Introduzione

I design pattern comportamentali definiscono **come un oggetto svolge la sua funzione** e **in che modo i vari oggetti comunicano fra di loro**. Affrontano problemi relativi agli algoritmi e all'assegnazione delle responsabilità tra oggetti, non riguardando solo la struttura delle classi ma anche i pattern di comunicazione tra esse.

Si fa un ampio uso di questi design pattern nei framework dei vari linguaggi come Spring (per Java, C#). Esistono 11 pattern comportamentali in totale, ma nel corso vengono analizzati in dettaglio i seguenti: **Command**, **Iterator**, **Observer**, **Strategy** e **Template Method**.

---

1. COMMAND PATTERN

Scopo

Incapsulare una richiesta in un oggetto, cosicché i **client siano indipendenti dalle richieste**. Il pattern incapsula in un oggetto tutti i dati e i metodi necessari per eseguire una certa azione (un comando), creando un accoppiamento molto lasco con i client che usano il command.

Motivazione

- Necessità di gestire richieste di cui non si conoscono i particolari
- I toolkit associano ai propri elementi richieste da eseguire
- Una classe astratta Command definisce l'interfaccia per eseguire la richiesta
- La richiesta diventa un semplice oggetto manipolabile

Struttura

Componenti principali:

1. **Command**: interfaccia di esecuzione delle richieste (metodo `execute()`)
2. **ConcreteCommand**: implementa la richiesta concreta, invocando l'operazione sul Receiver
3. **Receiver**: conosce come portare a termine la richiesta del comando concreto (metodo `Action()`)
4. **Invoker**: esegue il comando
5. **Client**: crea il comando concreto e lo configura

Flusso:

1. Client crea `new Command(aReceiver)`
2. Client chiama `StoreCommand(aCommand)` sull'Invoker
3. Invoker chiama `Execute()` sul Command
4. Command chiama `Action()` sul Receiver

Applicabilità

- **Parametrizzazione** di oggetti sull'azione da eseguire (callback function)
- **Specificare, accodare ed eseguire** richieste multiple volte
- **Supporto a operazioni di Undo e Redo**: il comando memorizza lo stato necessario per annullare i suoi effetti
- **Supporto a transazioni**: un comando equivale a un'operazione atomica
- Organizzare un sistema in operazioni ad alto livello costruite su operazioni a basso livello

Conseguenze

- **Accoppiamento lasco** tra oggetto invocante e quello che porta a termine l'operazione
- I command **possono essere estesi** facilmente
- I comandi **possono essere composti e innestati**
- È facile **aggiungere nuovi comandi** senza modificare le classi esistenti

Implementazione

- **Quanto deve essere intelligente un comando?**
  - Semplice binding fra receiver e azione da eseguire
  - Comandi agnostici e autoconsistenti
- **Supporto a undo/redo**: attenzione allo stato del sistema da mantenere (receiver, argomenti, valori originali)
- **History list** per gestire la sequenza di comandi
- Accumulo di errori durante l'esecuzione di più comandi successivi
- Utilizzo di template C++ o Generics Java

Esempio (Java)

```java
// Command interface
public interface Command {
    void execute();
}

// Concrete Command
public class PasteCommand implements Command {
    private TextEditor receiver;
```

```
    public PasteCommand(TextEditor receiver) {
        this.receiver = receiver;
    }

    public void execute() {
        receiver.paste();
    }
}

// Invoker
public class Button {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void click() {
        command.execute();
    }
}
```

---

2. ITERATOR PATTERN

Scopo

Fornisce l'**accesso sequenziale** agli elementi di un aggregato **senza esporre l'implementazione** dell'aggregato stesso.

Motivazione

- "Per scorrere non è necessario conoscere"
- Devono essere disponibili **diverse politiche di attraversamento**
- L'Iterator pattern **sposta la responsabilità di attraversamento** in un oggetto iteratore separato
- L'iteratore tiene traccia dell'elemento corrente

Applicabilità

- Accedere agli elementi di un oggetto aggregato senza esporre la sua rappresentazione sottostante
- Supportare **più attraversamenti** in modo uniforme
- Fornire un'**interfaccia uniforme** per attraversare diversi tipi di aggregati

Esempio

Utilizzato ampiamente in Java con l'interfaccia `Iterator` per attraversare collezioni (liste, set, alberi) senza conoscerne i dettagli implementativi.

---

3. OBSERVER PATTERN

Scopo

Definisce una **dipendenza 1..n** fra oggetti, riflettendo la modifica di un oggetto sui dipendenti. Conosciuto anche come **"Publish-Subscribe"**.

Motivazione

- Mantenere la **consistenza** fra oggetti
- Separare modello e viste ad esso collegate
- Il pattern definisce come implementare la relazione di dipendenza:
  - **Subject**: effettua le notifiche
  - **Observer**: si aggiorna in risposta a una notifica

Struttura

Componenti:

1. **Subject** (classe astratta, MAI interfaccia!):
   - Contiene come attributo una lista di Observer
   - Definisce operazioni per aggiungere e rimuovere Observer (`Attach()` , `Detach()`)
   - Contiene metodo `Notify()` (implementato) che notifica tutti gli Observer registrati
   - Mantiene lo stato di cui viene data una "vista" concreta
2. **ConcreteSubject**:
   - Implementa le operazioni di Subject
   - Mantiene lo stato (`subjectState`)
   - Fornisce `GetState()` per accedere allo stato
3. **Observer** (interfaccia, DEVE esserlo):
   - Definisce l'operazione di aggiornamento con metodo `Update()` astratto
4. **ConcreteObserver**:
   - Implementa l'operazione di aggiornamento
   - Implementa `Update()` chiedendo lo stato del ConcreteSubject
   - Ha un riferimento al soggetto concreto
   - Possiede lo stato che deve essere aggiornato (`observerState`)

Flusso:

1. ConcreteSubject cambia stato → chiama `SetState()`

2. ConcreteSubject chiama `Notify()` (ereditato da Subject)
3. Subject itera su tutti gli observer e chiama `Update()` su ciascuno
4. Ogni ConcreteObserver chiama `GetState()` sul ConcreteSubject per sincronizzarsi

Applicabilità

- Un'astrazione presenta due aspetti, di cui uno dipende dall'altro (incapsulando questi aspetti in oggetti separati, è possibile riusarli indipendentemente)
- Un cambiamento di stato in un oggetto richiede modifiche in altri oggetti dipendenti, ma non si conosce il numero di oggetti dipendenti
- Un oggetto deve notificare altri oggetti senza fare ipotesi su chi siano (alto livello di **disaccoppiamento**)

Conseguenze

- **Accoppiamento astratto** tra soggetti e osservatori (i soggetti non conoscono il tipo concreto degli osservatori)
- **Comunicazione broadcast**: libertà di aggiungere osservatori dinamicamente
- **Aggiornamenti non voluti**: un'operazione "innocua" sul soggetto può provocare una cascata "pesante" di aggiornamenti
- Gli osservatori non sanno cosa è cambiato nel soggetto

Implementazione

- **Protocolli di aggiornamento**:
  - **Push model**: il soggetto invia tutti i dati agli osservatori
  - **Pull model**: il soggetto invia solo la notifica, gli osservatori richiedono i dati necessari
- Evitare protocolli con assunzioni rigide
- Notifica delle modifiche: gli osservatori possono registrarsi su un particolare evento (Topic)
- **Chi attiva l'aggiornamento?**
  - Il soggetto, dopo ogni cambiamento di stato
  - Il client, al termine del processo di interazione
- Evitare puntatori "pendenti" (dangling)
- Notificare solo in stati consistenti (utilizzo del Template Method pattern)
- Si può notificare ogni tot aggiornamenti per non dover notificare ogni volta

Problematiche

- Non tutti gli Observer sono interessati alle medesime informazioni → utilizzare i **Topic**
- Possibilità di notificare ogni tot aggiornamenti dello stato

Uso Pratico

Questo pattern si usa molto come **base architetturale di sistemi event-driven** e nel pattern **MVC** (Model-View-Controller), dove:

- **View** è il Subject
- **Controller** è l'Observer
- Il Model è referenziato all'interno del Controller

---

4. STRATEGY PATTERN

Scopo

Definisce una **famiglia di algoritmi** che possono essere fra loro interscambiabili. Permette di cambiare il comportamento di un algoritmo a runtime senza modificare la classe che lo usa. Gli algoritmi sono **differenti varianti** dello stesso scopo.

Motivazione

- Necessità di avere differenti varianti dello stesso algoritmo
- Evitare la proliferazione di sottoclassi per ogni variante
- Incapsulare algoritmi come oggetti strategy
- I **comportamenti cambiano** ma la **struttura** dell'interfaccia resta sempre uguale

Struttura

Componenti:

1. **Strategy** (interfaccia):
   - Definisce l'operazione comune a tutti gli algoritmi
   - Modo unico per accedere all'algoritmo
   - Interfaccia supportata da tutti gli algoritmi
2. **ConcreteStrategy**:
   - Implementa l'operazione comune
   - Implementazione concreta di un algoritmo
3. **Context**:
   - Classe che usa l'algoritmo
   - Contiene un riferimento a Strategy
   - Configurato con una strategia concreta
   - Può definire un'interfaccia per l'accesso ai propri dati
   - Di solito è un'interfaccia

Applicabilità

- **Molte classi correlate differiscono solo per il comportamento**: Strategy fornisce un modo per configurare una classe con un comportamento scelto fra tanti
- Sono necessarie **più varianti di un algoritmo** (implementati come classi separate selezionabili a runtime)
- Un algoritmo usa una **struttura dati che non dovrebbe essere resa nota** ai client
- Una classe definisce molti comportamenti che compaiono all'interno di **scelte condizionali multiple** → spostare i blocchi di codice in classi Strategy dedicate
- Quando si parla di **formati** diversi (XML, JSON, PDF, ecc.) che richiedono algoritmi diversi
- **Intelligenza artificiale** o algoritmi che variano in efficienza

Conseguenze

- **Differenti implementazioni** dello stesso comportamento
- I client a volte devono conoscere dettagli implementativi per selezionare il corretto algoritmo
- **Comunicazione tra contesto e algoritmo**: alcuni algoritmi non utilizzano tutti gli input
- **Incremento del numero di oggetti** nell'applicazione

Implementazione

- Definire le interfacce di strategie e contesti:
  - Fornire singolarmente i dati alle strategie
  - Fornire l'intero contesto alle strategie
  - Inserire un puntamento al contesto nelle strategie
- Implementazione strategie: C++ Template, Java Generics (solo se l'algoritmo può essere determinato a compile time)
- Utilizzo di strategie opzionali: definire una strategia di default

Esempio (Java)

```java
// Strategy interface
public interface Discounter {
    double applyDiscount(double amount);
}

// Concrete Strategies
public class EasterDiscounter implements Discounter {
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.4; // 40% sconto
    }
}

public class ChristmasDiscounter implements Discounter {
```

```
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.15; // 15% sconto
    }
}

// Uso
public static void main(String[] args) {
    Discounter discounter;
    double discount;

    if (isEaster)
        discount = (new EasterDiscounter()).applyDiscount(13);
    else
        discount = (new ChristmasDiscounter()).applyDiscount(13);
}
```

Differenza con Template Method

- **Strategy**: definisco una **famiglia di algoritmi** che hanno lo stesso scopo ma sono fatti in modo diverso (cambiano i **comportamenti**, non la struttura)
- **Template Method**: definisco un **algoritmo UNICO** che è sempre quello, ma cambiano alcune **operazioni specifiche** (passi di esecuzione) al suo interno

---

5. TEMPLATE METHOD PATTERN

Scopo

Definisce lo **scheletro di un algoritmo**, lasciando l'implementazione di alcuni passi alle sottoclassi, senza modificare l'algoritmo originale. Riutilizzo il workflow dell'algoritmo ma l'implementazione delle singole operazioni può variare.

Motivazione

- Definire un algoritmo in termini di **operazioni astratte**
- Viene fissato solo l'**ordine delle operazioni**
- Le sottoclassi forniscono il **comportamento concreto**

Struttura

Componenti:

1. **AbstractClass** (classe astratta):
    - Definisce il **template method** (contiene il workflow dell'algoritmo) → dovrebbe essere `final`

- Definisce le **operazioni primitive** astratte (operazioni che variano)
- Definisce lo scheletro dell'algoritmo

2. **ConcreteClass**:
   - Implementa le operazioni primitive
   - Fornisce i passi concreti all'algoritmo

Applicabilità

- **Implementare le parti invarianti** di un algoritmo una volta sola
- **Evitare la duplicazione del codice** (principio "refactoring to generalize")
- **Controllare le possibili estensioni** di una classe (fornire operazioni astratte e operazioni hook)
- Un comportamento comune fra sottoclassi deve essere portato a fattore comune e localizzato

Conseguenze

- **Tecnica per il riuso del codice** (fattorizzazione delle responsabilità)
- **"The Hollywood principle"**: "Don't call us, we'll call you"
- **Tipi di operazioni possibili**:
  1. **Operazioni concrete** della classe astratta
  2. **Operazioni primitive** (astratte): devono essere implementate
  3. **Operazioni hook**: forniscono operazioni di default che non fanno nulla, ma rappresentano punti di estensione
- Documentare bene quali sono operazioni primitive e quali hook

Implementazione

- Le operazioni primitive dovrebbero essere membri **protetti**
- Il template method **non dovrebbe essere ridefinito** (Java: dichiarazione `final`)
- **Minimizzare il numero di operazioni primitive** (altrimenti resta poco nel template method)
- Definire una **naming convention** per i nomi delle operazioni di cui effettuare override

Problematiche

Usa l'**ereditarietà** anziché la **composizione** (meno flessibile dello Strategy)

Esempio (Java)

```
public abstract class Calcolatore {
    // Template Method (final!)
    public final int calcola(int[] array) {
        int value = valoreIniziale();
```

```
        for (int i = 0; i < array.length; i++) {
            value = esegui(value, array[i]);
        }
        return value;
    }

    // Operazioni primitive (abstract)
    protected abstract int valoreIniziale();
    protected abstract int esegui(int currentValue, int element);
}

public class CalcolatoreSomma extends Calcolatore {
    protected int esegui(int currentValue, int element) {
        return currentValue + element;
    }

    protected int valoreIniziale() {
        return 0;
    }
}

// Esempio Login Manager
public abstract class LoginManager {
    // Template Method
    public final User login(String username, String password) {
        validateNetworkConnection();  // Operazione concreta
        validateInput(username, password);  // Operazione primitiva
        authenticate(username, password);  // Operazione primitiva
        return getUserData(username);  // Operazione primitiva
    }

    // Operazione concreta (uguale per tutti)
    private void validateNetworkConnection() {
        // Verifica connessione internet
    }

    // Operazioni primitive (abstract)
    protected abstract void validateInput(String username, String password);
    protected abstract void authenticate(String username, String password);
    protected abstract User getUserData(String username);
}
```

Differenza con Strategy

- **Template Method**: ho passi uguali e mi serve uno **scheletro costante** dell'algoritmo. Definisco un algoritmo UNICO con workflow fisso. Usa **ereditarietà** (più rigido).
- **Strategy**: ho la stessa famiglia di algoritmi che **non hanno gli stessi passi**. Gli algoritmi sono completamente diversi internamente. Usa **composizione** (più flessibile).

Confronti tra Pattern

Strategy vs Template Method

Strategy	Template Method
Famiglia di algoritmi intercambiabili	Un singolo algoritmo con passi variabili
Gli algoritmi sono completamente diversi	Lo scheletro dell'algoritmo è fisso
Usa **composizione**	Usa **ereditarietà**
Più **flessibile** (cambio a runtime)	Più **rigido** (struttura fissa)
Cambiano i **comportamenti**	Cambiano alcuni **passi**

Observer vs Command

- **Observer**: sistema di notifiche 1-a-molti, comunicazione broadcast
- **Command**: incapsulamento di richieste come oggetti, supporto a undo/redo

---

Riconoscimento dei Pattern nelle Specifiche

Indicatori per Strategy:

- Parole chiave: **"algoritmo"**, **"varianti"**, **"formati"** (XML, JSON, PDF)
- "L'algoritmo sarà esteso con nuovi algoritmi"
- "Intelligenza artificiale via via migliore"
- "Sulla base del formato... seleziona un algoritmo differente"

Indicatori per Template Method:

- "Workflow", "scheletro", "passi comuni"
- "Alcuni passi dell'algoritmo variano"
- "Processo di login" con passi fissi ma implementazioni diverse

Indicatori per Observer:

- "Interfaccia interattiva", "notifiche", "eventi"
- "Aggiornamento automatico quando..."
- Pattern **MVC** richiede sempre Observer
- "Il sistema reagisce a modifiche di..."

Indicatori per Command:

- "Parametrizzare operazioni", "callback"
- "Supporto a undo/redo"
- "Accodare richieste", "transazioni"

Riferimenti del Corso

- **Design Patterns, Elements of Reusable Object Oriented Software**, GoF, 1995, Addison-Wesley
- Design Patterns: http://sourcemaking.com/design_patterns
- Java DP: http://www.javacamp.org/designPattern/
- Deprecating the Observer Pattern: http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf
- GitHub Repository: https://github.com/rcardin/swe

Note Finali

I design pattern comportamentali sono fondamentali per:

- Gestire la **comunicazione tra oggetti**
- Definire **responsabilità** chiare
- Mantenere **basso accoppiamento**
- Favorire **riuso** ed **estensibilità**

La scelta del pattern dipende da:

- Se serve cambiare **intere implementazioni** di algoritmi → **Strategy**
- Se serve variare **passi specifici** di un workflow fisso → **Template Method**
- Se serve un sistema di **notifiche** 1-a-molti → **Observer**
- Se serve **incapsulare richieste** come oggetti → **Command**
- Se serve **attraversare collezioni** senza esporre dettagli → **Iterator**