

1 Lezione del 09-12-25

Continuiamo l'argomento dei design pattern.

1.0.1 Pattern strutturali

Dopo i pattern *creazionali*, iniziamo a parlare dei pattern **strutturali**. Questi riguardano la *composizione* di oggetti. Un principio spesso usato è il *composition over inheritance*, che consiglia di preferire la composizione di oggetti all'ereditarietà per ridurre l'accoppiamento tra le classi e migliorare la modularità dei sistemi.

Questi pattern, quindi, sono:

- **Adapter**: usato per convertire l'interfaccia di una classe in un'altra interfaccia, che chi usa la classe si aspetta. Risulta utile quando vogliamo riutilizzare classi esistenti all'interno di un nuovo sistema senza modificare né le classi originali né il codice che usa la classe.

Vediamo ad esempio un lettore audio, per cui vogliamo adottare una certa classe *OldAudioReader* a supporto di una certa interfaccia *NewAudioReader*:

```
1 class OldAudioReader {
2     public void play() {
3         // ...
4     }
5 }
6
7 interface NewAudioReader {
8     void playMedia();
9 }
10
11 // adattatore
12 class OldAudioReaderAdapter implements NewAudioReader {
13     private OldAudioReader reader;
14     public OldAudioReaderAdapter(OldAudioReader reader) {
15         this.reader = reader;
16     }
17     @Override
18     public void playMediaMedia() {
19         reader.play(); // adattiamo il metodo esistente
20     }
21 }
```

- **Bridge**: l'obiettivo di questo pattern è disaccoppiare un'astrazione dalla sua implementazione in modo che queste possano variare indipendentemente.

Ad esempio, nel progettare un'interfaccia grafica portabile, poniamo di volere un'astrazione *Window* che funzioni sia su *X Window System* che su *IBM Presentation Manager (PM)*. Un approccio basato su ereditarietà potrebbe definire una classe astratta *Window* e due sottoclassi, *XWindow* e *PMWindow*, per i diversi sistemi. Questo approccio presenta però 2 problemi:

- Risulta difficile estendere l'astrazione *Window* per supportare nuovi tipi di finestre o nuove piattaforme;
- Per ogni nuova combinazione (es. *IconWindow* per finestre con icone) e piattaforma, bisogna creare più classi specifiche (*XIconWindow*, *PMIconWindow*, ecc.). Ogni nuova piattaforma moltiplica il numero di classi necessarie.

Vediamo quindi di usare un approccio alternativo, dove una sola classe può sfruttare più *implementor*, cioè interfacce che espongono funzioni di basso livello da ciascuna delle API:

```

1 // implementor astratto
2 interface DrawingAPI {
3     void drawWindow();
4 }
5
6 // implementor concreti
7
8 class XDrawingAPI {
9     public void drawWindow() {
10         // ...
11     }
12 }
13
14 class IBMDrawingAPI {
15     public void drawWindow() {
16         // ...
17     }
18 }
19
20 // classe vera e propria
21 class Window {
22     private DrawingAPI api;
23
24     public Window(DrawingApi api) {
25         this.api = api;
26     }
27
28     public void draw() {
29         // basso livello
30         api.drawWindow();
31     }
32
33     public void resize() {
34         // alto livello
35         // ...
36     }
37 }

```

A questo punto sarà semplice creare oggetti *Window* che si appoggiano all'una o l'altra API, come segue:

```

1 Window XWindow = new Window(new XDrawingAPI());
2 Window IBMWindow = new Window(new IBMDrawingAPI());

```

- **Composite:** l'idea è di comporre oggetti in strutture ad albero che rappresentano gerarchie parti-complesso. In questo modo si possono trattare collezioni di oggetti, o loro singole istanze, in maniera uniforme. La chiave di questo pattern è quindi una classe astratta che rappresenta sia degli oggetti *primitivi*, che dei *container* di tali oggetti.

L'esempio più semplice è quello di un file system, dove sia i file (*foglie* o *oggetti primitivi*) che le directory (*compositi* o *collezioni di oggetti*) sono istanze di una sola classe *FSNode*:

```

1 // interfaccia base di file e directory
2 interface FSNode {

```

```

3  String getName();
4  int getSize(); // in KB
5  void print(String indent);
6
7  // operazioni di composizione: default "non supportate" per le
   foglie
8  default void add(FSNode node) {
9      throw new UnsupportedOperationException(
10         getClass().getSimpleName() + " is a leaf");
11 }
12 default void remove(FSNode node) {
13     throw new UnsupportedOperationException(
14         getClass().getSimpleName() + " is a leaf");
15 }
16 }
17
18 // leaf (file)
19 class FileLeaf implements FSNode {
20     private final String name;
21     private final int sizeKB;
22
23     FileLeaf(String name, int sizeKB) {
24         this.name = name;
25         this.sizeKB = sizeKB;
26     }
27     public String getName() {
28         return name;
29     }
30     public int getSize() {
31         return sizeKB;
32     }
33     public void print(String indent) {
34         System.out.printf("%s- %s (%d KB)%n", indent, name, sizeKB);
35     }
36 }
37
38 // composite (directory)
39 class Directory implements FSNode {
40     private final String name;
41     private final List<FSNode> children = new ArrayList<>();
42
43     Directory(String name) {
44         this.name = name;
45     }
46     public String getName() {
47         return name;
48     }
49     public int getSize() {
50         // somma delle dimensioni dei figli
51         return children.stream().mapToInt((e) -> e.getSize()).sum();
52     }
53     public void add(FSNode node) {
54         children.add(node);
55     }
56     public void remove(FSNode node) {
57         children.remove(node);
58     }
59     public void print(String indent) {
60         System.out.printf("%s+ %s/ (%d KB)%n", indent, name, getSize());
61         String childIndent = indent + " ";

```

```

62     for (FSNode c : children) {
63         c.print(childIndent);
64     }
65 }
66 }

```

- **Decorator:** questo è un pattern che si usa per aggiungere funzionalità, in maniera dinamica, a oggetti preesistenti. Forniscono quindi un'alternativa alle sottoclassi per la gestione di funzionalità aggiuntive, quando vogliamo dare più responsabilità a singoli oggetti, e non ad intere classi.

Torniamo all'esempio delle interfacce grafiche. Magari il nostro toolkit vuole fornire la possibilità di rendere qualsiasi componente *cliccabile* o *scrollabile*.

Una soluzione semplice potrebbe essere quella di creare classi figlie di ogni componente, scrollabili e cliccabili.

Un'alternativa è quella di "avvolgere" gli oggetti componente in altri oggetti, detti decoratori. I decoratori si conformano all'interfaccia degli oggetti che decorano, così da essere trasparenti agli utenti della classe.

Vediamo quindi un esempio:

```

1  // componente base
2  interface Text {
3      String render();
4  }
5
6  // implementazione concreta
7  class PlainText implements Text {
8      private final String content;
9
10     PlainText(String content) {
11         this.content = content;
12     }
13     public String render() {
14         return content;
15     }
16 }
17
18 // classe base del decoratore, stessa interfaccia
19 abstract class TextDecorator implements Text {
20     protected final Text inner;
21
22     TextDecorator(Text inner) {
23         this.inner = inner;
24     }
25
26     // possiamo ridefinire questo metodo!
27     public abstract String render();
28 }
29
30 // decoratori concreti
31
32 class BoldDecorator extends TextDecorator {
33     BoldDecorator(Text inner) {
34         super(inner);
35     }
36     public String render() {
37         return "<b>" + inner.render() + "</b>";

```

```

38     }
39 }
40
41 class ItalicDecorator extends TextDecorator {
42     ItalicDecorator(Text inner) {
43         super(inner);
44     }
45     public String render() {
46         return "<i>" + inner.render() + "</i>";
47     }
48 }

```

- **Facade:** l'intento è di fornire un'interfaccia unificata ad un insieme di interfacce in un sottosistema. In breve, una facade fornisce un'interfaccia di livello più alto che rende l'intero sottosistema più semplice da usare.

Prendiamo l'esempio di un emulatore, che contiene più componenti:

```

1 // componente astratto
2 abstract class SimulationComponent {
3     void simulate();
4 }
5
6 // componenti concreti
7
8 class Bus extends SimulationComponent {
9     // ...
10 }
11 class Memory extends SimulationComponent {
12     // ...
13 }
14 class Processor extends SimulationComponent {
15     // ...
16 }
17
18 // simulazione concreta, rappresenta la facade del sistema
19 class Simulation {
20     Bus bus;
21     Memory mem;
22     Processor proc;
23
24     public Simulation(Bus bus, Memory mem, Processor proc) {
25         this.bus = bus;
26         this.mem = mem;
27         this.proc = proc;
28     }
29
30     public void step() {
31         // la facade nasconde la complessità
32         bus.step();
33         mem.step();
34         proc.step();
35     }
36 }

```

- **Flyweight:** utile quando abbiamo un grande numero di oggetti che condividono caratteristiche immutabili: decidiamo di utilizzare la condivisione delle risorse per risparmiare memoria. Sostanzialmente, quindi, l'approccio è simile a quello del *pooling*.

Vediamo ad esempio un sistema che gestisce più caselle, che possono condividere lo stesso tipo:

```

1 // stato condiviso
2 record TileType(String texture, int movementCost) { }
3
4 // fabbrica di flyweight
5 class TileTypeFactory {
6     private static final Map<String, TileType> cache = new HashMap
7     <>();
8
9     static TileType get(String texture, int cost) {
10         return cache.computeIfAbsent(texture, k -> new TileType(
11             texture, cost));
12     }
13 }
14
15 // stato univoco: otteniamo TileType condiviso dalla fabbrica
16 record Tile(int x, int y, TileType type) { }

```

- **Proxy:** qui vogliamo fornire un oggetto "fittizio", che fa da segnaposto per un altro oggetto. Il proxy può svolgere alcune funzioni:
 - Permettere di raggiungere oggetti in altri spazi di indirizzamento;
 - Controllare gli accessi agli oggetti che rappresenta;
 - Rappresentare un oggetto costoso e crearlo solo quando necessario (*lazy instantiation* o *lazy loading* (se ci sono accessi al disco)).

Un esempio di applicazione di questo pattern, tornando all'esempio del rendering dei documenti, è per gestire una classe *Image*, che va istanziata solo quando è effettivamente necessario disegnarla, ed è nel frattempo sostituita da un proxy.

```

1 // interfaccia comune
2 interface Image {
3     void display();
4 }
5
6 // classe reale (costosa da caricare)
7 class RealImage implements Image {
8     private final String filename;
9
10    public RealImage(String filename) {
11        this.filename = filename;
12        loadFromDisk();
13    }
14
15    private void loadFromDisk() {
16        System.out.println("Loading image from disk: " + filename);
17    }
18
19    public void display() {
20        System.out.println("Displaying image: " + filename);
21    }
22 }
23
24 // classe proxy
25 class ImageProxy implements Image {
26     private final String filename;
27     private RealImage realImage; // inizialmente null
28
29     public ImageProxy(String filename) {

```

```

28     this.filename = filename;
29 }
30 public void display() {
31     if (realImage == null) { // lazy loading
32         realImage = new RealImage(filename);
33     }
34     realImage.display();
35 }
36 }

```

1.0.2 Pattern comportamentali

Come ultima categoria di design pattern, parliamo dei pattern **comportamentali**. Questi riguardano il modo con cui gli oggetti interagiscono e si scambiano informazioni.

Sono, in particolare:

- **Chain of responsibility**: consiste nell'evitare di accoppiare la responsabilità di una richiesta al ricevitore di una richiesta, dando a più di un oggetto la possibilità di gestire tale richiesta. Quando arriva una richiesta, quindi, ogni gestore decide cosa fare e se necessario propagare la richiesta a altri gestori.

Vediamo ad esempio un videogioco dove un giocatore può subire danno, e questo danno può essere elaborato sulla base di più fattori (difficoltà del gioco, presenza o meno di armatura, ecc...)

```

1  class DamageRequest {
2      public double amount;
3      DamageRequest(double amount) {
4          this.amount = amount;
5      }
6  }
7
8  // handler astratto
9  abstract class DamageHandler {
10     protected DamageHandler next;
11     public DamageHandler setNext(DamageHandler next) {
12         this.next = next;
13         return next;
14     }
15     public void handle(DamageRequest req) {
16         if (!process(req))
17             return; // se false, la catena si ferma
18         if (next != null)
19             next.handle(req);
20     }
21     // true se deve continuare la catena
22     protected abstract boolean process(DamageRequest req);
23 }
24
25 class ArmorHandler extends DamageHandler {
26     private final double armor; // riduce di un certo numero di punti
27     // il danno
28     ArmorHandler(double armor) {
29         this.armor = armor;
30     }
31     protected boolean process(DamageRequest req) {
32         double before = req.amount;
33         req.amount = Math.max(0, req.amount - armor);
34         System.out.println("[Armor] " + before + " -> " + req.amount);

```

```

34     return req.amount > 0; // se il danno e' 0, fermo la catena
35 }
36 }
37
38 class DifficultyHandler extends DamageHandler {
39     private final double difficultyMultiplier;
40     DifficultyHandler(double difficultyMultiplier) {
41         this.difficultyMultiplier = difficultyMultiplier;
42     }
43     protected boolean process(DamageRequest req) {
44         double before = req.amount;
45         req.amount = req.amount * difficultyMultiplier;
46         System.out.println("[Diff] " + before + " -> " + req.amount);
47         return true;
48     }
49 }
50
51 class Player {
52     private double hp = 100;
53     public void applyDamage(double dmg) {
54         hp -= dmg;
55         System.out.println(">> Player danneggiato " + dmg + ", HP = " +
56             hp);
57     }
58 }
59
60 public class Example {
61     public static void main(String[] args) {
62         Player player = new Player();
63         // costruisco la catena: armor -> difficulty
64         DamageHandler chain = new ArmorHandler(5);
65         chain.setNext(new DifficultyHandler(1.5));
66         DamageRequest req = new DamageRequest(30);
67         System.out.println("Danno base: " + req.amount);
68         chain.handle(req);
69         player.applyDamage(req.amount);
70     }
71 }

```

- **Command:** qui vogliamo incapsulare richieste all'interno di oggetti, permettendo di parametrizzare le richieste, metterle in code, e renderle annullabili.

Il comando incapsula l'operazione da compiere, e su chi operare (il ricevitore), ma non l'implementazione dell'operazione da compiere (quella starà nel ricevitore).

Vediamo ad esempio il classico gestionale bancario, dove prevediamo comandi di *Deposit* e *Withdraw* che operano su un certo oggetto ricevitore *BankAccount*:

```

1 interface Command {
2     void execute();
3     void undo();
4 }
5
6 // oggetto ricevitore
7 class BankAccount {
8     private int balance = 0;
9
10    public void deposit(int amount) {
11        balance += amount;
12        System.out.println("Deposit " + amount + " => balance = " +
13            balance);
14    }
15 }

```



```
14 public void withdraw(int amount) {
15     balance -= amount;
16     System.out.println("Withdraw " + amount + " => balance = " +
17         balance);
18 }
19 public int getBalance() {
20     return balance;
21 }
22 }
23 class DepositCommand implements Command {
24     private final BankAccount account;
25     private final int amount;
26
27     public DepositCommand(BankAccount account, int amount) {
28         this.account = account;
29         this.amount = amount;
30     }
31     public void execute() {
32         account.deposit(amount);
33     }
34     public void undo() {
35         account.withdraw(amount);
36     }
37 }
38
39 class WithdrawCommand implements Command {
40     private final BankAccount account;
41     private final int amount;
42
43     public WithdrawCommand(BankAccount account, int amount) {
44         this.account = account;
45         this.amount = amount;
46     }
47     public void execute() {
48         account.withdraw(amount);
49     }
50     public void undo() {
51         account.deposit(amount);
52     }
53 }
54
55 class CommandManager {
56     // mettiamo i comandi in una pila
57     private final Stack<Command> history = new Stack<>();
58
59     public void run(Command cmd) {
60         cmd.execute();
61         history.push(cmd);
62     }
63     public void undo() {
64         if (history.isEmpty()) {
65             System.out.println("Nothing to undo");
66             return;
67         }
68         Command last = history.pop();
69         last.undo();
70     }
71 }
72 }
```

```

73 public class Example {
74     public static void main(String[] args) {
75         BankAccount account = new BankAccount();
76         CommandManager manager = new CommandManager();
77         manager.run(new DepositCommand(account, 100));
78         manager.run(new WithdrawCommand(account, 40));
79         manager.run(new DepositCommand(account, 50));
80         System.out.println(">> Undo last operation");
81         manager.undo();
82         System.out.println(">> Undo another operation");
83         manager.undo();
84         System.out.println("Final balance = " + account.getBalance());
85     }
86 }

```

- **Interpreter:** un interprete per un linguaggio permette di definire delle **DSL** (*Domain Specific Language*), e avere un modo per interpretarle.

L'implementazione degli *interpreter* è simile a quella dei *composite*, in quanto il modo più naturale per gestire grammatiche e attraverso strutture ad albero di dati nodi sintattici. La differenza principale dal *composite* è però che si definisce, appunto, una *grammatica*, cioè una struttura specifica che l'albero che formiamo deve assumere.

In un linguaggio come il Java, scrivere interpreti risulta molto semplice:

```

1  // espressione astratta
2  interface Expression {
3      int interpret();
4  }
5  // espressione terminale (numero)
6  class Number implements Expression {
7      private final int value;
8
9      public Number(int value) {
10         this.value = value;
11     }
12     public int interpret() { return value; }
13 }
14
15 // espressioni non terminali: somma o sottrazione
16
17 class Plus implements Expression {
18     private final Expression left, right;
19
20     public Plus(Expression left, Expression right) {
21         this.left = left;
22         this.right = right;
23     }
24     public int interpret() {
25         return left.interpret() + right.interpret();
26     }
27 }
28
29 class Minus implements Expression {
30     private final Expression left, right;
31
32     public Minus(Expression left, Expression right) {
33         this.left = left;
34         this.right = right;

```

```

35     }
36     public int interpret() {
37         return left.interpret() - right.interpret();
38     }
39 }
40
41 // ...

```

- **Iterator:** gli iteratori, come già visti in C++, forniscono un modo per accedere agli elementi di un oggetto aggregato in maniera sequenziale, mascherando all'utente dell'iteratore il funzionamento di tale attraversamento sequenziale (e quindi fornendo un interfaccia unificata).

```

1  interface MyIterator<T> {
2      boolean hasNext();
3      T next();
4  }
5
6  class MyCollection<T> {
7      private List<T> l = new ArrayList<>();
8
9      public void add(T elem) {
10         l.add(elem);
11     }
12     public MyIterator<T> iterator() {
13         return new MyListIterator<>(l);
14     }
15 }
16
17 class MyListIterator<T> implements MyIterator<T> {
18     private List<T> list;
19     private int pos = 0;
20
21     public MyListIterator(List<T> list) {
22         this.list = list;
23     }
24     @Override
25     public boolean hasNext() {
26         return pos < list.size();
27     }
28     @Override
29     public T next() {
30         if (!hasNext()) {
31             throw new IllegalStateException("Non ci sono piu' elementi");
32         }
33         return list.get(pos++);
34     }
35 }

```

- **Mediator:** è un pattern dove gestiamo più oggetti che comunicano fra di loro attraverso un singolo oggetto mediatore. Permette di gestire un insieme di oggetti con molte relazioni oggetto-oggetto.

In questo caso, anziché avere molte relazioni, ogni oggetto comunica solo con il mediatore. Riduce accoppiamento e semplifica manutenzione e evoluzione. La logica di comunicazione diventa centralizzata. Ad esempio, in una GUI, se un campo viene selezionato se ne deselectano altri e così via...

Vediamo l'esempio di una chat room dove più utenti si scambiano messaggi comunicando con lo stesso mediatore:

```

1 interface ChatMediator {
2     void sendMessage(String msg, User sender);
3     void addUser(User user);
4 }
5
6 class ChatRoom implements ChatMediator {
7     private final List<User> users = new ArrayList<>();
8     @Override
9     public void addUser(User user) {
10         users.add(user);
11     }
12     @Override
13     public void sendMessage(String msg, User sender) {
14         for (User u : users) {
15             if (u != sender) {
16                 u.receive(msg, sender.getName());
17             }
18         }
19     }
20 }
21 abstract class User {
22     protected ChatMediator mediator;
23     protected String name;
24     public User(ChatMediator mediator, String name) {
25         this.mediator = mediator;
26         this.name = name;
27         mediator.addUser(this);
28     }
29     public String getName() {
30         return name;
31     }
32     public abstract void send(String msg);
33     public abstract void receive(String msg, String from);
34 }
35 class ChatUser extends User {
36     public ChatUser(ChatMediator mediator, String name) {
37         super(mediator, name);
38     }
39     @Override
40     public void send(String msg) {
41         System.out.println(name + " sends: " + msg);
42         mediator.sendMessage(msg, this);
43     }
44     @Override
45     public void receive(String msg, String from) {
46         System.out.println(name + " receives from " + from + ": " + msg);
47     }
48 }
49 public class Example {
50     public static void main(String[] args) {
51         ChatMediator mediator = new ChatRoom();
52         User mario = new ChatUser(mediator, "Mario");
53         User luigi = new ChatUser(mediator, "Luigi");
54         User peach = new ChatUser(mediator, "Peach");
55         mario.send("Ciao a tutti!");
56         luigi.send("Ciao!");
57         // ....
58     }
59 }

```

59 }

Notiamo che un pattern che è emerso finora è quello del *programming against interfaces*: quello che vogliamo in genere fare è definire prima le interfacce, e poi le classi concrete che implementano tali interfacce.

In questo modo, il codice degli utenti delle classi non dovrà cambiare: piuttosto, a cambiare sarà l'istanziamento delle classi concrete vere e proprie.