

# **Design Pattern**

## **Indice**

<b>1 Strutturali</b>	<b>3</b>
1.1 Adapter . . . . .	3
1.2 Decorator . . . . .	4
1.2.1 Utilizzo . . . . .	4
1.3 Facade . . . . .	4
1.4 Proxy . . . . .	5
<b>2 Creazionali</b>	<b>6</b>
2.1 Singleton . . . . .	6
2.2 Builder . . . . .	7
2.2.1 Utilizzo . . . . .	7
2.3 Abstract Factory . . . . .	8
2.3.1 Casi tipici . . . . .	8
<b>3 Comportamentali</b>	<b>9</b>
3.1 Command . . . . .	9
3.1.1 Utilizzo . . . . .	9
3.1.2 Casi tipici . . . . .	10
3.2 Iterator . . . . .	10
3.2.1 Casi tipici . . . . .	11
3.3 Observer . . . . .	11
3.3.1 Utilizzo . . . . .	12
3.3.2 Casi tipici . . . . .	12
3.4 Strategy . . . . .	12
3.4.1 Casi tipici . . . . .	13
3.5 Template Method . . . . .	13
3.5.1 Casi tipici . . . . .	14
3.6 Singleton . . . . .	14
<b>4 Architetturali</b>	<b>14</b>
4.1 Dependency Injection . . . . .	14
4.2 Singleton . . . . .	14

## **Elenco delle figure**

1 Adapter . . . . .	3
2 Decorator . . . . .	4
3 Facade . . . . .	5
4 Proxy . . . . .	6

5	Singleton . . . . .	6
6	Builder . . . . .	7
7	Diagramma di sequenza di un builder . . . . .	8
8	Proxy . . . . .	8
9	Command . . . . .	9
10	Diagramma di sequenza - Command pattern . . . . .	10
11	Iterator . . . . .	10
12	Proxy . . . . .	11
13	Proxy . . . . .	12
14	Strategy . . . . .	13
15	Template method . . . . .	14

# 1 Strutturali

Affrontano i problemi che riguardano la composizione di classi e oggetti, consentendo il riutilizzo di oggetti esistenti, sfruttando l'ereditarietà e l'aggregazione.

## 1.1 Adapter

Viene utilizzato per cambiare l'interfaccia di una classe in un'altra, in modo da poter inserire all'interno delle gerarchie dell'applicazione classi già esistenti o componenti di toolkit.

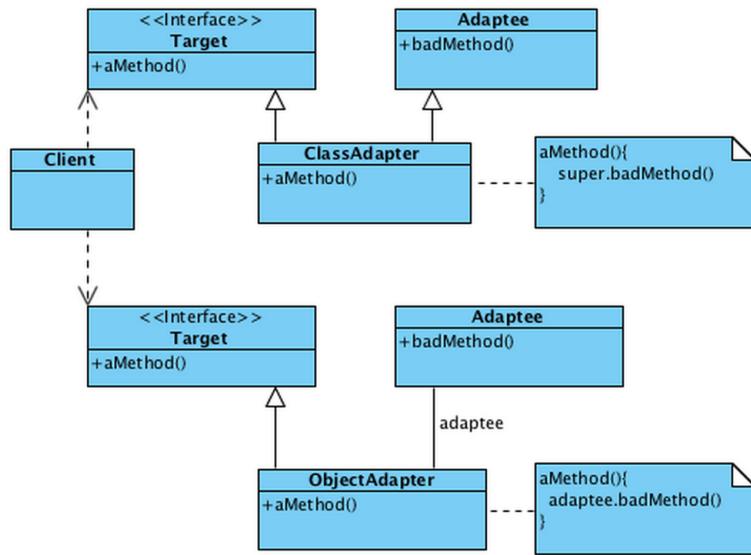


Figura 1: Adapter

Può essere realizzato in due modi distinti:

- **Class Adapter:** in cui la classe Adapter implementa sia l'interfaccia della classe da adattare (Adaptee) sia l'interfaccia Target. Questa modalità non funziona quando bisogna adattare una classe e anche le sue sottoclassi, però permette all'Adapter di modificare alcune caratteristiche dell'Adaptee.
- **Object Adapter:** in cui la classe Adapter implementa solamente l'interfaccia Target e utilizza un'oggetto di tipo Adaptee sul quale esegue le azioni. Questa modalità permette ad Adapter di funzionare sia con Adaptee sia con le sue sottoclassi, tuttavia non è possibile modificare l'oggetto Adaptee.

In ogni caso un oggetto di tipo Adapter non è sottotipo di Adaptee.

## 1.2 Decorator

Viene utilizzato per aggiungere funzionalità ad un oggetto dinamicamente, senza usare il subclassing. Si vuole "wrappare" un componente dentro un decoratore per aggiungergli delle funzionalità. Per rendere più potente il tutto, sfruttando un'interfaccia comune è possibile decorare un decoratore e così via. Al livello implementativo il Component dovrebbe essere stateless e le modifiche apportate dal Decorator dovrebbero essere leggere, in altri casi conviene usare il pattern Strategy, in modo da evitare decoratori costosi da manutenere.

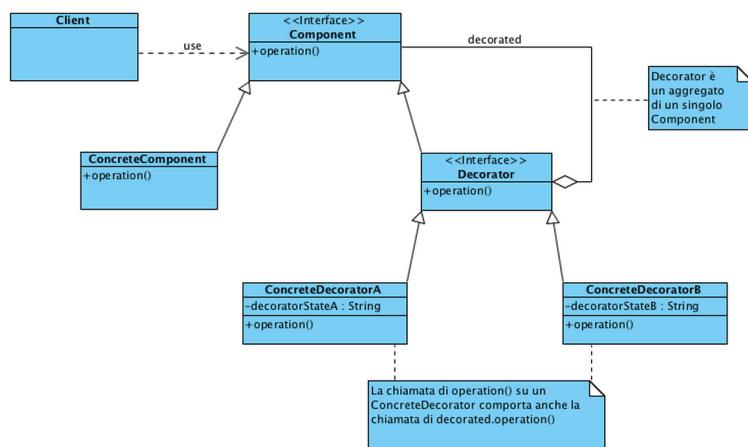


Figura 2: Decorator

### 1.2.1 Utilizzo

Prima creo il **ConcreteComponent**, poi usando creo i vari **Decoratori**, passandogli come riferimento il compoente:

```
Component c = new ConcreteDecoratorB(new ConcreteDecoratorA(new ConcreteComponent));
```

## 1.3 Facade

Viene usato per fornire un'interfaccia unica e semplice per un sotto sistema complesso. In questo modo vengono semplificate le varie dipendenze tra i sotto sistemi, senza nascondere le funzionalità di basso livello. Di conseguenza:

- Diminuiscono le classi del sotto sistema con il quale il client deve interagire;
- C'è un accoppiamento lasco tra i sotto sistemi, senza dipendenze circolari;
- Viene introdotto un single point of failure;

- È necessario prestare attenzione al dimensionamento della classe Facade che non deve essere troppo grande.

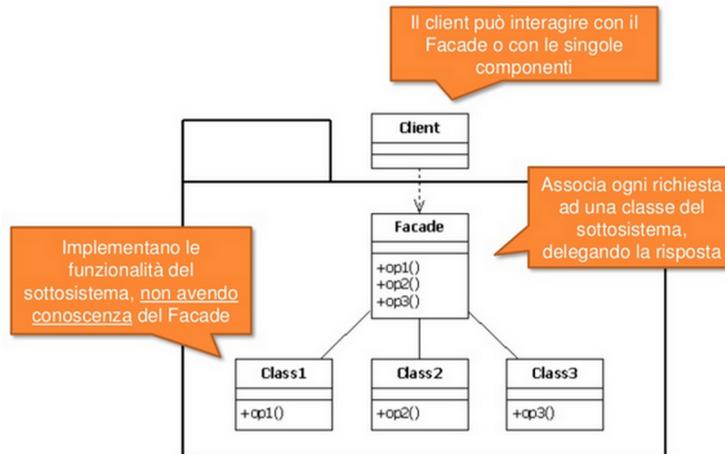


Figura 3: Facade

## 1.4 Proxy

Viene utilizzato per fornire un surrogato di un altro oggetto di cui si vuole controllare l'accesso. Questo surrogato deve quindi agire come l'oggetto che rappresenta e perciò devono condividere la stessa interfaccia. Questo pattern ha diversi utilizzi pratici:

- **Remote proxy:** rappresentazione locale di un oggetto remoto (JavaRMI)
- **Virtual proxy:** creazione ritardata di oggetti (Lazy instantiation)
- **Protection proxy:** controllo degli accessi all'oggetto originale
- **Puntatore intelligente:** per una gestione efficace della memoria (Copy on edit)

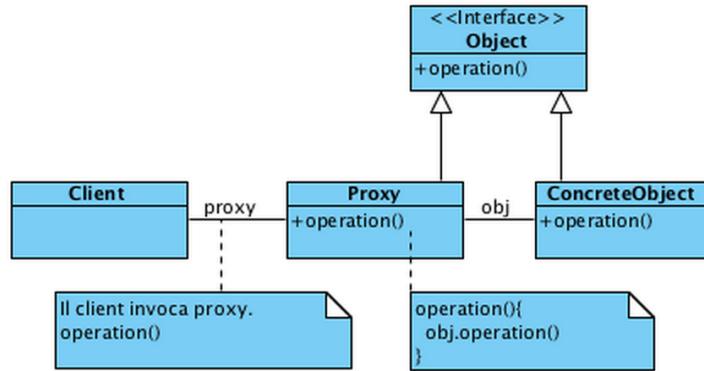


Figura 4: Proxy

## 2 Creazionali

Lo scopo di questa serie di design pattern è quello di rendere un sistema indipendente dall’implementazione delle sue componenti, nascondendo i tipi concreti mediante l’utilizzo di interfacce e classi astratte.

### 2.1 Singleton

Viene utilizzato quando si vuole assicurare l’esistenza di un’unica istanza di una determinata classe e si vuole fornire un unico punto d’accesso globale a questa istanza.

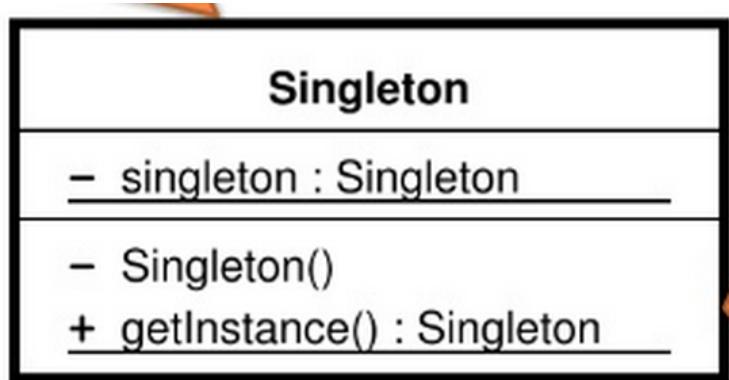


Figura 5: Singleton

La classe viene quindi definita con un costruttore privato e l’unico modo per crearla è tramite un metodo pubblico, che, controlla se esiste già un’istanza e nel caso esista, ritorna un riferimento a quell’istanza anziché creare un nuovo oggetto. Utilizzando questo pattern si riesce ad avere un controllo completo sul numero di

istanze presenti e sul come i vari client possono accedervi, evitando l'utilizzo di variabili globali e rendendo possibile applicare anche il polimorfismo.

## 2.2 Builder

Viene utilizzato per separare la logica di costruzione di un oggetto complesso dalla sua rappresentazione. Permette inoltre di creare dinamicamente dei nuovi oggetti mediante degli algoritmi di creazione che sono facilmente intercambiabili tra loro.

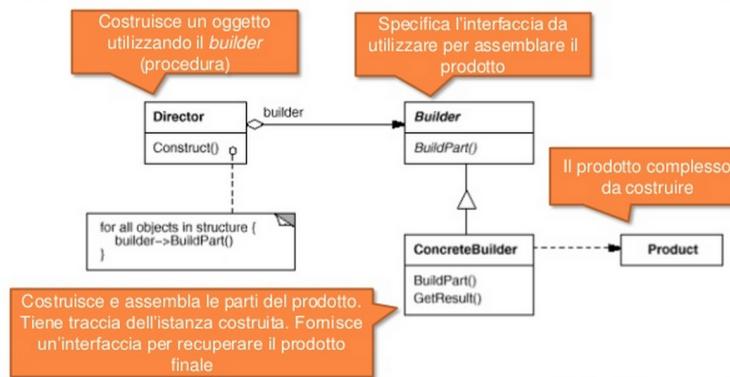


Figura 6: Builder

In questo modo viene isolato il codice di costruzione dell'oggetto, rendendolo più facile da estendere senza far esplodere il numero di parametri da passare al costruttore (telescoping). Inoltre, se vengono aggiunti campi, non è necessario andare a modificare il codice del client, ma le modifiche vengono limitate al director.

### 2.2.1 Utilizzo

1. Il client crea un builder per l'oggetto che deve costruire;
2. Il client crea un director passandogli il builder con cui costruire l'oggetto;
3. Il director crea l'oggetto finale utilizzando i vari metodi messi a disposizione dal builder;
4. Il client ottiene il riferimento all'oggetto finale.

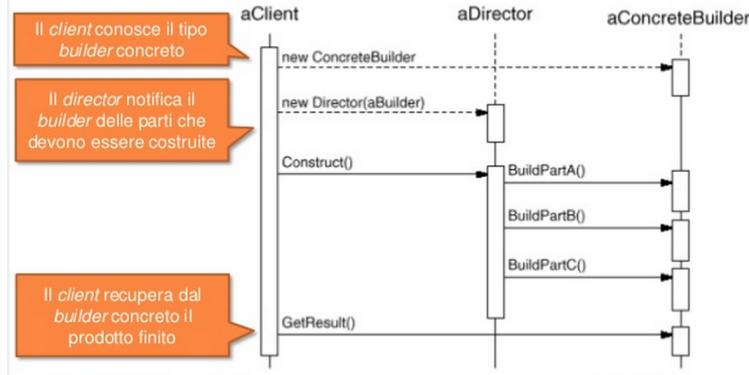


Figura 7: Diagramma di sequenza di un builder

### 2.3 Abstract Factory

Viene utilizzato per fornire la possibilità al client di creare oggetti appartenenti ad una gerarchia di prodotti, senza dover specificare le classi concrete.

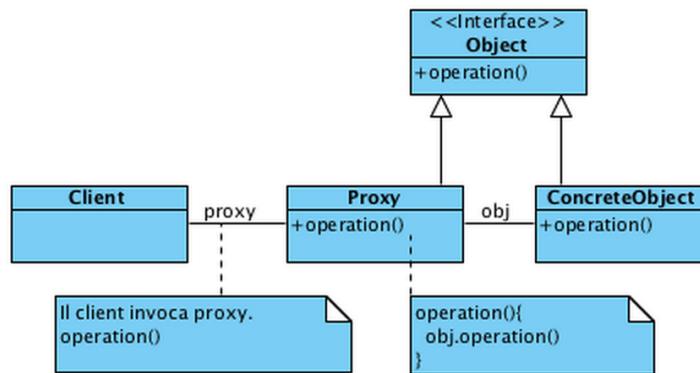


Figura 8: Proxy

Utilizzando questo pattern le classi client possono lavorare semplicemente con le interfacce, senza aver bisogno di conoscere i tipi concreti. Tendenzialmente una Abstract Factory implementa un Singleton.

#### 2.3.1 Casi tipici

- La creazione di oggetti deve essere indipendente dal sistema che li utilizza;
- Il sistema deve poter lavorare con più famiglie degli stessi oggetti;
- Varie famiglie di oggetti devono essere usate assieme;
- Devono essere pubblicate delle librerie e non si vogliono mostrare i dettagli implementativi;

- Il client non deve conoscere le classi concrete con cui lavora.

## 3 Comportamentali

Lo scopo di questa serie di design pattern è quello di poter definire dinamicamente come degli oggetti collaborano tra loro e come svolgono la loro funzione.

### 3.1 Command

Viene utilizzato per encapsulare delle richieste all'interno di un oggetto, in modo che il client siano indipendenti dal tipo di richiesta.

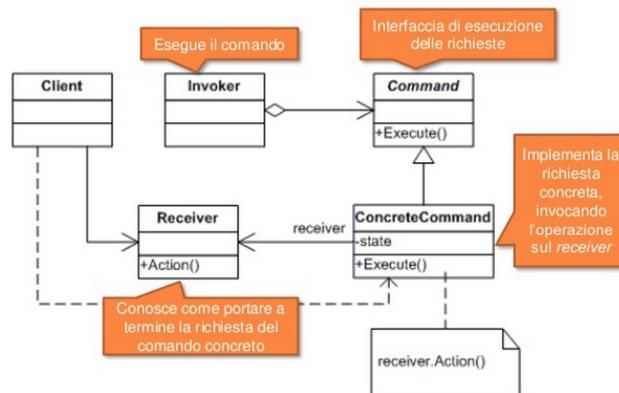


Figura 9: Command

#### 3.1.1 Utilizzo

1. Il client crea il comando da eseguire;
2. Il client passa il comando da eseguire all'invoker (oggetto che si occupa di gestire i comandi);
3. L'invoker esegue il comando, producendo degli effetti sul receiver.

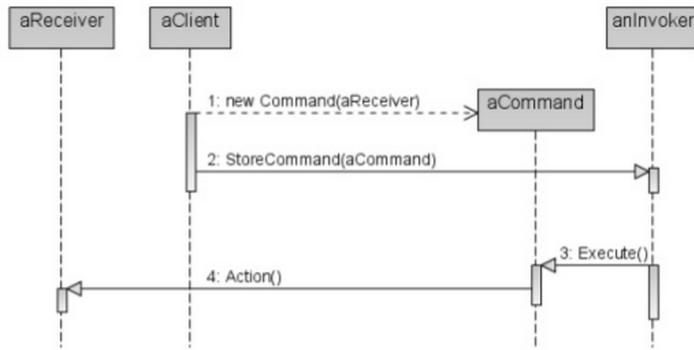


Figura 10: Diagramma di sequenza - Command pattern

### 3.1.2 Casi tipici

- Sono necessarie delle funzionalità di callback;
- Le richieste devono essere gestite in momenti differenti o in ordine diverso;
- È necessario tenere uno storico delle richieste effettuate;
- Tra il chiamante e il ricevente deve esserci un accoppiamento lasco.

## 3.2 Iterator

Viene utilizzato per permettere ad una classe client di effettuare l'accesso sequenziale agli elementi di un aggregato, senza esporre l'implementazione dell'oggetto.

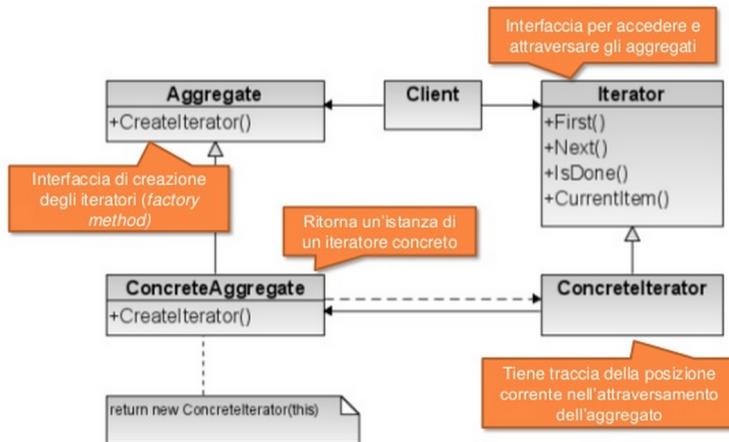


Figura 11: Iterator

Nonostante il concetto di iteratore sia di per sé semplice, quando viene implementato è necessario fare alcune considerazioni riguardo la logica di attraversamento e al comportamento quando vengono aggiunti o tolti dei componenti durante l'uso dell'iteratore. Difatti, il controllo dell'iterazione può essere fatto sia dal client, invocando un metodo dell'iteratore, sia dall'iteratore stesso in modo automatico. Allo stesso modo, l'algoritmo di attraversamento può essere definito dall'aggregato oppure dall'iteratore stesso.

### 3.2.1 Casi tipici

- È necessario accedere agli elementi di una collezione senza sapere come sono memorizzati;
- C'è bisogno di poter attraversare la collezione più molte, anche in modo concorrente;
- Si vuole fornire un'interfaccia uniforme di attraversamento;
- Ci possono essere più modi per attraversare la stessa collezione.

## 3.3 Observer

Viene utilizzato per stabilire una relazione 1 a n tra un oggetto e altri vari oggetti che dipendono da esso. Ogni volta che l'oggetto osservato viene modificato, questo si preoccupa di comunicarlo agli oggetti che lo osservano.

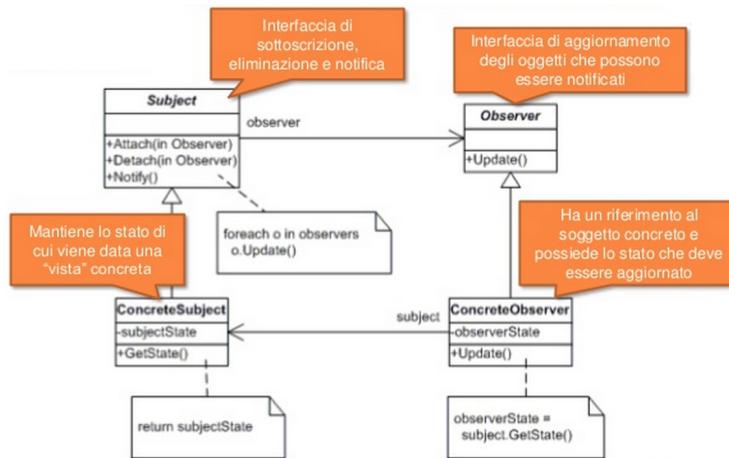


Figura 12: Proxy

Possono essere fatte versioni più smart, dove l'oggetto osservato comunica anche che cosa è stato modificato oppure avverte solo alcuni oggetti registrati, in modo da evitare chiamate a funzioni inutili.

### 3.3.1 Utilizzo

1. Il soggetto viene modificato;
2. Il soggetto notifica tutti gli osservatori registrati;
3. I vari osservatori si sincronizzano sui cambiamenti subiti dal soggetto.

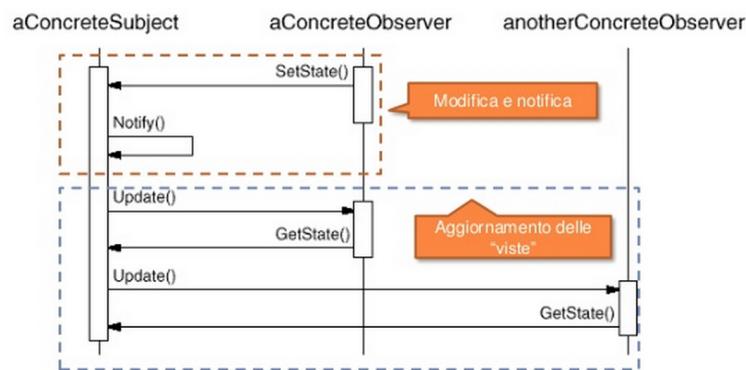


Figura 13: Proxy

### 3.3.2 Casi tipici

- Quando è necessario che al cambiamento di stato di un oggetto, altri oggetti eseguano delle azioni;
- Quando è richiesto di fare il broadcast di un evento;
- Per ottenere un accoppiamento “astratto” tra soggetti e osservatori.

## 3.4 Strategy

Viene utilizzato quando si ha un gruppo di algoritmi e si vuole poter scegliere l'algoritmo da adottare a runtime.

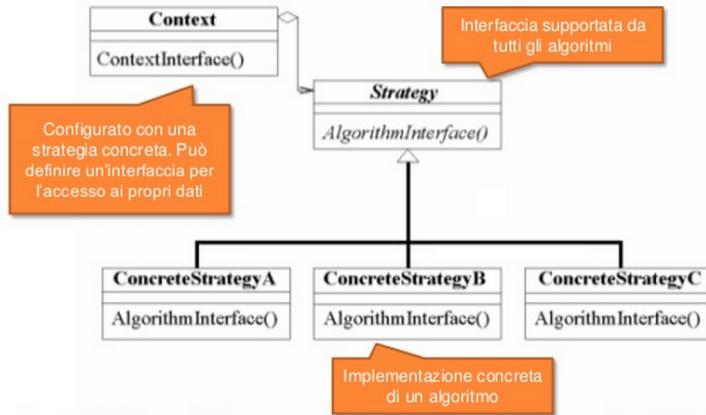


Figura 14: Strategy

In questo modo è anche più semplice inserire un nuovo algoritmo più efficiente, infatti, basta creare una nuova classe che lo implementa.

### 3.4.1 Casi tipici

- L'unica differenza tra varie classi correlate è il comportamento;
- Sono necessarie più versioni di uno stesso algoritmo;
- Gli algoritmi accedono o utilizzano dei dati che il codice client non deve conoscere;
- Il comportamento di una classe deve essere definito a runtime.
- Quando una classe definisce una serie di differenti comportamenti condizionali<sup>1</sup>.

## 3.5 Template Method

Viene utilizzato per definire un algoritmo templatizzato, fornendo una sequenza di operazioni ma senza specificare come queste vengono svolte, sarà compito delle sottoclassi definire queste operazioni. Differisce dallo Strategy in quanto, in questo caso l'algoritmo è uno solo che deve essere implementato, mentre nello Strategy ci sono più algoritmi differenti.

---

<sup>1</sup>Quando ci sono tanti if per scegliere l'algoritmo effettivo

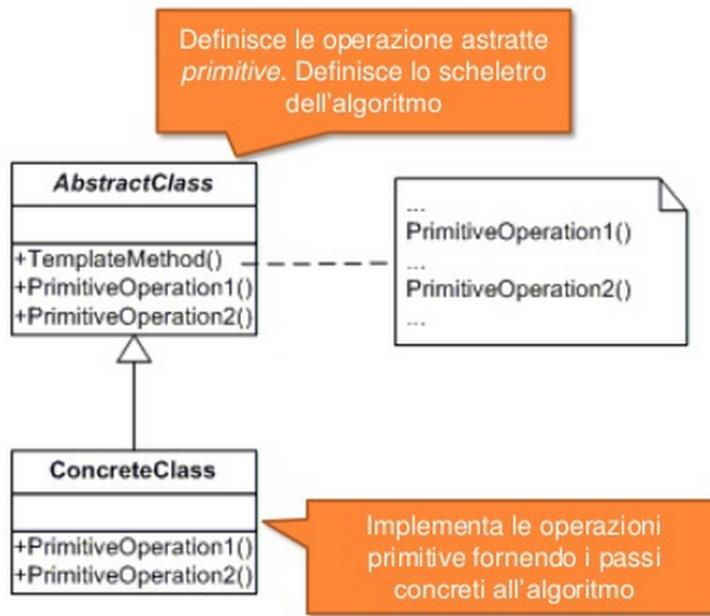


Figura 15: Template method

### 3.5.1 Casi tipici

- È necessaria un'unica implementazione astratta di un algoritmo;
- Si vuole raccogliere un comportamento comune a più classi in un'unica classe;
- La classe padre deve essere in grado di eseguire i metodi delle classi figlie;
- Tutte o quasi tutte le sottoclassi devono implementare lo stesso comportamento.

## 3.6 Singleton

# 4 Architetturali

## 4.1 Dependency Injection

Viene utilizzato per separare dalla logica di funzionamento di un componente la risoluzione delle dipendenze con gli altri oggetti. In questo modo è possibile realizzare un *inversion of control*, con un contenitore esterno che si occupa di gestire il ciclo di vita degli oggetti dell'applicazione.

Le dipendenze con gli oggetti esterni vengono quindi *iniettate* dal contenitore esterno, ottenendo così una riduzione delle dipendenze e rendendo più facili i test.

La dependency injection può essere fatta in due modi:

- **Constructor injection:** le dipendenze vengono iniettate passando gli oggetti come parametri del costruttore. In questo modo l'oggetto è subito utilizzabile appena viene costruito, c'è però il rischio di *telescoping* sui parametri del costruttore.
- **Setter Injection:** le dipendenze vengono iniettate passando gli oggetti mediante l'utilizzo di metodi *setter*. Così viene evitato il telescoping ma per un po' di tempo dopo la creazione, l'oggetto finale rimane in uno stato inconsistente.

#### 4.1.1 Utilizzo

Un sacco di framework moderni lo utilizzano:

- AngularJS;
- Spring;
- Google Guice.

### 4.2 Singleton