Introduzione

I design pattern strutturali sono **focalizzati sulle dipendenze degli oggetti** affinché abbiano certe caratteristiche. Affrontano problemi che riguardano la **composizione di classi ed oggetti**, sfruttando **ereditarietà** ed **aggregazione** per consentire il riutilizzo di oggetti esistenti.

Esistono 7 pattern strutturali in totale. Nel corso vengono analizzati in dettaglio: **Adapter**, **Decorator**, **Facade** e **Proxy**. Non vengono trattati: Bridge, Composite e Flyweight.

---

1. ADAPTER PATTERN

Scopo

Convertire l'interfaccia di una classe in un'altra. Consente di adattare l'interfaccia di una libreria esterna (Adaptee) a un'interfaccia che mi serve (Target) per poterla usare all'interno del mio sistema. Torna molto utile quando bisogna utilizzare assieme due oggetti che sono tipi diversi e quindi non sono compatibili fra loro.

Motivazione

- Spesso i **toolkit non sono riusabili** perché non conformi all'interfaccia richiesta
- **Non è corretto (e possibile) modificare il toolkit!**
- Si definisce una classe (adapter) che adatti le interfacce tramite ereditarietà o composizione
- La classe adapter può fornire funzionalità che la classe adattata non possiede

Struttura

Esistono **due modi** per implementare l'Adapter pattern:

1. Class Adapter (Adapter di classe)

- Utilizza l'**ereditarietà**
- L'Adapter eredita dalla libreria (Adaptee) e implementa l'interfaccia Target

Componenti:

- **Target**: interfaccia di dominio che il client si aspetta
- **Adaptee**: interfaccia esistente che deve essere adattata
- **Adapter**: adatta l'interfaccia di Adaptee a Target tramite ereditarietà

2. Object Adapter (Adapter di oggetto)

- Utilizza la **composizione**
- Si crea un oggetto che implementa l'interfaccia Target e contiene Adaptee come attributo

Componenti:

- **Target**: interfaccia di dominio
- **Adaptee**: interfaccia esistente da adattare
- **Adapter**: implementa Target e contiene un riferimento ad Adaptee (wrapping)

Applicabilità

- **Riutilizzo di una classe esistente** che non è conforme all'interfaccia target
- **Creazione di classi riusabili** anche con classi non ancora analizzate o viste
- Non è possibile adattare l'interfaccia attraverso ereditarietà (in questo caso si usa Object adapter)
- Il codice deve essere **compatibile con più librerie** che espongono interfacce diverse per lo stesso servizio

Conseguenze

Class Adapter:

- ✗ Non funziona quando bisogna adattare una classe e le sue sottoclassi
- ✓ Permette all'Adapter di **modificare alcune caratteristiche** dell'Adaptee

Object Adapter:

- ✓ Permette ad un Adapter di **adattare più tipi** (Adaptee e le sue sottoclassi)
- ✗ Non permette di modificare le caratteristiche dell'Adaptee
- Un oggetto adapter non è sottotipo dell'adaptee

Implementazione

- Individuare l'**insieme minimo di funzioni (narrow)** da adattare → più semplice da implementare e manutenere
- Utilizzo di operazioni astratte
- Diverse varianti strutturali alternative possibili

Problematiche

Più metodi da implementare, più complessità

Esempio (Java)

Target (interfaccia desiderata):

```
public interface Shape {
    void draw();
    void resize();
    String description();
}
```

Adaptee (libreria esterna con interfaccia diversa):

```
public class Triangle {
    public void drawShape() {
        System.out.println("Drawing triangle");
    }
    // Altri metodi con nomi diversi
}
```

1. Object Adapter:

```
class FigureObjectAdapter implements Shape {
    private final Triangle adaptee;

    public FigureObjectAdapter(Triangle adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void draw() {
        this.adaptee.drawShape(); // Adatta l'interfaccia
    }

    @Override
    public void resize() {
        System.out.println(description() + " cannot be resized.");
    }

    @Override
    public String description() {
        return "Triangle object";
    }
}

// Uso
List<Shape> shapes = new ArrayList<>();
shapes.add(new Rectangle());
shapes.add(new Circle());
shapes.add(new FigureObjectAdapter(new Triangle()));
```

2. Class Adapter:

```
class TriangleAdapter extends Triangle implements Shape {
    public TriangleAdapter() {
        super();
    }

    @Override
    public void draw() {
        this.drawShape(); // Eredita da Triangle
    }

    @Override
    public void resize() {
        System.out.println(description() + " cannot be resized.");
    }

    @Override
    public String description() {
        return "Triangle object";
    }
}

// Uso
List<Shape> shapes = new ArrayList<>();
shapes.add(new Rectangle());
shapes.add(new Circle());
shapes.add(new TriangleAdapter());
```

Quando Usare Quale Approccio

- **Object Adapter**: preferibile perché rende il coupling più fino, maggiore flessibilità
- **Class Adapter**: non utilizzabile se la classe padre è final, crea forte dipendenza tra classi, ma permette di aggiungere comportamenti

Differenza con Proxy

La differenza fondamentale rispetto al Proxy pattern è che **l'interfaccia esposta dall'adapter non è uguale a quella dell'oggetto sottostante**.

---

2. DECORATOR PATTERN

Scopo

Aggiungere responsabilità a un oggetto dinamicamente (senza utilizzo di ereditarietà). Permette di aggiungere funzioni e funzionalità a una funzionalità base. Il Decorator "ingloba" un componente in un altro oggetto che aggiunge la funzionalità.

Motivazione

- Il **subclassing non può essere sempre utilizzato** (ereditarietà statica)
- Necessità di aggiungere funzionalità **prima o dopo** l'originale
- Evitare classi "agglomerati di funzionalità" in posizioni alte della gerarchia
- Le classi componenti diventano più semplici

Struttura

Componenti:

1. **Component** (interfaccia o classe astratta):
   - Interfaccia degli oggetti da estendere
   - Definisce l'oggetto base (es. `Pizza`)
2. **ConcreteComponent**:
   - Oggetto da estendere
   - Implementazione dell'interfaccia Component (es. `PizzaBase`)
3. **Decorator** (classe astratta):
   - Permette la decorazione ("ricorsione")
   - Implementa l'interfaccia Component
   - Ha un attributo di tipo Component
   - Fa da "innesto" per far partire la catena (es. `PizzaIngredients`)
4. **ConcreteDecorator** (classi concrete):
   - Aggiungono le funzionalità al componente
   - Implementazione di Decorator
   - Prima (o dopo) dell'invocazione del metodo della classe concreta, esegue operazioni aggiuntive (decorazioni)
   - Possono avere stato aggiuntivo (`addedState`)

Caratteristica chiave: Il componente contenuto da un decoratore può essere a sua volta un'istanza di Decorator: ciò permette di **applicare successivamente più decoratori** a uno stesso oggetto, aggiungendo sempre più funzionalità.

Applicabilità

- **Aggiungere responsabilità a singoli oggetti dinamicamente** e in modo trasparente, senza coinvolgere altri oggetti
- Si vuole poter **rimuovere responsabilità** aggiunte
- Quando l'**ereditarietà è inapplicabile** (staticamente definita, non può essere modificata, o si desidera evitare esplosione di sottoclassi)

Conseguenze

- ✓ **Maggiore flessibilità** della derivazione statica

- ✓ Evita classi "agglomerati di funzionalità" in posizioni alte della gerarchia
- ✓ Le classi componenti diventano più semplici
- ✓ Utile per Software as a Service (SaaS)
- ✗ **Il decoratore e le componenti non sono uguali** (non usare se la funzionalità si basa sull'identità)
- ✗ **Proliferazione di piccole classi simili** (facili da personalizzare, ma difficili da comprendere e testare in isolamento)

Implementazione

- **Interfaccia del decoratore DEVE essere conforme** a quella del componente
- Omissione della classe astratta del decoratore in grandi gerarchie già presenti
- Mantenere "**leggera**" (stateless) l'implementazione del Component
- **Decorator vs Strategy**:
  - Decorator: quando le componenti sono "leggere"
  - Strategy: quando le componenti hanno un'implementazione corposa
- Evita decoratori troppo "costosi" da manutenere

Esempio (Java - Pizza)

```java
// Component
public interface Pizza {
    String getDescription();
    double getCost();
}

// ConcreteComponent
public class PizzaBase implements Pizza {
    @Override
    public String getDescription() {
        return "Pizza base";
    }

    @Override
    public double getCost() {
        return 4.00;
    }
}

// Decorator (classe astratta base)
public abstract class PizzaIngredients implements Pizza {
    protected Pizza pizza;

    public PizzaIngredients(Pizza pizza) {
        this.pizza = pizza;
    }
```

```
    @Override
    public String getDescription() {
        return pizza.getDescription();
    }

    @Override
    public double getCost() {
        return pizza.getCost();
    }
}

// ConcreteDecorator - Mozzarella
public class Mozzarella extends PizzaIngredients {
    public Mozzarella(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", mozzarella";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 1.50;
    }
}

// ConcreteDecorator - Tomato
public class Tomato extends PizzaIngredients {
    public Tomato(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", tomato";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 0.50;
    }
}

// ConcreteDecorator - Cipolla (Zeola)
public class Zeola extends PizzaIngredients {
    public Zeola(Pizza pizza) {
        super(pizza);
```

```
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", onion";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 0.75;
    }
}

// Uso (l'ordine non conta!)
public class Pizzaiolo {
    public static void main(String[] args) {
        Pizza margherita = new Mozzarella(new Tomato(new PizzaBase()));
        System.out.println(margherita.getDescription() + " - €" +
margherita.getCost());

        Pizza cipolla = new Zeola(new Tomato(new Mozzarella(new
PizzaBase())));
        System.out.println(cipolla.getDescription() + " - €" +
cipolla.getCost());
    }
}
```

Riconoscimento nelle Specifiche

"Ogni qual volta si ha un particolare oggetto con una **estensione** bisogna pensare al DECORATOR."

Si fa uso del DECORATOR soltanto quando nel testo si richiede di **aggiungere funzionalità ad un oggetto**.

Attenzione: non confonderlo con l'Abstract Factory quando ci sono famiglie di elementi.

---

3. FACADE PATTERN

Scopo

Fornire un'**interfaccia unica semplice** per un sottosistema complesso. Permette di fornire un'interfaccia unica (accesso semplice) per un sottosistema complesso, ma **non nasconde completamente** le funzionalità del sottosistema. **Accentra le dipendenze**.

Motivazione

- **Strutturazione di un sistema in sottosistemi**
  - Diminuisce la complessità del sistema
  - Ma aumenta le dipendenze tra sottosistemi
- L'utilizzo di un **Facade semplifica queste dipendenze**
- Ma **non nasconde le funzionalità low-level**
- Rende topologicamente più semplice il sistema
- Evita la ripetizione del codice

Struttura

Componenti:

1. **Facade**:
   - Classe che fornisce l'interfaccia unica
   - Associa ogni richiesta a una classe del sottosistema, delegando la risposta
   - Fa da punto di innesto unico
2. **Subsystem** (classi del sottosistema):
   - Implementano le funzionalità del sottosistema
   - Non hanno conoscenza del Facade
   - Il client può interagire con il Facade o con le singole componenti

Applicabilità

- Necessità di una **singola interfaccia semplice**
  - I design pattern tendono a generare tante piccole classi
  - Vista di default di un sottosistema
- **Disaccoppiamento** tra sottosistemi e client
  - Nasconde i livelli fra l'astrazione e l'implementazione
- **Stratificazione di un sistema** (architettura Three-tier)
- Si desidera fornire un'**interfaccia unificata** a un insieme di interfacce in un sottosistema
- Si desidera **raggruppare** un insieme di interfacce complesse in un'unica interfaccia più semplice

Conseguenze

- ✓ **Riduce il numero di classi** del sottosistema con cui il client deve interagire
- ✓ Realizza un **accoppiamento lasco** tra i sottosistemi e i client
- ✓ **Eliminazione delle dipendenze circolari**
- ✓ Aiuta a ridurre i **tempi di compilazione e di building**
- ✓ Non nasconde completamente le componenti di un sottosistema (client può comunque accedervi direttamente)
- ✗ **Single point of failure**

- ✗ Sovradimensionamento della classe Facade (rischio di classe grande)

Implementazione

- **Classe Facade come classe astratta**: una classe concreta per ogni "vista" (implementazione) del sottosistema
- Gestione di classi da **più sottosistemi**
- Definizione d'interfacce **"pubbliche" e "private"**:
  - Facade nasconde l'interfaccia "privata"
  - Module pattern in Javascript
- **Singleton pattern**: una sola istanza del Facade

Problematiche

- Da gestire molte dipendenze
- Rischio di introdurre bug
- Rischio di classe grande
- Deve fare un'interfaccia stabile

Esempio (Java)

```
// Subsystem classes
class SubsystemA {
    public int operationA1() { return 10; }
    public int operationA2() { return 20; }
    public int operationA3() { return 30; }
}

class SubsystemB {
    public int operationB1() { return 5; }
    public int operationB2() { return 15; }
}

class SubsystemC {
    public int operationC1() { return 8; }
    public int operationC2() { return 12; }
}

class SubsystemD {
    public int operationD1() { return 3; }
    public int operationD2() { return 7; }
}

// Facade
class SubsystemFacade {
    private SubsystemA a;
    private SubsystemB b;
```

```
    private SubsystemC c;
    private SubsystemD d;

    public SubsystemFacade() {
        // Gestisce lui le new
        a = new SubsystemA();
        b = new SubsystemB();
        c = new SubsystemC();
        d = new SubsystemD();
    }

    // Interfaccia semplificata
    public int operation1() {
        return a.operationA1() + b.operationB1() +
               c.operationC1() + d.operationD1();
    }

    public int operation2() {
        return a.operationA2() + b.operationB2() +
               c.operationC2() + d.operationD2();
    }
    // ... altri metodi semplificati
}

// Client
public class Client {
    public static void main(String[] args) {
        SubsystemFacade facade = new SubsystemFacade();

        // Interfaccia semplice invece di gestire A, B, C, D
        int result1 = facade.operation1();
        int result2 = facade.operation2();
    }
}
```

Analogia: È come se i subsystem fossero classi che controllano vari hardware del computer (HDD, GPU, RAM) e il facade fosse una classe Computer che con semplici comandi fa partire tutti i componenti e dice il loro stato.

---

4. PROXY PATTERN

Scopo

Fornire un **surrogato di un altro oggetto** di cui si vuole **controllare l'accesso**. Rinvia il costo di creazione di un oggetto all'effettivo utilizzo (**on demand**, lazy evaluation). Il proxy agisce come l'oggetto che ingloba, presentando la **stessa interfaccia**.

Motivazione

- Rinviare il costo di creazione di un oggetto all'effettivo utilizzo (**on demand**)
- Il proxy agisce come l'oggetto che ingloba (stessa interfaccia)
- Le funzionalità dell'oggetto "inglobato" vengono accedute attraverso il proxy
- ...o senza l'accesso vero e proprio (virtual proxy)
- Mostrare una versione "lite" di un oggetto, un'anteprima poco pesante, che poi è in grado di generare l'oggetto vero e proprio

Struttura

Componenti:

1. **Subject** (interfaccia):
   - Definisce l'interfaccia comune che permette di utilizzare il proxy come l'oggetto inglobato
   - Definisce l'oggetto base
2. **RealSubject**:
   - L'oggetto rappresentato dal proxy
   - Implementazione dell'interfaccia Subject
3. **Proxy**:
   - Implementa l'interfaccia Subject
   - Ha un attributo di tipo Subject (RealSubject)
   - Attraverso il puntatore al RealSubject, permette l'accesso controllato alle sue funzionalità

Tipologie di Proxy

1. Virtual Proxy

- **Creazione di oggetti complessi on-demand**
- Carica l'oggetto in memoria solo quando strettamente necessario
- Esempio: anteprime (thumbnail) di immagini o video

2. Remote Proxy

- **Rappresentazione locale** di un oggetto che si trova in uno **spazio di indirizzi differente**
- Si fa vedere locale un oggetto che in realtà è remoto (sfruttando la rete)
- Esempio: classi stub in Java RMI
- Indicatore: "componenti su **macchine differenti**"

3. Protection Proxy

- **Controllo degli accessi** (diritti) all'oggetto originale
- Permette l'accesso ad alcune funzionalità solo a certi utenti (client)
- Fa il controllo dei diritti di accesso

4. Puntatore "intelligente" (Smart Pointer)

- Gestione della memoria
- Esempio: gestione della memoria in Objective-C
- Tiene traccia degli oggetti istanziati
- Può tenere il conto dei riferimenti ad un oggetto e lo elimina quando il conto raggiunge zero

Applicabilità

- **Accesso remoto**: proxy remoto fornisce interfaccia locale per oggetto in sistema remoto
- **Proxy virtuale**: può creare un oggetto complesso solo quando viene richiesto
- **Proxy di protezione**: controlla l'accesso all'oggetto originale (verifica permessi, semaforo per accesso concorrente)
- **Riferimento intelligente**: gestisce il ciclo di vita di un oggetto originale

Conseguenze

- ✓ Introduce un **livello di indirezione** che può essere "farcito":
  - Remote proxy: nasconde dove un oggetto risiede
  - Virtual proxy: effettua delle ottimizzazioni
  - Protection proxy: definisce ruoli di accesso alle informazioni
- ✓ **Copy-on-write**: la copia di un oggetto viene eseguita unicamente quando la copia viene modificata

Implementazione

- Implementazione "**a puntatore**" (overload operatore `->` e `*` in C++)
- Alcuni proxy agiscono in modo differente rispetto alle operazioni
  - In Java: costruzione tramite reflection (Spring, Hibernate...)
- **Proxy per più tipi**:
  - Subject è una classe astratta
  - Ma non se il proxy deve istanziare il tipo concreto!
- Rappresentazione del subject nel proxy

Esempio (Java - Virtual Proxy per Video)

```
// Subject (interfaccia comune)
public interface Video {
    void playVideo();
```

```
}

// RealSubject (oggetto "pesante")
public class RealVideo implements Video {
    private final byte[] file;

    public RealVideo(byte[] file) {
        this.file = file;
        System.out.println("Loading video..."); // Operazione costosa
    }

    @Override
    public void playVideo() {
        System.out.println("Reproducing video...");
    }
}

// Proxy (versione "lite")
public class VideoProxy implements Video {
    private final String filePath;
    private RealVideo video; // Lazy initialization

    public VideoProxy(String filePath) {
        this.filePath = filePath;
    }

    @Override
    public void playVideo() {
        // Carica il video solo quando necessario (lazy)
        if (video == null) {
            byte[] bytes = getBytesFromVideoPath(filePath);
            this.video = new RealVideo(bytes);
        }
        this.video.playVideo();
    }

    private byte[] getBytesFromVideoPath(String path) {
        // Legge i bytes dal file
        return new byte[0]; // Semplificato
    }
}

// Document che usa i proxy
public class Document {
    private final List<Video> videos;

    public Document(List<Video> videos) {
        this.videos = videos;
    }
```

```
    public void show() {
        System.out.println("Showing videos list:");
        // Carica i video in memoria solo quando necessario
        videos.stream().forEach(Video::playVideo);
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        List<Video> list = new ArrayList<>();
        list.add(new VideoProxy("roberto.mp4"));
        list.add(new VideoProxy("roberta.mp4"));

        // In memoria non ci sono video caricati, ci sono solo i proxy
        final Document document = new Document(list);

        // Adesso dopo la show ci sono i video in memoria
        document.show();
    }
}
```

Vantaggi dell'esempio: al momento della creazione della lista dei video si ha molta efficienza perché si caricano in memoria i bytes solo quando strettamente richiesto.

Differenza con Adapter

La differenza fondamentale rispetto all'Adapter pattern è che **l'interfaccia esposta dal proxy è uguale a quella dell'oggetto sottostante**, mentre nell'Adapter le interfacce sono diverse.

Confronti tra Pattern

Adapter vs Proxy

Adapter	Proxy
Converte un'interfaccia in un'altra	Mantiene la stessa interfaccia
Risolve incompatibilità di interfacce	Controlla l'accesso all'oggetto
Le interfacce sono **diverse**	Le interfacce sono **uguali**

Decorator vs Adapter

Decorator	Adapter
Aggiunge funzionalità mantenendo l'interfaccia	Cambia l'interfaccia
Supporta composizione ricorsiva	Non supporta ricorsione
Enfatizza l'estensione dinamica	Enfatizza la compatibilità

Facade vs Proxy

Facade	Proxy
Semplifica interfacce complesse	Controlla l'accesso
Può raggruppare più oggetti	Rappresenta un singolo oggetto
Non nasconde completamente i subsystem	Presenta stessa interfaccia del subject

Decorator vs Strategy

- **Decorator**: modifica la "pelle" (quando le componenti sono "leggere")
- **Strategy**: modifica la "pancia" (quando le componenti hanno implementazione corposa)

Riconoscimento dei Pattern nelle Specifiche

Indicatori per Adapter:

- "Utilizzare una **libreria esterna**"
- "Adattare l'interfaccia di..."
- "Compatibilità con più librerie"
- Necessità di far lavorare insieme oggetti con **interfacce incompatibili**

Indicatori per Decorator:

- "**Aggiungere funzionalità** ad un oggetto"
- "Estensione dinamica"
- "Configurazione" (es. auto con cerchi in lega)
- Ogni volta che si ha un oggetto con una **estensione**

Indicatori per Facade:

- "Interfaccia **semplificata**"
- "**Sottosistema complesso**"
- "Sistema con molti componenti"
- "Punto di accesso unico"

Indicatori per Proxy:

- "Componenti su **macchine differenti**" → Remote Proxy
- "Caricamento **lazy**" o "**on-demand**"
- "**Anteprima**" o "thumbnail"
- "Controllo degli **accessi**" → Protection Proxy
- "Ottimizzazione della memoria"

Classificazione dei Pattern Strutturali

Pattern	Tipo di Relazione	Scopo Principale
Adapter (Class)	Class	Adattamento interfacce tramite ereditarietà
Adapter (Object)	Object	Adattamento interfacce tramite composizione
Decorator	Object	Aggiunta dinamica di responsabilità
Facade	Object	Semplificazione interfaccia sottosistema
Proxy	Object	Controllo accesso a oggetto
