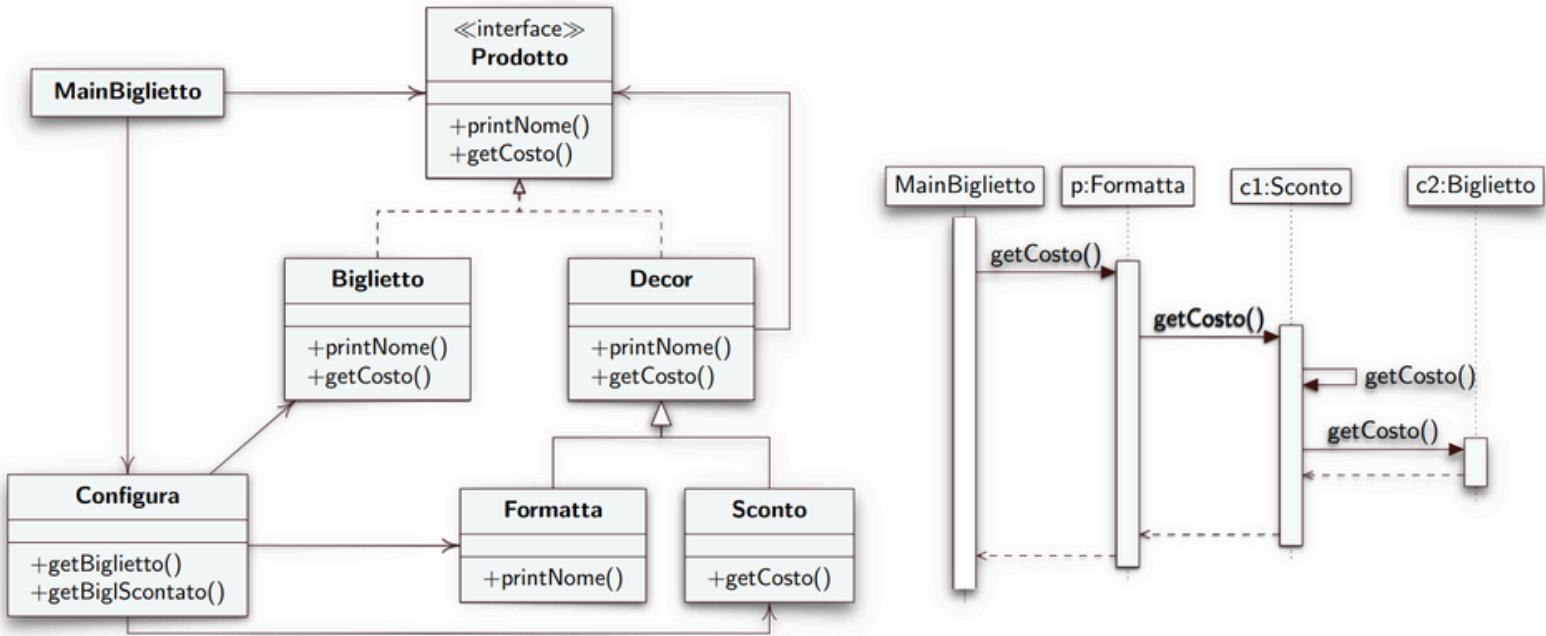


Component c = new ConcreteDecorator(new ConcreteComponent());

Decorator

ESEMPIO

- Vi sono alcuni prodotti, ad es. Biglietto, ognuno con un nome ed un costo; e si hanno diversi modi di calcolare il costo dei prodotti, in base agli sconti e diversi modi di stampare i dettagli
- Si vuol poter combinare a runtime sconti (Sconto) e messaggi (Formatta) sui prodotti, per singole istanze di Biglietto

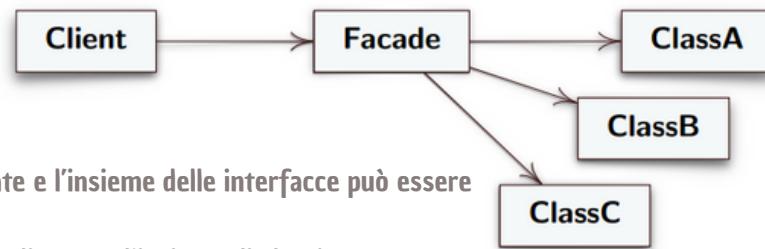


CONSEGUENZE

- Più flessibilità rispetto all'ereditarietà poiché si possono aggiungere responsabilità dinamicamente
- La stessa responsabilità può essere aggiunta più volte, semplicemente aggiungendo due istanze dello stesso **ConcreteDecorator**
- Prevedere per le classi in alto nella gerarchia tutte le responsabilità che servono significherebbe avere per esse troppe responsabilità. I **ConcreteDecorator** sono invece indipendenti e permettono di aggiungere responsabilità successivamente
- L'identità (tipo e riferimento) del **ConcreteDecorator** non è quella del **ConcreteComponent**, quindi non si dovrebbero confrontare i riferimenti
- Si avranno tanti piccoli oggetti, che differiscono nel modo in cui sono interconnessi

Facade

INTENTO: Fornire un'interfaccia unificata al posto di un insieme di interfacce in un sottosistema (consistente di un insieme di classi). Definire un'interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare



PROBLEMA

- Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complesso
- Può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi
- Si vogliono ridurre le comunicazioni e le dipendenze dirette fra i client ed il sottosistema

SOLUZIONE

- **Facade** fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate. **Facade** invoca i metodi degli oggetti che nasconde
- **Client** interagisce solo con l'oggetto **Facade**

CONSEGUENZE

- Nasconde ai client l'implementazione del sottosistema
- Promuove l'accoppiamento debole tra sottosistema e client
- Riduce le dipendenze di compilazione in sistemi grandi. Se si cambia una classe del sottosistema, si può ricompilare la parte di sottosistema fino al facade, quindi non i vari client
- Non previene l'uso di client più complessi, quando occorre, che accedono ad oggetti del sottosistema

IMPLEMENTAZIONE: Per rendere gli oggetti del sottosistema non accessibili al client le corrispondenti classi possono essere annidate dentro la classe **Facade**

```

public class English {
    private String text = " ";
    private List<String> d =
        Arrays.asList("Alright", "Hello",
                     "Understood", "Yes");

    public boolean test(String s) {
        return d.contains(s);
    }

    public void add(String s) {
        text = text + " " + s;
    }

    public String getText() {
        return text;
    }

    public int getIndex(String s) {
        return d.indexOf(s);
    }

    public void printText() {
        System.out.println(text);
    }
}

public class Italian {
    private String text = " ";
    private List<String> d =
        Arrays.asList("Va bene", "Ciao",
                     "Capito", "Sì");

    public void add(int i) {
        text = text + " " + d.get(i);
    }

    public void printText() {
        System.out.println(text);
    }
}
  
```

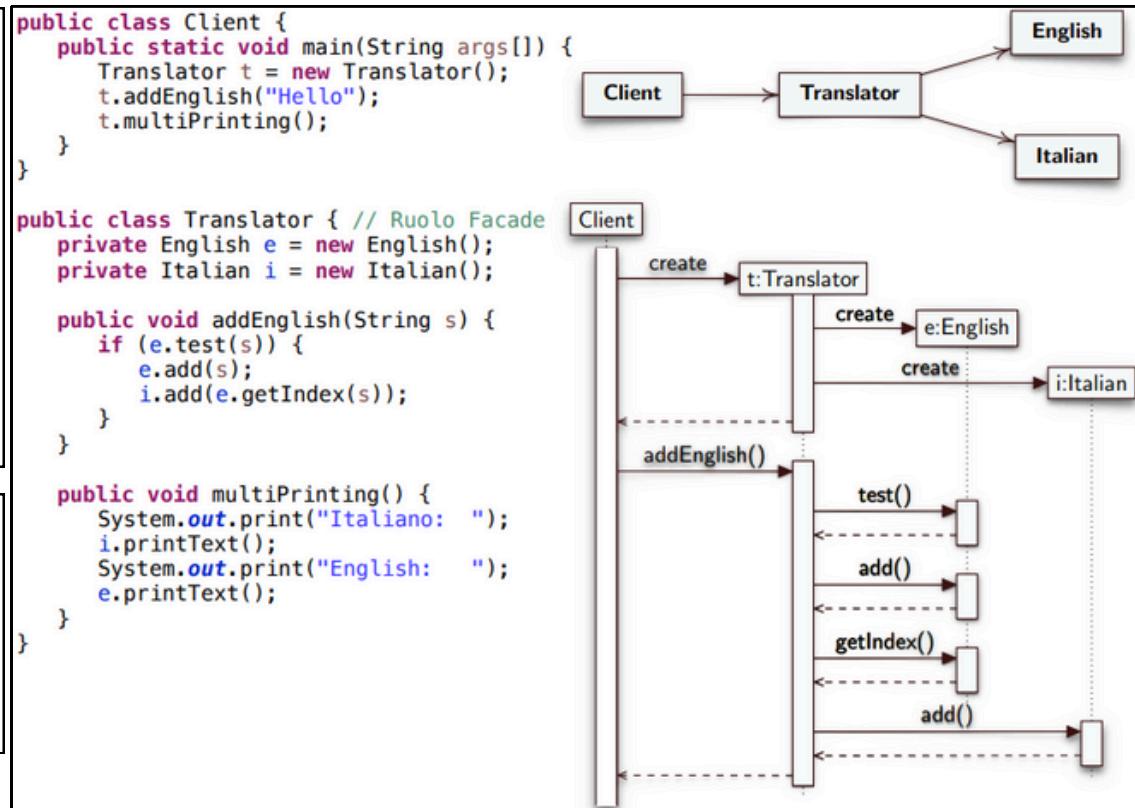
```

public class Client {
    public static void main(String args[]) {
        Translator t = new Translator();
        t.addEnglish("Hello");
        t.multiPrinting();
    }
}

public class Translator { // Ruolo Facade
    private English e = new English();
    private Italian i = new Italian();

    public void addEnglish(String s) {
        if (e.test(s)) {
            e.add(s);
            i.add(e.getIndex(s));
        }
    }

    public void multiPrinting() {
        System.out.print("Italiano: ");
        i.printText();
        System.out.print("English: ");
        e.printText();
    }
}
  
```

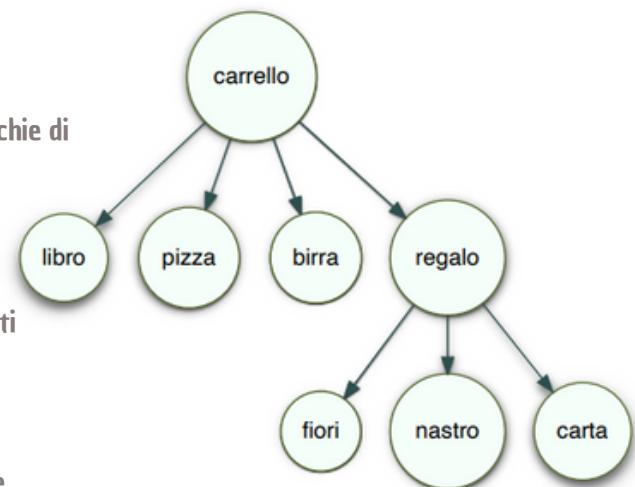


Composite

INTENTO: Comporre oggetti in strutture ad albero per rappresentare gerarchie di parti o del tutto. Composite permette ai client di trattare oggetti singoli e composizioni di oggetti uniformemente

MOTIVAZIONE

- E' necessario raggruppare elementi semplici tra loro per formare elementi più grandi
- Se nell'implementazione c'è distinzione tra classi per elementi semplici e classi per contenitori di questi elementi semplici, il codice che usa queste classi deve trattarli in modo differente. Questa distinzione rende il codice più complicato
- Composite permette di descrivere una composizione ricorsiva, in modo che i client non debbano fare distinzione tra tipi di elementi. I client tratteranno tutti gli oggetti della struttura uniformemente



ESEMPIO 1:FILE

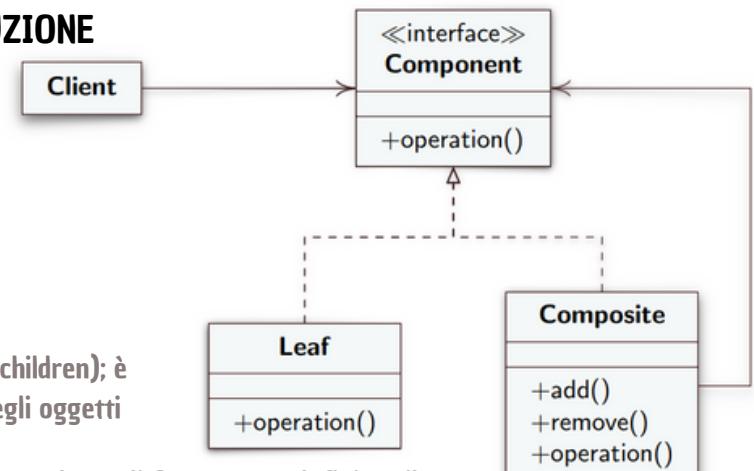
- Le operazioni su disco permettono la gestione di file (immagini, testo, etc.) e la gestione di cartelle. Esempi di operazioni: creazione, lettura, scrittura, esecuzione
- Le classi client devono poter fare operazioni su file e cartelle, senza distinguere fra essi
- Una cartella deve poter contenere al suo interno sia file che altre cartelle, queste a sua volta contengono altri elementi, e così via

ESEMPIO 2:FIGURE

- In un editor di figure ho sia figure semplici, es. Linea, Box, che figure composte, ovvero raggruppamenti di figure semplici e composte
- Le classi client devono poter trattare allo stesso modo sia le figure semplici che quelle composte, quando avviano operazioni come disegna, sposta, etc.

SOLUZIONE

- **Component:** interfaccia (o classe abstract) che rappresenta elementi semplici e non; dichiara le operazioni degli oggetti della composizione; implementa le operazioni comuni alle sottoclassi, in modo appropriato; dichiara le operazioni per l'accesso e la gestione degli elementi semplici; può definire un'operazione che permette ad un elemento di accedere all'oggetto padre nella struttura ricorsiva
- **Leaf:** classe che rappresenta elementi semplici (detti child/ children); è sottoclasse di Component; implementa il comportamento degli oggetti semplici
- **Composite:** classe che rappresenta elementi contenitori; è sottoclasse di Component; definisce il comportamento per l'aggregato di elementi child; tiene il riferimento a ciascuno degli elementi child; implementa operazioni per gestire elementi child



COLLABORAZIONI

I client usano l'interfaccia di Component per interagire con elementi della struttura composita. Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente. Se il ricevente è un Composite, questo invia la richiesta ai suoi child e possibilmente avvia operazioni addizionali prima e dopo

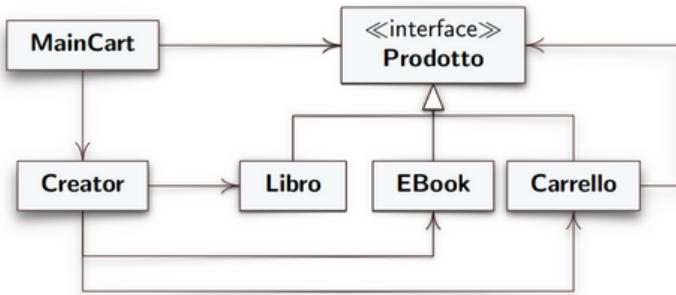
Composite

CONSEGUENZE

- Elementi semplici possono essere composti in elementi più complessi, questi possono essere composti, e così via (ovvero composizione ricorsiva)
- Un client che si aspetta un elemento semplice può riceverne uno composto
- I client sono semplici, trattano strutture composte e semplici uniformemente. I client non devono sapere se trattano con un Leaf o un Composite
- Nuovi tipi di elementi (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti
- Non è possibile a design time vincolare il Composite solo su certi componenti Leaf (in base al tipo), invece dovranno essere fatti dei controlli a runtime

ALTRI DETTAGLI

- Per facilitare la navigazione degli oggetti, gli elementi child mantengono il riferimento all'oggetto che li contiene. Il riferimento è inserito in Component, mentre Leaf e Composite possono implementare le operazioni per gestirlo
- Per avere client che non distinguono se trattano con Leaf o Composite (è uno degli obiettivi), la classe Component dovrebbe definire quante più operazioni in comune possibile. Le classi Leaf e Composite faranno override delle operazioni



- Si può inserire una operazione `getComposite()` in Component che ritorna null e che è ridefinita in Composite per ritornare il riferimento a se stesso. I client dovrebbero comunque distinguere il tipo di risultato e fare operazioni differenti, niente trasparenza
- La lista che contiene elementi child deve essere definita in Composite, altrimenti se fosse definita in Component si sprecherebbe spazio, poiché ogni Leaf avrebbe tale variabile anche se non deve usarla mai
- L'ordinamento di child per un Composite potrebbe essere importante e va tenuto in considerazione su certe implementazioni
- Il Composite potrebbe implementare una cache, per ottimizzare le prestazioni, è utile quando si implementa un'operazione che deve ricercare tra tutti i suoi componenti child. I componenti child devono poter accedere ad un'operazione che permette di invalidare la cache del Composite

DOVE DICHIARARE LE OPERAZIONI DI GESTIONE DI CHILD, ADD() E REMOVE()?

- Se dichiarate in Component si ha trasparenza, ma per le classi Leaf tali operazioni non hanno significato. La sicurezza nell'uso è quindi compromessa, poiché i client potrebbero avviare su Leaf. Se i client avviano un'operazione `add()` su Leaf e si ignora la richiesta, questo potrebbe indicare un difetto del programma
- Se dichiarate in Composite si ha sicurezza, poiché a compile time si verifica che tali operazioni non possono essere chiamate su Leaf, ma si perde la trasparenza

ESEMPIO

Si vuol gestire un prodotto e un insieme di prodotti (nel carrello) allo stesso modo

