Il pattern **Decorator** è un modello di progettazione strutturale che permette di aggiungere comportamenti o responsabilità a un oggetto in modo dinamico, senza alterare il suo codice. Questo pattern è utile quando si desidera estendere le funzionalità di una classe in modo flessibile e modulare.

Componenti del Pattern Decorator

1. **Component**: Un'interfaccia o classe astratta che definisce il comportamento base degli oggetti che possono essere decorati.
2. **ConcreteComponent**: Un'implementazione concreta del `Component` che verrà decorata.
3. **Decorator**: Una classe astratta che implementa l'interfaccia `Component` e ha un riferimento a un `Component`. Delega le operazioni al componente che decora e può aggiungere funzionalità.
4. **ConcreteDecorator**: Un'implementazione concreta del `Decorator` che aggiunge comportamenti specifici al `Component`.

Come Funziona

- Il **Decorator** contiene un riferimento a un **Component**.
- Le operazioni sul **Component** vengono delegate al componente di riferimento, con la possibilità di aggiungere comportamenti aggiuntivi prima o dopo la delega.

Esempio con Pokémon

Immagina di voler decorare un Pokémon con diverse abilità speciali, come "Volare" o "Nuotare". Puoi utilizzare il pattern Decorator per aggiungere queste abilità in modo dinamico.

Interfaccia Component

java — Copia codice

```java
public interface Pokemon {
    void attacca();
}
```

ConcreteComponent

java — Copia codice

```java
public class BasicPokemon implements Pokemon {
    private String nome;

    public BasicPokemon(String nome) {
        this.nome = nome;
    }

    @Override
    public void attacca() {
        System.out.println(nome + " attacca!");
    }
}
```

Decorator

java — Copia codice

```java
public abstract class PokemonDecorator implements Pokemon {
    protected Pokemon decoratedPokemon;

    public PokemonDecorator(Pokemon decoratedPokemon) {
        this.decoratedPokemon = decoratedPokemon;
    }

    @Override
    public void attacca() {
        decoratedPokemon.attacca();
    }
}
```

ConcreteDecorator

Volare

java — Copia codice

```java
public class Volare extends PokemonDecorator {
    public Volare(Pokemon decoratedPokemon) {
        super(decoratedPokemon);
    }

    @Override
    public void attacca() {
        super.attacca();
        vola();
    }

    private void vola() {
        System.out.println(decoratedPokemon.getClass().getSimpleName() + " vola alto nel cielo!");
    }
}
```

Nuotare

java — Copia codice

```java
public class Nuotare extends PokemonDecorator {
    public Nuotare(Pokemon decoratedPokemon) {
        super(decoratedPokemon);
    }

    @Override
    public void attacca() {
        super.attacca();
        nuota();
    }

    private void nuota() {
        System.out.println(decoratedPokemon.getClass().getSimpleName() + " nuota rapidamente nell'acqua!");
    }
}
```

Applicazione Principale

java — Copia codice

```java
public class MainApp {
    public static void main(String[] args) {
        Pokemon pikachu = new BasicPokemon("Pikachu");
        Pokemon charmander = new BasicPokemon("Charmander");

        System.out.println("Pikachu senza decorazioni:");
        pikachu.attacca();

        System.out.println("\nPikachu con abilità di volare:");
        Pokemon pikachuVolante = new Volare(pikachu);
        pikachuVolante.attacca();

        System.out.println("\nCharmander con abilità di nuotare:");
        Pokemon charmanderNuotatore = new Nuotare(charmander);
        charmanderNuotatore.attacca();

        System.out.println("\nCharmander con abilità di volare e nuotare:");
        Pokemon charmanderVolanteNuotatore = new Volare(new Nuotare(charmander));
        charmanderVolanteNuotatore.attacca();
    }
}
```

Come Funziona Questo Esempio

1. **Component (Pokemon)**: L'interfaccia `Pokemon` definisce il metodo `attacca` che tutti i Pokémon devono implementare.
2. **ConcreteComponent (BasicPokemon)**: La classe `BasicPokemon` implementa l'interfaccia `Pokemon` e fornisce un comportamento di base per l'attacco.
3. **Decorator (PokemonDecorator)**: La classe astratta `PokemonDecorator` implementa l'interfaccia `Pokemon` e ha un riferimento a un oggetto `Pokemon` che decora. Il metodo `attacca` delega l'operazione al Pokémon decorato.
4. **ConcreteDecorator (Volare, Nuotare)**: Le classi `Volare` e `Nuotare` estendono `PokemonDecorator` e aggiungono nuovi comportamenti (`vola` e `nuota`) oltre al comportamento di base.

Benefici del Pattern Decorator

- **Flessibilità**: Permette di aggiungere comportamenti a un oggetto in modo dinamico.
- **Composizione**: I decorator possono essere combinati in modo flessibile per creare comportamenti complessi.
- **Open/Closed Principle**: Il pattern segue il principio di apertura/chiusura, permettendo di estendere le funzionalità senza modificare il codice esistente.

Conclusione

Il pattern Decorator è utile quando vuoi aggiungere responsabilità o comportamenti a un oggetto in modo flessibile e dinamico. Nell'esempio dei Pokémon, è possibile decorare i Pokémon con diverse abilità speciali senza modificare le classi originali. Questo approccio rende il codice più modulare e facile da mantenere.