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*:

```
class OldAudioReader {
    public void play() {
      // ...
    }
}

interface NewAudioReader {
    void playMedia();
}

// adattatore
class OldAudioReaderAdapter implements NewAudioReader {
  private OldAudioReader reader;
  public OldAudioReaderAdapter(OldAudioReader reader) {
    this.reader = reader;
  }
  @Override
  public void playMediaMedia() {
    reader.play(); // adattiamo il metodo esistente
  }
}
```

- **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:

```
// impementor astratto
interface DrawingAPI {
    void drawWindow();
}

// implementor concreti

class XDrawingAPI {
    public void drawWindow() {
      // ...
    }
}

class IBMDrawingAPI {
    public void drawWindow() {
      // ...
    }
}

// classe vera e propria
class Window {
    private DrawingAPI api;

    public Window(DrawingApi api) {
      this.api = api;
    }

    public void draw() {
      // basso livello
      api.drawWindow();
    }

    public void resize() {
      // alto livello
      // ...
    }
}
```

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

```
Window XWindow = new Window(new XDrawingAPI());
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*:

```
// interfaccia base di file e directory
interface FSNode {
```

```
  String getName();
  int getSize(); // in KB
  void print(String indent);

  // operazioni di composizione: default "non supportate" per le
    foglie
  default void add(FSNode node) {
    throw new UnsupportedOperationException(
        getClass().getSimpleName() + " is a leaf");
  }
  default void remove(FSNode node) {
    throw new UnsupportedOperationException(
        getClass().getSimpleName() + " is a leaf");
  }
}

// leaf (file)
class FileLeaf implements FSNode {
  private final String name;
  private final int sizeKB;

  FileLeaf(String name, int sizeKB) {
    this.name = name;
    this.sizeKB = sizeKB;
  }
  public String getName() {
    return name;
  }
  public int getSize() {
    return sizeKB;
  }
  public void print(String indent) {
    System.out.printf("%s- %s (%d KB)%n", indent, name, sizeKB);
  }
}

// composite (directory)
class Directory implements FSNode {
  private final String name;
  private final List<FSNode> children = new ArrayList<>();

  Directory(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public int getSize() {
    // somma delle dimensioni dei figli
    return children.stream().mapToInt((e) -> e.getSize()).sum();
  }
  public void add(FSNode node) {
    children.add(node);
  }
  public void remove(FSNode node) {
    children.remove(node);
  }
  public void print(String indent) {
    System.out.printf("%s+ %s/ (%d KB)%n", indent, name, getSize());
    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>";
```

```
  }
}

class ItalicDecorator extends TextDecorator {
  ItalicDecorator(Text inner) {
    super(inner);
  }
  public String render() {
    return "<i>" + inner.render() + "</i>";
  }
}
```

- **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:

```
// componente astratto
abstract class SimulationComponent {
  void simulate();
}

// componenti concreti

class Bus extends SimulationComponent {
  // ...
}
class Memory extends SimulationComponent {
  // ...
}
class Processor extends SimulationComponent {
  // ...
}

// simulazione concreta, rappresenta la facade del sistema
class Simulation {
  Bus bus;
  Memory mem;
  Processor proc;

  public Simulation(Bus bus, Memory mem, Processor proc) {
    this.bus = bus;
    this.mem = mem;
    this.proc = proc;
  }

  public void step() {
    // la facade nasconde la complessita'
    bus.step();
    mem.step();
    proc.step();
  }
}
```

- **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:

```
// stato condiviso
record TileType(String texture, int movementCost) { }

// fabbrica di flyweight
class TileTypeFactory {
    private static final Map<String, TileType> cache = new HashMap
    <>();

    static TileType get(String texture, int cost) {
        return cache.computeIfAbsent(texture, k -> new TileType(
    texture, cost));
    }
}

// stato univoco: ottieniamo TileType condiviso dalla fabbrica
record Tile(int x, int y, TileType type) { }
```

- **Proxy**: qui vogliamo fornire un oggetto "fittizio", che fa da segnaposto per un altro oggeto. Il proxy può svolgere alcune funzioni:
  - Permettere di raggiungere oggetti in altri spazi di indirizzamento;
  - Controllare gli accessi agli oggett 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.

```
// interfaccia comune
interface Image {
  void display();
}

// classe reale (costosa da caricare)
class RealImage implements Image {
  private final String filename;

  public RealImage(String filename) {
    this.filename = filename;
    loadFromDisk();
  }
  private void loadFromDisk() {
    System.out.println("Loading image from disk: " + filename);
  }
  public void display() {
    System.out.println("Displaying image: " + filename);
  }
}

// classe proxy
class ImageProxy implements Image {
  private final String filename;
  private RealImage realImage; // inizialmente null

  public ImageProxy(String filename) {
```

```
    this.filename = filename;
  }
  public void display() {
    if (realImage == null) { // lazy loading
      realImage = new RealImage(filename);
    }
    realImage.display();
  }
}
```

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...)

```
class DamageRequest {
  public double amount;
  DamageRequest(double amount) {
    this.amount = amount;
  }
}

// handler astratto
abstract class DamageHandler {
  protected DamageHandler next;
  public DamageHandler setNext(DamageHandler next) {
    this.next = next;
    return next;
  }
  public void handle(DamageRequest req) {
    if (!process(req))
      return; // se false, la catena si ferma
    if (next != null)
      next.handle(req);
  }
  // true se deve continuare la catena
  protected abstract boolean process(DamageRequest req);
}

class ArmorHandler extends DamageHandler {
  private final double armor; // riduce di un certo numero di punti
    il danno
  ArmorHandler(double armor) {
    this.armor = armor;
  }
  protected boolean process(DamageRequest req) {
    double before = req.amount;
    req.amount = Math.max(0, req.amount - armor);
    System.out.println("[Armor] " + before + " -> " + req.amount);
```

```
    return req.amount > 0; // se il danno e' 0, fermo la catena
  }
}

class DifficultyHandler extends DamageHandler {
  private final double difficultyMultiplier;
  DifficultyHandler(double difficultyMultiplier) {
    this.difficultyMultiplier = difficultyMultiplier;
  }
  protected boolean process(DamageRequest req) {
    double before = req.amount;
    req.amount = req.amount * difficultyMultiplier;
    System.out.println("[Diff] " + before + " -> " + req.amount);
    return true;
  }
}

class Player {
  private double hp = 100;
  public void applyDamage(double dmg) {
    hp -= dmg;
    System.out.println(">> Player danneggiato " + dmg + ", HP = " +
     hp);
  }
}

public class Example {
  public static void main(String[] args) {
    Player player = new Player();
    // costruisco la catena: armor -> difficulty
    DamageHandler chain = new ArmorHandler(5);
    chain.setNext(new DifficultyHandler(1.5));
    DamageRequest req = new DamageRequest(30);
    System.out.println("Danno base: " + req.amount);
    chain.handle(req);
    player.applyDamage(req.amount);
  }
```

- **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*:

```
interface Command {
  void execute();
  void undo();
}

// oggetto ricevitore
class BankAccount {
  private int balance = 0;

  public void deposit(int amount) {
    balance += amount;
    System.out.println("Deposit " + amount + " => balance = " +
     balance);
  }
```

```
14    public void withdraw(int amount) {
15      balance -= amount;
16      System.out.println("Withdraw " + amount + " => balance = " +
       balance);
17    }
18    public int getBalance() {
19      return balance;
20    }
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
```

```
public class Example {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    CommandManager manager = new CommandManager();
    manager.run(new DepositCommand(account, 100));
    manager.run(new WithdrawCommand(account, 40));
    manager.run(new DepositCommand(account, 50));
    System.out.println(">> Undo last operation");
    manager.undo();
    System.out.println(">> Undo another operation");
    manager.undo();
    System.out.println("Final balance = " + account.getBalance());
  }
}
```

- **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:

```
// espressione astratta
interface Expression {
  int interpret();
}
// espressione terminale (numero)
class Number implements Expression {
  private final int value;

  public Number(int value) {
    this.value = value;
  }
  public int interpret() { return value; }
}

// espressioni non terminali: somma o sottrazione

class Plus implements Expression {
  private final Expression left, right;

  public Plus(Expression left, Expression right) {
    this.left = left;
    this.right = right;
  }
  public int interpret() {
    return left.interpret() + right.interpret();
  }
}

class Minus implements Expression {
  private final Expression left, right;

  public Minus(Expression left, Expression right) {
    this.left = left;
    this.right = right;
```

```
  }
  public int interpret() {
    return left.interpret() - right.interpret();
  }
}

// ...
```

- **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).

```
interface MyIterator<T> {
  boolean hasNext();
  T next();
}

class MyCollection<T> {
  private List<T> l = new ArrayList<>();

  public void add(T elem) {
    l.add(elem);
  }
  public MyIterator<T> iterator() {
    return new MyListIterator<>(l);
  }
}

class MyListIterator<T> implements MyIterator<T> {
  private List<T> list;
  private int pos = 0;

  public MyListIterator(List<T> list) {
    this.list = list;
  }
  @Override
  public boolean hasNext() {
    return pos < list.size();
  }
  @Override
  public T next() {
    if (!hasNext()) {
      throw new IllegalStateException("Non ci sono piu' elementi");
    }
    return list.get(pos++);
  }
}
```

- **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 deselezionano altri e così via...

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

```
interface ChatMediator {
  void sendMessage(String msg, User sender);
  void addUser(User user);
}

class ChatRoom implements ChatMediator {
  private final List<User> users = new ArrayList<>();
  @Override
  public void addUser(User user) {
    users.add(user);
  }
  @Override
  public void sendMessage(String msg, User sender) {
    for (User u : users) {
      if (u != sender) {
        u.receive(msg, sender.getName());
      }
    }
  }
}
abstract class User {
  protected ChatMediator mediator;
  protected String name;
  public User(ChatMediator mediator, String name) {
    this.mediator = mediator;
    this.name = name;
    mediator.addUser(this);
  }
  public String getName() {
    return name;
  }
  public abstract void send(String msg);
  public abstract void receive(String msg, String from);
}
lass ChatUser extends User {
  public ChatUser(ChatMediator mediator, String name) {
    super(mediator, name);
  }
  @Override
  public void send(String msg) {
    System.out.println(name + " sends: " + msg);
    mediator.sendMessage(msg, this);
  }
  @Override
  public void receive(String msg, String from) {
    System.out.println(name + " receives from " + from + ": " + msg);
  }
}
public class Example {
  public static void main(String[] args) {
    ChatMediator mediator = new ChatRoom();
    User mario = new ChatUser(mediator, "Mario");
    User luigi = new ChatUser(mediator, "Luigi");
    User peach = new ChatUser(mediator, "Peach");
    mario.send("Ciao a tutti!");
    luigi.send("Ciao!");
    // ....
  }
```

```
}
```

- **Memento**: il memento è un pattern secondo il quale immagazziniamo lo stato (immutabile) di un oggetto per poterlo ricaricare in un istante successivo. Il *memento* è quindi l'oggetto immutabile che mantiene lo stato, mentre un'altro oggetto (detto *caretaker*) sarà quello che si occuperà di prelevare, memorizzare e restituire successivamente lo stato.

  Poniamo l'esempio di un sistema di salvataggio in un videogioco:

```
// memento
class PlayerState {
  private final int life;
  private final int x, y;
  private final List<String> inventory;

  public PlayerState(int life, int x, int y, List<String> inventory)
    {
    this.life = life;
    this.x = x;
    this.y = y;
    // Crea una copia della lista
    this.inventory = new ArrayList<>(inventory);
  }
  public int getLife() {
    return life;
  }
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public List<String> getInventory() {
    // restituisce una copia non modificabile
    return Collections.unmodifiableList(inventory);
  }
}

// oggetto originatore dei dati
class Player {
  private int life = 100;
  private int x = 0, y = 0;
  private final List<String> inventory = new ArrayList<>();

  public void move(int dx, int dy) {
    x += dx;
    y += dy;
  }
  public void damage(int amount) {
    life -= amount;
  }
  public void addItem(String item) {
    inventory.add(item);
  }
  public void printStatus() {
    System.out.println(
        "Life=" + life + ", pos=(" + x + "," + y + "), items=" +
    inventory);
  }
```

```
  // crea un memento dallo stato
  public PlayerState saveState() {
    return new PlayerState(life, x, y, inventory);
  }
  // ripristina lo stato da un memento
  public void restoreState(PlayerState m) {
    this.life = m.getLife();
    this.x = m.getX();
    this.y = m.getY();
    inventory.clear();
    // Li riaggiunge tutti
    inventory.addAll(m.getInventory());
  }
}

// caretaker
class SaveManager {
  private PlayerState checkpoint;

  public void save(PlayerState m) {
    checkpoint = m;
  }
  public PlayerState load() {
    return checkpoint;
  }
}
```

A questo punto salvataggio e caricamento dello stato del giocatore sono banali come:

```
Player player = new Player();
SaveManager saver = new SaveManager();

saver.save(player.playerState()); // salva

// ...

player.restoreState(saver.load()); // carica
```

- **Observer**: questo pattern permette di realizzare un sistema di *eventi*, anche detto un sistema *publish-subscribe*, dove un certo oggetto (detto *osservatore*) sottoscrive agli eventi generati da altri oggetti (che vengono detti *osservati*).

  Questo può essere utile nel caso di oggetti che devono propagare cambiamenti di stato verso l'alto, senza necessariamente chi è interesato ad elaborare tali cambiamenti.

  Un esempio può essere quello dell'emulatore visto per le *facade*: poniamo infatti che ogni componente sia capace di generare un evento, e che la nostra facade voglia rilevare tali eventi:

```
// evento
class SimulationEvent {
  private final String mess;

  public String getMess() {
    return mess;
  }
}

```

```
10 interface SimulationListener {
11   void onEvent(SimulationEvent event);
12 }
13
14 // componente astratto
15 abstract class SimulationComponent {
16   private List<SimulationListener> listeners = new ArrayList<>();
17
18   void subscribe(SimulationListener listener) {
19     listeners.add(listener);
20   }
21
22   void simulate();
23   void raiseEvent(SimulationEvent event) {
24     for(SimulationListener listener : listeners) {
25       listener.onEvent(event);
26     }
27   }
28 }
29
30 // componenti concreti
31
32 class Bus extends SimulationComponent {
33   // ...
34 }
35 class Memory extends SimulationComponent {
36   // ...
37 }
38 class Processor extends SimulationComponent {
39   // ...
40 }
41
42 // simulazione concreta, rappresenta la facade del sistema
43 class Simulation implements SimulationListener {
44   Bus bus;
45   Memory mem;
46   Processor proc;
47
48   public Simulation(Bus bus, Memory mem, Processor proc) {
49     this.bus = bus;
50     this.mem = mem;
51     this.proc = proc;
52   }
53
54   public void init() {
55     bus.subscribe(this);
56     mem.subscribe(this);
57     proc.subscribe(this);
58   }
59
60   public void onEvent(SimulationEvent e) {
61     System.out.println(e.getMess());
62   }
63
64   public void step() {
65     // la facade nasconde la complessita'
66     bus.step();
67     mem.step();
68     proc.step();
69   }
```

```
}
```

- **State**: vogliamo incapsulare lo *stato* di un oggetto all'interno di classi, e gestire tali stati attraverso un *contesto* comune. Le chiamate al contesto verrano redirette allo stato corrente, e lo stato corrente avrà un riferimento al contesto per poter aggiornare lo stato.

  Facciamo l'esempio di un riproduttore musicale che ha 2 stati: lo stato in *pausa* e lo stato in *riproduzione* di musica:

```
// interfaccia di stato
interface PlayerState {
  void pressButton(MusicPlayer player);
}

// stati concreti
class PlayingState implements PlayerState {
  @Override
  public void pressButton(MusicPlayer player) {
    System.out.println("Pausing music...");
    player.setState(new PausedState());
  }
}

class PausedState implements PlayerState {
  @Override
  public void pressButton(MusicPlayer player) {
    System.out.println("Playing music...");
    player.setState(new PlayingState());
  }
}

// contesto
class MusicPlayer {
  private PlayerState state = new PausedState(); // stato iniziale
  public void setState(PlayerState state) {
    this.state = state;
  }
  public void pressButton() {
    state.pressButton(this);
  }
}

public class Esempio {
  public static void main(String[] args) {
    MusicPlayer player = new MusicPlayer();
    player.pressButton(); // Playing
    player.pressButton(); // Pausing
    player.pressButton(); // Playing
  }
}
```

- **Strategy**: questo pattern consiste nel prendere famiglie di algoritmi (algoritmi con gli stessi argomenti che danno risultati dello stesso tipo, quindi rappresentabili da interfacce), ed incapsularli per renderli intercambiabili.

  Gli oggetti che useranno gli algoritmi potranno quindi vedere la strategia usata cambiare nel tempo sulla base di diverse implementazioni.

Poniamo l'esempio di un validatore di password che può usare diversi algoritmi di validazione:

```
// interfaccia di strategia
interface PasswordStrategy {
  boolean isValid(String password);
}

// strategie concrete

class SimplePassword implements PasswordStrategy {
  public boolean isValid(String pwd) {
      // piu' di 5 caratteri
      // ...
  }
}

class StrongPassword implements PasswordStrategy {
  public boolean isValid(String pwd) {
      // piu' di 7 caratteri
      // ....
  }
}

// contesto delle strategie
class PasswordValidator {
  private PasswordStrategy strategy;
  public void setStrategy(PasswordStrategy strategy) {
    this.strategy = strategy;
  }
  public void validate(String pwd) {
    if (strategy == null) {
      System.out.println("Nessuna strategia impostata!");
      return;
    }
    System.out.println(
        "Password \"" + pwd + "\" valida? " + strategy.isValid(pwd));
  }
}

public class Example {
  public static void main(String[] args) {
    PasswordValidator validator = new PasswordValidator();
    validator.setStrategy(new SimplePassword());
    validator.validate("ciao12"); // true
    validator.setStrategy(new StrongPassword());
    validator.validate("ciao12"); // false
    validator.validate("Ciao1234"); // true
  }
}
```

- **Template method**: l'idea è di definire lo *"scheletro"* di un algoritmo che definisce un operazione, mantenendo astratti i passaggi dell'esecuzione di tale algoritmo. In seguito, si potrà specializzare la classe, riempiendo i passaggi astratti con implementazioni concrete.

  Vediamo un esempio:

```
// template method
abstract class Processor {
  public final void() {
```

```
    step0();
    step1();
    step2();
  }

  protected abstract void step0();
  protected abstract void step1();
  protected abstract void step2();
}

// implementazioni concrete

class ProcessorA {
  protected void step0() {
    // ...
  }
  protected void step1() {
    // ...
  }
  protected void step2() {
    // ...
  }
}

class ProcessorB {
  protected void step0() {
    // ...
  }
  protected void step1() {
    // ...
  }
  protected void step2() {
    // ...
  }
}
```

- **Visitor**: fornisce un alternativa alla overriding di metodi astratti. Si prevede infatti un a certa classe, detta *visitor*, che prevede più metodi, ognuno capace di gestire un istanza di classe in una certa gerarchia di classi.

  A questo punto le classi nella gerarchia dovranno solo implementare un metodo per l'accettazione del visitor.

  Vediamo l'esempio banale, che è quello di un visitor per il calcolo delle aree di diversi specializzazioni di una classe *Shape*:

```
// elemento astratto
interface Shape {
  void accept(ShapeVisitor visitor);
}

// elementi concreti
class Circle implements Shape {
  double radius;

  public Circle(double radius) {
    this.radius = radius;
  }
  public double getRadius() {
    return radius;
```

```
  }

  // questo va ridefinito qui (!)
  // non possiamo usare this nei metodi default
  @Override
  public void accept(ShapeVisitor visitor) {
    visitor.visit(this);
  }
}

class Rectangle implements Shape {
  double w, h;

  public Rectangle(double w, double h) {
    this.w = w;
    this.h = h;
  }
  public double getWidth() {
    return w;
  }
  public double getHeight() {
    return h;
  }

  // come sopra
  @Override
  public void accept(ShapeVisitor visitor) {
    visitor.visit(this);
  }
}

// visitor
class AreaVisitor {
  public void visit(Circle c) {
    double area = Math.PI * c.getRadius() * c.getRadius();
    System.out.println("Area cerchio: " + area);
  }

  public void visit(Rectangle r) {
    double area = r.getWidth() * r.getHeight();
    System.out.println("Area rettangolo: " + area);
  }
}
```

A questo punto si può calcolare l'area di una qualsiasi forma con:

```
Shape rect = new Rectangle(2, 5);
Shape circ = new Circle(3);

AreaVisitor visitor = new AreaVisitor();
rect.accept(visitor); // area rettangolo
circ.accept(visitor); // area cerchio
```

Questo è utile, in quanto l'alternativa basata sull'ereditarietà prevedeva la definizione di un qualche metodo:

```
public double calcola(Shape s) {
  if (s instanceof Circle) {
    Circle c = (Circle) s;
    return Math.PI * c.getRadius() * c.getRadius();
  } else if (s instanceof Rectangle) {
```

```
6       Rectangle r = (Rectangle) s;
7       return r.getWidth() * r.getHeight();
8     }
9     throw new IllegalArgumentException("Forma sconosciuta");
10  }
```

che è molto brutto in quanto:

- Presenta molti `if-else`;
- Richiede il costrutto `instanceof`, e quindi il controllo dinamico del tipo di classe (con riflessione).

Il nostro approccio è più compatto e rimanda la discriminazione fra classi ad un meccanismo base del linguaggio (l'overriding delle funzioni con stesso nome ma diversi argomenti).

Notiamo infine che un altro vantaggio è quello che si possono definire più visitatori (magari da un interfaccia comune), che si occupano di implementare operazioni diverse sugli oggetti che visitano (senza dover modificare gli oggetti stessi).

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'istanziazione delle classi concrete vere e proprie.