Indice

1 Lambda expression **3**
- 1.1 Intro . . . 3
- 1.2 Lambda expression . . . 4
- 1.3 Method references . . . 5
- 1.4 Scope di una lambda . . . 5

2 Stream **6**
- 2.1 Creazione di uno stream . . . 6
  - 2.1.1 Stream infiniti . . . 6
- 2.2 Operazioni intermedie . . . 7
- 2.3 Operazione di riduzione di uno stream . . . 7
  - 2.3.1 Optional . . . 7
- 2.4 Operazione di raccolta di uno stream . . . 7

3 INFORMATION HIDING **9**
- 3.1 Design Pattern VS Design Principle . . . 10

4 ACRONIMO SOLID **11**
- 4.1 Single Responsability Principle (SRP) . . . 11
- 4.2 Open-Closed Principle (OCP) . . . 11
- 4.3 Liskow Substitution Principle (LSP) . . . 13
- 4.4 Iterafce Segregation Principle (ISP) . . . 15
- 4.5 Dependency Inversion Principle (DIP) . . . 15

5 COVARIANZA E CONTROVARIANZA **17**

6 DESIGN PATTERN **20**
- 6.1 Notazione Uml . . . 20
- 6.2 Interaction diagram . . . 20

7 TEMPLATE E STRATEGY **21**
- 7.1 Template Method . . . 21
  - 7.1.1 Struttura e partecipanti . . . 21
- 7.2 Strategy . . . 22
  - 7.2.1 Conseguenza . . . 23
  - 7.2.2 Struttura e partecipanti . . . 23
  - 7.2.3 Differenza con il templateMethod . . . 23
  - 7.2.4 Strategy e classi anonime . . . 24
  - 7.2.5 Strategy e lambda . . . 24

8 COMPOSITE **25**
- 8.1 Struttura . . . 25

9 ITERATOR **27**
- 9.1 Struttura . . . 28
- 9.2 Iterator e classi anonime . . . 28
- 9.3 Iterator vs Stream . . . 28

10 PATTERN CREAZIONALI **29**
- 10.1 PERCHE'? . . . 29
- 10.2 Factory Method . . . 29
  - 10.2.1 Struttura . . . 29
  - 10.2.2 Factory Method vs Template Method . . . 29
  - 10.2.3 Principale utilizzo . . . 30
  - 10.2.4 Vantaggi e svantaggi . . . 30
- 10.3 Abstract Factory . . . 30
  - 10.3.1 Struttura . . . 30
  - 10.3.2 Principale utilizzo . . . 31
  - 10.3.3 Accedere e modificare la factory . . . 31
  - 10.3.4 AbstractFactory vs Factory Method . . . 31
- 10.4 Static Factory Methods . . . 31
- 10.5 Singleton . . . 32

10.6 Fluent Interface . . . 32
10.7 Builder . . . 32

11 OBSERVER **34**
11.1 Struttura . . . 34
11.2 Interazione . . . 34
11.3 Conseguenze . . . 34
11.4 Notifica . . . 35
11.5 Svantaggio . . . 35

12 PUBLISH SUBSCRIBER **36**
12.1 Publish subscribe vs Observer . . . 36

13 DECORATOR **37**
13.1 Struttura . . . 37
13.2 Conseguenza . . . 37
13.3 Decorator vs Strategy . . . 37
13.4 Implementazione senza e con l'entità AbstractDecorator . . . 38
13.5 L'inoltro . . . 38
13.6 Lambda . . . 38

14 CHAIN OF RESPONSABILITY **39**
14.1 Applicabilità . . . 39
14.2 Struttura . . . 39
14.3 Conseguenze . . . 39
14.4 L'inoltro . . . 39
14.5 Decorator vs Chain of responsability . . . 39
14.6 Lambda . . . 40

15 Visitor **41**
15.1 Funzionamento . . . 41
15.2 Double dispatch . . . 41
15.3 Collaborazioni . . . 42
15.4 Varianti Visitor . . . 42
15.4.1 Visitor generico . . . 42
15.4.2 Visitor void . . . 42
15.4.3 Coseguenze . . . 42

16 ADAPTER **43**
16.1 Applicabilità . . . 43
16.2 Struttura . . . 43
16.3 Conseguenze . . . 43
16.4 Altro utilizzo di Adapter . . . 44
16.5 Adapter vs default methods . . . 44

17 FACADE **45**
17.1 Esempio Visitor con Expression . . . 45
17.2 Altro caso di utilizzo . . . 46

Lambda expression

1.1 Intro

Per comprendere ed usare le lambda, bisogna partire dalle *interfacce*.
Un'interfaccia è un meccanismo per definire un *contratto* tra due parti, il fornitore del servizio (l'interfaccia stessa) e le classi che vogliono che i loro oggetti siano utilizzabili con quel servizio.
Prendiamo ad esempio l'interfaccia Comparable<T>, avente un metodo, compareTo(T o), che restiruisce un intero e assicura di confrontare solo oggetti dello stesso tipo.
Se una classe decidesse di implementare questa interfaccia, quindi fornire un'implementazione del metodo, allora i suoi oggetti potrebbero essere ordinati da java.
Il contratto è che x.compareTo(y) deve restituire:

- un intero positivo se x viene dopo di y;
- un intero negativo nel caso contrario;
- 0 altrimenti.

N.B. La classe String implementa di suo questa interfaccia e implementa compareTo con il confronto lessicografico.
Questo spezzone di codice funziona in quanto *Arrays.sort* riesce a ordinare oggetti la cui classe implementa Comparable e, come detto prima, String la implementa.

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends); // friends is now ["Mary", "Paul", "Peter"]
System.out.println(Arrays.toString(friends));
```

Così come potremmo ordinare oggetti *Employee* in base al loro nome

```
public class Employee implements Comparable<Employee> {
    private String name;

    @Override
    public int compareTo(Employee other) {
        return name.compareTo(other.getName());
    }
}
```

dove, in questo caso, il compareTo di Employee delega il confronto al compareTo di String.
Se volessimo ordinare Employee/String con un altro criterio, non potremmo farlo in quanto non sarebbe possibile definire due metodi compareTo e non sarebbe possibile modificare la classe java.
Esiste una variante di Arrays.sort che oltre ad accettare una lista da ordinare, accetta un'altra interfaccia, Comparator<T>, avente un metodo compare(T o1, T o2) che restituisce un intero.
Quindi per definire un nuovo criterio, dovremo definire una nuova classe, che implementa Comparator, e passarla al metodo Arrays.sort.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new LengthComparator());
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
...
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

Se il metodo di confronto ci servisse solo in quel punto di SortDemo, saremmo comunque costretti a definire una classe e istanziarla.
Per questo motivo ci sono le *classi anonime*, un meccanismo che riduce la verbosità del codice

- che permette di dichiarare e istanziare una classe allo stesso tempo;

- sono simile alle classi locali, solo che non hanno un nome;
- la loro invocazione avviene come quella di un costruttore, solo che al suo interno c'è una classe vera e propria.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new Comparator<String>() {
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
        });
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
```

Quindi, fino ad ora, abbiamo visto due interfacce, avente un singolo metodo, il contratto che stiamo usando dipende dal quel singolo metodo e se non ci fosse bisogno di mantenersi uno stato per implementare quel metodo, allora sarebbe più comodo poter specificare solo quel singolo blocco di codice invece di creare una classe che implementa l'interfaccia e istanziarla o creare una classe anonima.

1.2 Lambda expression

E' una *funzione anonima*, un blocco di codice che può essere passato, assegnato, restituito in modo da essere eseguito in un secondo momento, una o più volte.
I valori gestiti sono *funzioni* e non oggetti, in java una funzione è un'istanza di un oggetto che implementa una certa interfaccia.
Nell'esempio di LengthComparator, a noi basterebbe dire che, per confrontare due stringhe, bisogna usare il blocco di codice di compare, specificando che first e second sono oggetti di tipo String.
Quindi dovremmo passare ad Arrays.sort una funzione che, dati due oggetti String, restituisce
first.length() − second.length().
In java, la sintassi per definire questa funzione è

```
(String first, String second) -> first.length() - second.length()
```

che risulta essere la nostra lambda expression.
Quindi, nel metodo di Arrays.sort, invece di passare un'istaza di una classe che implementa Comparator o una classe anonima, gli passiamo la lambda.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new Comparator<String>() {
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
        });
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
...
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, (String first, String second) -> first.length() - second.length());
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
```

Il body di una lambda viene eseguito non quando viene passata al metodo sort ma quando bisogna effetivamente confrontare gli oggetto (*esecuzione differita*) e se il body dovesse aver bisogno di più righe, allora si userebbero le parentesi graffe e il return.

Java può inferire il tipo dei parametri della lambda dal contesto, in tal caso si possono omettere i tipi, stessa cosa per il tipo di ritorno anche se qui java fa un controllo che sia utilizzabile nel contesto in cui viene usata la lambda.
Si può assegnare/passare una lambda quando ci si aspetta un oggetto dichiarato di tipo interfaccia

- che ha un singolo metodo astratto;
- purché la lambda sia compatibile con tale metodo, considerando il tipo dei parametri della lambda, che devono essere compatibili coi parametri del metodo, e del tipo inferito del body della lambda che deve essere compatibile col tipo di ritorno del metodo.

Una tale interfaccia è detta *interfaccia funzionale* o *SAM* (Single Abstract Method).

1.3 Method references

Il codice che si scrive in una lambda expression richiama semplicemente un metodo che è già implementato, in questi casi, invece di passare/assegnare una lambda che chiama semplicemente quel metodo passandogli i parametri della lambda, si passa/assegna un riferimento a quel metodo (*method reference*), attraverso la notazione '::'.
Abbiamo tre tipi di method reference

- Class::instanceMethod dove il primo parametro diventa il ricevente del metodo, gli altri sono passati al metodo

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));

Arrays.sort(strings, String::compareToIgnoreCase);
```

- Class::staticMethod dove tutti parametri sono passati al metodo statico

```
list.removeIf(x -> Objects.isNull(x));

list.removeIf(Objects::isNull);
```

- Object:instanceMethod dove il metodo viene richiamato sull'object specificato prima dei :: mentre gli altri parametri sono passati al metodo

```
strings.forEach(x -> System.out.println(x));

strings.forEach(System.out::println);
```

Con i method reference si usa uno stile più dichiarativo, si scrive meno codice e lo si legge meglio.

1.4 Scope di una lambda

Un'interfaccia funzionale può avere tanti metodi statici e di default ma basta che abbiamo un singolo metodo astratto.
Un'interfaccia conviene annotarla con il tag @FunctionalInterface, così facendo il compilatore controllerà che il vincolo sia rispettato e gli altri utenti sapranno che quell'interfaccia è pensata come funzionale.
Non possiamo dichiarare variabili locali in una lambda o parametri di una lambda con lo stesso nome di variabili già definite nei blocchi esterni.
Una lambda può riferirsi a variabili definite NON dentro la lambda purchè dichiarate final o effectively final, si dice che ha un *ambito di visibilità circostante* (enclosing scope), per esempio

```
String message = "Hello ";
repeat(10, (x) -> System.out.println(message + x));
```

La lambda si riferisce alla variabile 'message' ma il body della lambda sarà effettivamente eseguito da dentro il metodo repeat e, da dentro il metodo repeat, la variabile 'message' non è visibile, eppure il codice è lecito e tutto funziona.
Una lambda ha tre ingredienti, parametri, body e i valori delle *variabili libere*, ovvero parametri che non fanno parte dei parametri della lambda e che non fanno parte delle variabili dichiarate nel blocco di codice della lambda.
Nell'esempio di prima, 'x' non è una variabile libera, è legata al parametro della lambda, mentre 'message' si.
Una lambda expression cattura il valore di queste variabili libere e, quando viene eseguita, il body è chiuso rispetto ad esse ed è per questo motivo che a runtime una lambda è detta *chiusura* (closure).
Ovvero prima di passare la lambda a repeat, è come se java sostituisse message direttamente con "Hello ".

Stream

Una “vista” dei dati per specificare computazioni a un più alto livello concettuale rispetto alle collezioni e gli iteratori.
Supponiamo di voler calcolare la media dei salari di una collezione di oggetti Employee.

Se decidessimo di usare gli iteratori

- dovremmo dichiarare una variabile per accumulare i risultati intermedi;
- iterare sulla sorgente dati, dove, ad ogni iterazione, aggiorniamo i risultati intermedi;
- alla fine, calcolare la media.

Se usassimo gli stream, basterebbe

- specificare la sorgente dati;
- la proprietà di interesse;
- cosa vogliamo fare con quella proprietà.

La libreria stream farà tutto il resto ottimizzando il calcolo.

Rispettano il principio ” “what, not how”, ovvero si specifica cosa si vuole fare e non il come deve essere fatto.
Stream e collezioni, superficialmente, sembrano simili, entrambi permettono di trasformare e recuperare dati ma

- uno stream non *memorizza* i dati, sono memorizzati nella collezione originale o generati su richiesta;
- uno stream non *modifica* i dati originali, ma genera un nuovo stream dove alcuni elementi dello stream precendete non sono presenti nello stream corrente;
- le operazioni sono ”*lazy*” quanto il più possibile, ovvero non vengono eseguite finché non serve il risultato (si possono avere anche stream infiniti).

Gli stream si basano sul *method chaining*, ovvero chiamano un metodo sul risultato di un altro metodo, senza memorizzare i risultati intermedi.
Il workflow tipico di uno stream consiste nel creare una “pipeline” di operazioni in 3 fasi:

- creazione dello stream;
- specifica delle operazioni intermedie;
- applicazione di un’operazione finale di riduzione per produrre un risultato oppure un’operazione di raccolta.

2.1 Creazione di uno stream

Gli stream si creano da collezioni e array o usando generatori o iteratori.
Per le collezioni abbiamo i metodi stream() e parallelStream(), per gli array o un numero arbitrario di argomenti (vararg) abbiamo il metodo statico Stream.of() e poi abbiamo i metodi Stream.generate(Supplier <T>) e Stream.iterate(T seed, UnaryOperator<T> f).
Gli ultimi due metodi possono genera stream infiniti.

2.1.1 Stream infiniti

Stream.generate(Supplier <T>) prende una lambda senza argomenti e viene chiamato solo quando viene richiesto allo stream il prossimo elemento

```
Stream<String> echos = Stream.generate(() -> "Echo");
Stream<Double> randoms = Stream.generate(Math::random);
```

Stream.iterate(T seed, UnaryOperator<T> f) dove seed è il ”seme” iniziale mentre f è una funzione che sarà applicata al valore precendete

```
Stream<BigInteger> integers = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE))
```

2.2 Operazioni intermedie

Le operazioni intermedie di uno stream sono metodi che trasformano lo stream in un altro stream.
Tra questi metodi troviamo filter() che si occupa di filtrare uno stream, map() che si occupa di trasformare gli elementi di uno stream, limit() che limita ad un certo numero di elementi dello stream (occhio a dove lo mettiamo), skip() che salta determinati elementi dello stream, distinct() che scarta i duplicati, etc...
Con gli stream si scrive meno codice, lo si rende più leggibile ma allo stesso tempo è molto facile commettere errori, i due prossimi stream forniscono un risultato differente

```
#SI LIMITA ALLE PRIME 5 STRINGHE CHE RISPETTANO IL FILTRO
long count = words.stream()
                  .filter(w -> w.length() > 12)
                  .limit(5)
                  .count();
#SI LIMITA ALLE PRIME 5 STRINGHE
long count = words.stream()
                  .limit(5)
                  .filter(w -> w.length() > 12)
                  .count();
```

2.3 Operazione di riduzione di uno stream

Una volta applicate tutte le operazioni di trasformazione, alla fine si deve applicare un'operazione di riduzione.
Alcuni metodi di riduzione sono min() e max() che restituiscono, rispettivamente, il minimo ed il massimo di uno stream in base ad un criterio oppure il metodo reduce(...) che prende una funziona binaria e continua ad applicarla partendo dai primi due elementi.

```
List<Integer> values = ...;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

Le operazioni di raccolta/riduzione forzeranno l'esecuzione di tutte le operazioni, lazy, precedenti e dopo lo stream non potrà più essere usato.
Alcuni di queste operazioni, come reduce(...), restituiscono un oggetto di classe Optional, un'alternativa più sicura della gestione dei valori null come ad esempio findFirst(), ifPresent(), etc...

2.3.1 Optional

Un oggetto Optional<T> è un wrapper di un oggetto T oppure nessun oggetto.
L'idea di base sull'utilizzo di un oggetto Optional è quella di usare i suoi metodi che permettono di produrre un'alternativa se il valore non è presente o consumarlo.
Questo oggetto deve essere usato in modo appropriato, altrimenti si hanno gli stessi problemi che si hanno con null, ad esempio

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod();

T value = ...;
value.someMethod();
```

Questi due blocchi sono uguali, se optionalValue non contenesse alcun valore, allora avremo una NPE, stessa cosa per quest'altro esempio

```
if (optionalValue.isPresent())
    optionalValue.get().someMethod();

if (value != null)
    value.someMethod();
```

2.4 Operazione di raccolta di uno stream

Un'alternativa alla riduzione di uno stream è la raccolta.

Ci sono molti metodi tra cui il metodo collect() che prende in input oggetti che implementano l'interfaccia Collector, tra cui Collectors che fornisce metodi per creare istanze delle collezioni più comuni come ad esempio toList() o toSet(), iterator() che ritorna un iteratore o anche toArray().
Spesso si vuole una mappa che associa ad una chiave, una collezione di oggetti e qui entrano in gioco i metodi di raggruppamento o partizionamento.
Si usa il primo specificando la lambda che calcola la chiave, detta *classifier function* mentre si usa il secondo quando quest'ultima è booleana

```
#per ogni stringa del nome abbiamo la lista delle relative persone
Map<String, List<Person>> sameName = people.collect(Collectors.groupingBy(Person::getName));

#abbiamo due liste, una che contiene il nome e una che non lo contiene
Map<Boolean, List<Person>> emptyNames = people.collect(Collectors.partitioningBy(p ->
    p.getName().isEmpty()));
```

Di default groupingBy raggruppa in una List ma ci sono altre versioni che permettono di specificare un downstream collector

```
#visto prima
Map<String, List<Person>> sameName = people.collect(Collectors.groupingBy(Person::getName));

#nuovo tipo
Map<String, Long> sameName = people.collect(Collectors.groupingBy(Person::getName, Collectors.counting()));
```

INFORMATION HIDING

Utilizzo di meccanismi per evitare che tutti possano accedere a certe parti di codice.
Noi abbiamo visto i livelli di accessibilità in Java

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Per convenzione

- le variabili d'instanza dovrebbero essere private (evita anche protected);
- i campi static, per rappresentare costanti accessibili a tutti, dovrebbero essere public final;
- i metodi che utilizzano dettagli interni dovrebbero essere private;
- i metodi che potrebbero servire alle sottoclassi, dovrebbero essere protected;
- i metodi che saranno usati dalle altre classi, dovranno essere public.

Creare un oggetto che ha parametri d'instanza, privati, che però vi si possonono accedere, con un get, o settare, tramite un setter, è la cosa sbagliata.
Supponiamo di avere una classe Person tale che

```
  public class Person {
    private String name;

    public Person(String name) {
      this.name = name;
    }

    @Override
    public String toString() {
      return "Person [name=" + name + "]";
    }
}
```

Se instaziassi la classe e cercassi di accedere al campo name, il compilatore mi segnalerebbe errore.
L'information hiding è pensato per essere usato a tempo di compilazione e non a run time.
Java permette di accedere alle variabili, anche private, di una classe.
infatti

```
  @Test public void testReadPrivateMember() throws ... {
    Person person = new Person("John");
    // riferimento al campo (anche se privato)
    Field nameField = person.getClass().getDeclaredField("name");
    // aggira il controllo di accesso
    nameField.setAccessible(true);
    // legge il valore del campo
    Object nameValue = nameField.get(person);
    assertEquals("John", nameValue);
  }
```

N.B. Il metodo setAccessible(boolean bol) permette, effettivamente di accedere ai campi private, altrimenti non sarebbe possibile farlo.
Se io dichiarassi name public, legittimerei il suo uso da parte dei cliente, imponendomi il vincoli non poter mai più modificare quella variabile, altrimenti i client verranno rotti (non compileranno più).

Ogni client che userà quel membro sarà strettamente accoppiato (tightly coupled) a quel membro (noi questo non lo vogliamo).
Stessa cosa per protected, solo che qui saranno le sottoclassi ad essere tightly coupled.
Dichiarando un membro private, lo nascondo al "mondo", non legittimiamo il suo uso da parte dei client.
Il compilatore controllerà il corretto uso nei vari client e

- se compileranno con errori di accesso, significa che i client non sono legittimati all'uso;
- se accederanno al membro via reflection (run time) e qualcosa non dovesse funziona, sarà colpa loro.

Più dettagli interni nascondiamo, più avremo disaccoppiamento e più sarà facile testare singole componenti in isolamento.
Se diachiarassimo qualcosa come public/protected, come le API, e volessimo effettuare dei cambiamento, dovremmo notificare i client quando rilasceremo il nuovo aggiornamento, specificando se si tratta di qualcosa che rompe API, implementa qualcosa di nuovo oppure la risoluzione di bug.
Un modo per indicare questi cambiamenti è l'utilizzo della semantic versioning, un numero composta da tre parti, $X.Y.Z$, dove se

- dovesse cambiare X, significherebbe aver introdotto qualcosa che rome l'API;
- dovesse cambiare Y, significherebbe aver introdotto qualcosa di nuovo;
- dovesse cambiare Z, significherebbe aver risolto dei bug (sperando di non aver introdotto nuovi bug).

3.1 Design Pattern VS Design Principle

Un design pattern sbalisce la linea guida su cosa è giusto e su cosa è sbagliato quando si fa il design di un'applicazione, dice cosa fare (e non fare) e non come farlo.
Un design principle è una soluzione generica e riusabile per un problema comune, dicono come risolvere un problema in un certo contesto software, fornendo chiare linee guida.

ACRONIMO SOLID

I principi SOLID prescrivono come organizzare funzioni e dati in classi e come tali classi dovrebbero essere interconnesse.
Lo scopo di questi principi è la creazione di strutture software che tollerano il cambiamento, sono facili da comprendere/testare e sono riusabili in diversi sistemi software.

4.1 Single Responsability Principle (SRP)

Prevede che una classe deve avere una sola responsabilità, ovvero deve avere una e una sola ragione per essere modificata.
Si dice che la classe deve essere

- *coesiva*, ovvero quanto le cose di un certo gruppo hanno ragione di stare insieme;
- *loosely coupling*, ovvero deve dipendere il meno possibile dai metodi di altre classi.

Se una classe è responsabile di più cose, la si dovrà cambiare più spesso e cambiare frequentemente una classe porta ad avere ripercussioni su tutte quelle classi che dipendono da lei.
Ciò comporterà, ad ogni modifica, nuova ricompliazione e nuovo test dei metodi.
Prima di aggiungere qualcosa di nuovo a una classe ci si dovrebbe interrogare su quale sia la responsabilità di questa classe, se la risposta comprende funzionalità scorrelate congiunte con un "e" o un "oppure" allora, probabilmente, si sta violando l'SRP.

4.2 Open-Closed Principle (OCP)

Il principio aperto/chiuso stabilisce che le classi debbano essere aperte alle estensioni e chiuse alle modifiche.
Per modifica si intende il cambiamento del codice di una classe esistente ed estensione significa aggiungere nuove funzionalità.
Tutto ciò sta a significare che dovremmo essere in grado di aggiungere nuove funzionalità senza toccare il codice attuale della classe, questo perché ogni volta che modifichiamo il codice, rischiamo di dare vita a potenziali bug.
Se nell'SRP si cerca di decomporre la responsabilità, qui si cerca di capire quali parti devono essere concrete (da implementare) e quali devono essere astratte (saranno implementate dai consumatori del software).
Negli OOP abbiamo a disposizione il meccanismo dell'ereditarità per creare una classe derivata da una già esistente, il meccanismo dell'override per ridefinire un metodo della classe padre e la funzionalità del binding dinamico che permette, dato un oggetto, di chiamare il metodo giusto a runtime.
Questo, però, non è detto che basti a rispettare OCP, specialmente se si estende una classe concreta o se la classe derivata si basa, fortemente, su dettagli implementatitivi della superclasse. Per esempio, voglio estendere una classe, MySet, per contare gli elementi inseriti

```
public class MySet<E> {
  public void add(E o) {
  // add the element to the internal set
  }

  public void addAll(Collection<E> c) {
    for (E e : c) {
      add(e);
    }
  }
}
```

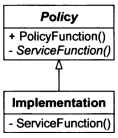
```
public class MyCountingSet<E> extends MySet<E> {
  private int count = 0;

  @Override
  public void add(E o) {
    super.add(o);
    count++;
  }
}
```

Se io modificassi MySet, ad esempio modifico addAll(), non chiamando più dentro add(), allora la mia classe derivata non sarebbe più corretta, in quanto non avremo più un aumento del contatore.
Quindi, per risolvere questo problema, devo modificare la mia classe, facendo override anche del secondo metodo

```
public class MySet<E> {
  public void add(E o) {
  // add the element to the internal set
  }

  public void addAll(Collection<E> c) {
    // add directly all elements
    // WITHOUT relying on add
  }
}
```

```
public class MyCountingSet<E> extends MySet<E> {
  private int count = 0;

  @Override
  public void add(E o) {
    super.add(o);
    count++;
  }

  @Override
  public void addAll(Collection<E> c) {
    super.addAll(c);
    count += c.size();
  }
}
```

Inoltre, se ad un certo punto tornassi alla versione originale di MySet, avrei lo stesso problema, ovvero MyCountingSet non funzionerebbe più in quanto il contatore verrebbe incrementato due volte ogni volta.
Quindi possiamo vedere che le modifiche fatte a MySet possono anche essere giuste o minimali ma sono devastanti per la mia sottoclasse.
Questo porta al concetto della **fragile base class problem**, ovvero una classe è considerata fragile se piccole modifiche corrette e sicure alla classe base, possono portare le classi derivate a non funzionare più correttamente.
Quindi sarebbe meglio proibire l'estensione della classe (con un final), proibire la ridefinizione di alcuni metodi (anche qui final) e documentare come una classe base dovrebbe essere usata/estesa.
Allora, dal concetto di classi concrete ed ereditarietà, arriviamo al concetto di classi astratte o interfacce ed, eventualmente, al principio di composizione e delega.
Prediamo come esempio la classe astratta Policy che ha un metodo, public e probabilmente final, PolicyFunction che, a sua volta, implementa una certa politica in termini di un metodo, protetto e astratto, ServiceFunction.
Policy, quindi, ha una parte concreta (PolicyFunction) ed una astratta (ServiceFunction), si dice che è aperta all'estensione, tramite la parte astratta e chiuse alle modifiche dovute al fatto che PolicyFunction è final.



Supponiamo di avere una classe Client che si occupa di fare logging e di voler aggiungere un nuovo tipo di log, basterà, quindi, aggiungere un nuovo tipo di logging ad enum ed un nuovo ramo allo if-else.
Come si vede, la classe è si configurabile rispetto al log ma non è aperta ad estensioni del log in quanto la sua gestione dipende da un if-else/switch e quanfo si usa un if-else/switch, si sta violando OCP, inoltre c'è il dubbio su cosa faccia l'ultimo else (in questo caso non fa nulla).

```
public class Client {
  public enum LogConf { CONSOLE, FILE }
  private LogConf logConf;

  public Client(LogConf logConf) { this.logConf = logConf; }

  public void aMethod() {
    // do something before or after the log method...
    log("a message");
  }

  private void log(String message) {
    if (logConf == LogConf.CONSOLE) {
    // log to System.out
    } else if (logConf == LogConf.FILE) {
    // log to a File
    } else {
    // what to do?!?
    }
  }
}
```

Quindi, per risolvere ciò

- decidiamo di astrarre la funzionalità di logging attraverso l'interafccia Log che avrà diverse implementazioni come ConsoleLog o FileLog;
- la classe Client dichiara una variabile privata di tipo Log;
- l'istanza Log viene passata al costruttore;
- Client delega completamente il logging all'istanza di Log.

La nuova classe è estendibile a nuovi sistemi di logging, infatti basta creare una nuova implementazione di Log, il Client non è stato minimamente toccato, non c'è stato bisogno di ricompilare.

```
public class Client {
  private Log logger;

  public Client(Log log) {
    this.logger = log;
  }

  public void aMethod() {
    // do something...
    log("a message");
    // do something...
  }

  private void log(String message) {
    logger.log(message);
  }
}
```

```
public interface Log {
  public void log(String message)
}

public class ConsoleLog implements Log {...}

public class FileLog implements Log {...}
```

Ovviamente le due classi concrete fanno override del metodo log.
Esempio pratico sono i software basati sui plug-in, ovveri software che non è possibile modificare ma estendibili tramite plug-in.

4.3 Liskow Substitution Principle (LSP)

Questo principio si basa sul concetto di sottotipo.
In java, un oggetto di tipo T sottotitpo di S, detto supertipo, può essere sempre

- assegnato a una variabile dichiarata di tipo S;
- passato come argomento di un metodo che si aspetta un parametro di tipo S;
- restituito in un metodo il cui tipo di ritorno è S.

Per un sottotitpo valgono la proprietà riflessiva, ovvero data una classe A, A è sottotipo di se stessa (si indica con $A <: A$) e la proprietà transitiva, ovvero siano A, B e C tre classi, se $A <: B$ e $B <: C \Rightarrow A <: C$.
Il prinicipio di Liskow è qualcosa di più forte della definizione di sottotipo, ovvero, vuole che il comportamento del programma, quando sostituisco S con T, non cambia, non si rompe, continua a funzionare correttamente (non significa che non cambia nulla, altrimenti non avrebbe senso).
Per esempio, ho una classe Rectangle con due campi, height e weight, metodi setter ed un metodo area.

```
public class Rectangle {
  private int height, width;

  public void setHeight(int height) {...}

  public void setWidth(int width) {...}

  public int area() { return height * width;}
}
```

```
@Test
public void testRectangle() {
  Rectangle r = new Rectangle();
  r.setHeight(5);
  r.setWidth(3);
  assertEquals(15, r.area());
}
```

Ora voglio creare la classe Square da rettangolo, tanto basta porre entrambi i lati uguali.

```
public class Square extends Rectangle {

@Override
  public void setHeight(int height) {
    super.setHeight(height);
    super.setWidth(height);
  }

  @Override
  public void setWidth(int width) {
    super.setWidth(width);
    super.setHeight(width);
  }
}
```

```
@Test
public void testSquare() {
  Square s = new Square();
  s.setHeight(5);
  assertEquals(25, s.area());
  s.setWidth(3);
  assertEquals(9, s.area());
}
```

Presi singolarmente, i test passano, ora, però, bisogna vedere che, se applicando il principio di sostituzione, tutto rimane invariato.

```
@Test
public void testSquareAsRectangle() {
  Rectangle r = new Square();
  r.setHeight(5);
  r.setWidth(3);
  assertEquals(15, r.area());
}
```

Questo test fallisce perchè, quando chiamo in sequenza i due metodi setter, questi saranno chiamati da quadrato e non da rettangolo (binding dinamico).
Quindi Square non è sostituibile a Rectangle anche perchè alla classe Square, alla fine dei conti serve un solo campo.
Per risolvere il problema possiamo pensare di eliminare i due metodi setter ed introdurre un'interafccia, Shape, con all'interno la definizione del metodo area.
Così facendo, entrambe le classi concrete, implementeranno Shape e ridefineranno, a modo loro, il metodo area attraverso override.

```
public interface Shape {
  public int area();
}
```

```
public class Rectangle implements Shape {
  private int height, width;

  public Rectangle(int height, int width) {
    this.height = height;
    this.width = width;
  }

  @Override
  public int area() { return height * width;}
}
```

```
public class Square implements Shape {
  private int side;

  public Square(int side) {
    this.side = side;
  }

  @Override
  public int area() { return side * side;}
}
```

I test che useranno Shap, passandogli prima Rectangle e poi Square, funzioneranno

```
@Test
public void testShape() {
  Shape s = new Rectangle(5,3);
  assertEquals(15, s.area());
  s = new Square(5);
  assertEquals(25, s.area());
}
```

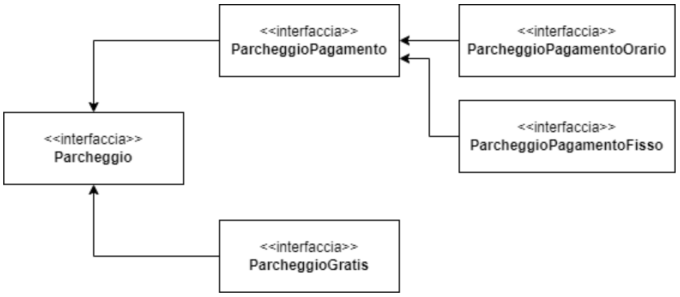
Detto in parole povere, bisogna cercare di lavorare, il più possibile, con classi astratte o interfacce, lavorare verso l'astrazione e non verso l'implementazione.

4.4 Iterafce Segregation Principle (ISP)

È simile all'SRP ma incentrato su interfacce e client.
Supponiamo di avere a disposizione un'interfaccia contente metodi che svolgono differenti compiti.
Un client che implementerà questa interfaccia, sarà costretto ad implementare tutti i suoi metodi, anche quelli che non gli servono.
Se un client richiedesse una modifica all'interfaccia, anche gli altri client ne sarebbero influenzati, anche se la modifica dovesse riguardare un metodo che a loro non serve.
Quindi sarebbe conveniente dividere l'interfaccia originale in interfacce più piccole, così facendo l'interfaccia originale conterrà solo i metodi comuni ai client, mentre le interfacce più piccole si occuperanno dei metodi specifici.
Prendiamo ad esempio il sistema di pagamento di un parcheggio

```
public interface Parcheggio {

  void parcheggiaAuto(); // Diminuisce parcheggi vuoti di 1
  void esceAuto(); // Aumenta parcheggi vuoti di 1
  void getCapienza(); // Restituisce capienza auto
  double calcolaQuota(Auto auto); // Restituisce il prezzo in base al numero di ore
  void faiPagamento(Auto auto);
}

class Auto {
  ...
}
```

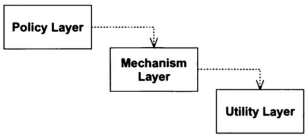
Immaginiamo di voler implementare un parcheggio gratuito (che estenderà Parcheggio) e notiamo che abbiamo obbligato questa classe ad implementare i metodi di pagamento del parcheggio anche se non ne avrebbe bisogno (è gratis) e magari, nel metodo pagamento, lanciamo un'eccezione gestita con un semplice "il parchegio è gratis".
Quindi si nota che l'interfaccia Parcheggio si occupa di due logiche distinte, quella del parcheggio e quella del pagamento del parcheggio stesso.
Una possibile soluzione, quindi, sarebbe quella di scindere l'interfaccia parcheggio in altre due interfacce, quella a pagamento e quella gratis, la prima oltre ad ereditare i metodi riferiti al parcheggio, aggiungerà i metodi riguardanti i pagamenti, mentre la seconda, implementrà l'interfaccia padre.
Con questo nuovo modello, possiamo anche andare oltre e dividere l'interfaccia pagamento per supportare diverse modalità di pagamento.



4.5 Dependency Inversion Principle (DIP)

Questo principio ci dice che i sistemi più flessibili sono quelli dove le dipendenze del codice sorgente si riferiscono solo alle astrazioni.
Per dipendenza nel codice sorgente si intende che, dato un file Client.java, se in dato file nomino un tipo A, allora diremo che il codice sorgente di Client dipende da A, anche se dovessimo usare un sottotipo di A, B, diremo che Client dipende, staticamente, da A.
Da qui utilizzeremo il concetto di modulo/componente inteso come raggruppamento di classi.
Tradizionalemente, i moduli di alto livello dipendono dai moduli di basso livello, per dipendere si intende chiamare direttamente codice di basso livello o istanziare direttamente classi concrete.
Per codice di alto livello, detto anche core, intendiamo la parte di un'applicazione computazionale, algoritmica o che elabora, ovvero è la parte che contraddistingue un'applicazione.
Mentre per codice di basso livello intendiamo interfaccia utente o database.
Quindi se il codice di alto livello dipende dal codice di basso livello, significa che modifiche a quest'ulitmo, avranno un impatto sul codice di alto livello.

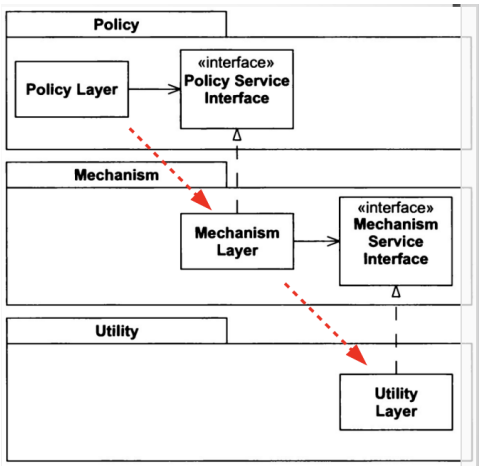
Si prende come esempio un'architettura OO dove la parte più alta indica codice di alto livello.



Il layer di alto livello usa solo il layer sottostante ma usare significa che dipende dal layer sottostante.
Inolte se il layer sottostante dipende dal layer di basso livello, allora il layer di alto livello, transitivamente, dipende dal layer di basso livello.
Per risolvere questo problema

- ogni layer di alto livello dovrebbe dichiarare una propria interfaccia per i servizi di cui ha bisogno e farla implementare al layer sottostante;
- il layer sottostante implementerà l'interfaccia dichiarata dal layer superiore.

Così facendo, le dipendenze vengono invertite, ovvero sono i layer di basso livello a dipendere dai layer di alto livello.



staticamente Policy chiama un metodo dell'interfaccia, a runtime verrà invocato l'implementazione di tale metodo dal layer inferiore, così facendo, posso cambiare un layer sottostante senza modificare un layer soprastante e posso testare un layer soprastante senza avere un layer sottostante ed inoltre è buona cosa non menzionare mai il nome di qualcosa che è concreto (non astratto) e volatile (che cambia spesso) e non avervi riferimento.
Quindi, i design pattern guidano lo sviluppo del codice e il refactoring aiutando a sviluppare software pulito e facile da estendere, riusare, mantenere e testare.

COVARIANZA E CONTROVARIANZA

Dalla matematica, una funzione *f*, tale che f: $D \to C$, può essere sosituita da una funzione *g*, tale che g: $D' \to C'$, sse $C' <: C$ (**covariante** sul tipo di **ritorno**) e $D <: D'$ (**controvariante** su tipo del **parametro**), ovvero una funzione g, che deve sostituire f, può essere più specifica sul tipo di ritorno e più generale sul parametro in input.
Supponiamo di avere due classi, Employee e Manager, tale che Manager<:Employee.
Employee ha un metodo setSalary() e Manager ha il metodo setBonus(int).
Prendiamo altre due classi, EmployyeDB e ManagerDB, tale che ManagerDB<:EmployyeDB.
La prima ha al suo interno un metodo find(int) che restiusce un Employee, mentre il secondo ridefinisce il metodo facendo tornare un Manager.

```
public class EmployeeDatabase {
    public Employee findEmployee(int id) {
        ...
    }
}
```

```
public class ManagerDatabase extends
    EmployeeDatabase {
    @Override
    public Manager findEmployee(int id) {
        ...
    }
}
```

Se eseguissi questo codice, tutto funzionerebbe.

```
EmployeeDB d = new ManagerDB();
Employee e = d.find(1);
```

Per il binding dinamico, d staticamente è EmployeeDB ma a runtime è ManagerDB, inoltre il metodo find resituisce un Manager e non abbiamo problemi anche perchè è un sottotipo di Employee.
Questo è consistente col tipo di ritorno che è covariante e che può essere uguale o più specializzato.
Ipotizziamo che il concetto di covarianza valga anche per i parametri e quindi supponiamo che EmployeeDB abbia un metodo updateEmployee(Employee), che setta il salario di un Employee, e che la classe sottotipo, ridefinisca il metodo cambiandone il parametro in un Manager, e aggiungendo al suo interno il metodo setBonus(int) (può farlo in quanto è di Manager).

```
public class EmployeeDB {
...
    public void updateEmployee(Employee e) {
        e.setSalary(...);
    }
}
```

```
public class ManagerDB extends EmployeeDB {
...
    @Override
    public void updateEmployee(Manager e) {
        super.updateEmployee(e);
        e.setBonus(0);
    }
}
```

Eseguendo questo codice

```
EmployeeDatabase d = new ManagerDatabase();
d.updateEmployee(new Employee());
```

per il binding dinamico, d staticamente è EmployeeDB ma a runtime è ManagerDB, quindi il metodo che sarà chiamato, sarà quello di ManagerDB che però non esiste e quindi lancerà un'eccezione del tipo NoSuchMethodException.
A livello di compilazione, tutto è ok perchè il metodo updateEmployee(Employee), staticamente è di tipo EmployeeDB ma a runtime chiamerà il metodo di ManagerDB dove abbiamo deciso di mettere in posizione covariante anche i parametri e questo non va bene perchè, per il binding dinamico, verrebbe chiamato updateEmployee di ManagerDatabase che peraltro si aspettava un Manager ed invece si ritrova un Employee e boom, eccezione del tipo NoSuchMethodException.
Java adotta un sistema di tipi **sound**, ovvero un programma, accettato staticamente dal compilatore, non generarà mai errori NoSuchMethodException a runtime.
Nello specifico i tipi statici, controllati durante la compilazione, durante l'esecuzione si mantengono come tali oppure diventano sottotipi.
Java non permette la controvarianza durante la ridefinizione di un metodo in quanto richiede che i tipi dei parametri non cambino affatto (si dice che java è invariante sul tipo dei parametri).

Supponiamo che Employee abbia un supertipo, Person e che EmployeeDB sia esteso da un'altra classe, PersonDB che esegue override del metodo chiamando dentro il suo metodo setName().

```
public class EmployeeDatabase {
    ...
    public void updateEmployee(Employee e) {
        e.setSalary(...);
    }
}

public class PersonDataBase extends EmployeeDatabase {
    @Override
    public void updateEmployee(Person e) {
        super.updateEmployee(e);
        e.setName("");
    }
}
```

Dal punto di vista della controvarianza, ciò è corretto, però avremo un errore, in quanto non potremmo usare il metodo super perchè richiede Employee, mentre gli stiamo passando un Person e un Person NON è un Employee. Quindi possiamo dire che Java, per la ridefinizione di metodi, adotta l'approcio sicuro, ovvero covariante sul tipo di ritorno e invariante sui parametri.
Magari una soluzione potrebbe essere quella di preferire l'overloading all'override, magari abbiamo lo stesso metodo, uno che prende in input Employee e l'altro che prende Person.

```
public class EmployeeDatabase {
    ...
    public void updateEmployee(Employee e) {
        e.setSalary(...);
    }
}

public class ManagerDatabase extends EmployeeDatabase {
    ...
    public void updateEmployee(Manager e) {
        super.updateEmployee(e);
        e.setBonus(0);
    }
}
```

Anche qui ci saranno problemi, infatti dato il seguente spezzone di codice

```
EmployeeDatabase d1 = new ManagerDatabase();
ManagerDatabase d2 = new ManagerDatabase();

// chiama EmployeeDatabase.updateEmployee(Employee)
d1.updateEmployee(new Employee());

// chiama ManagerDatabase.updateEmployee(Manager)
d2.updateEmployee(new Manager());

/*  chiama ANCORA EmployeeDatabase.updateEmployee(Employee) perche' d1 e' staticamente EmployeeDatabase
 *   ManagerDatabase.updateEmployee(Manager) aggiunge un metodo in overloading ma NON ridefinisce
 *   EmployeeDatabase.updateEmployee(Employee) quindi non si applica il binding dinamico!
*/
d1.updateEmployee(new Manager());
```

Quindi potremmo risolvere passando dall'usare l'overloading in una gerarchia di classi, all'usare l'overloading in una singola classe.

```
public class EmployeeDatabase {
    ...
    public void updateEmployee(Employee e) {
        e.setSalary(0);
    }

    public void updateEmployee(Manager e) {
        e.setBonus(0);
    }
```

```
}
```

però anche nel prossimo spezzone, avremo un comportamento non prevedibile

```
EmployeeDatabase d1 = new ManagerDatabase();

// chiama updateEmployee(Employee): l'argomento e' un Employee
d1.updateEmployee(new Employee());

// chiama updateEmployee(Manager): l'argomento e' un Manager
d1.updateEmployee(new Manager());

Employee e1 = new Employee();
Employee e2 = new Manager();

// chiama updateEmployee(Employee)
d1.updateEmployee(e1);

// chiama sempre updateEmployee(Employee)
d1.updateEmployee(e2);
```

perchè in java, l'overloading è un meccanismo statico.
Implementare una sorta di overloading dinamico è difficile dal punto di vista dell'efficienza (diverso dalla complessità costante del binding dinamico).
Quindi java, per la ridefinizione dei metodi, adotta l'approccio sicuro, covariante sul tipo di ritorno e invariante sui tipi dei parametri.
Per le lambda invece si, possiamo essere controvarianti sui parametri, esempio il metodo map degli stream che è definito come

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

ovvero, una lambda di tipo Function<T, R> può accettare anche una lambda di (sotto)tipo Function<T', R'> dove T <: T' e R' <: R

DESIGN PATTERN

Sono una soluzione generica e riusabile per un problema comune, ci dicono come risolvere un problema in un certo contesto software, fornendo chiare linee guida.
I pattern sono schemi, modelli che descrivono relazioni tra interfacce e classi e interazioni tra oggetti.
Sono pensati per risolvere specifici problemi di design, rendendolo flessibile, elegante e riusabile.
Ogni pattern è composto da 4 elementi, il *nome* per riferirsi al pattern, il *problema* che ci dice quando si applica e non (in alcuni casi ci sono delle condizioni da soddisfare), la *soluzione* che descrive gli elementi (classi, interfacce, oggetti) e le loro relazioni (responsabilità e collaborazioni) e le *conseguenze* che mostrano i risultati e i compromessi dall'applicazione del pattern.
Alcuni pattern sopperiscono alle mancanze di funzionalità/caratteristiche del linguaggio di programmazione (ad esempio il Visitor sopperisce alla mancanza dell'overloading dinamico).
Si differenziano per **purpose** (proposito) che riguardano le tipologie di pattern (creazionali, strutturali e comportamentali) e per **scope** (campo di azione) che riguardano le relazioni tra classi e sottoclassi (ereditarietà e overriding) e oggetti (object composition e delegation).
I class pattern trattano relazioni fra classi e sottoclassi, tali relazioni sono stabilite tramite inheritance (relazioni statiche e fisse a tempo di compilazione), mentre gli object pattern trattano relazioni fra oggetti che possono essere modificate a run-time (relazioni dinamiche).
Nei pattern creazionali i class patterns delegano a sottoclassi mentre gli object patterns delegano a un altro oggetto.
Nei pattern strutturali i class patterns usano l'inheritance per comporre classi mentre gli object patterns descrivono modi per assemblare oggetti.
Nei pattern comportamentali i class patterns usano l'inheritance per descrivere algoritmi e il "flow of control" mentre gli object patterns descrivono come un gruppo di oggetti cooperano per eseguire un certo task.

6.1 Notazione Uml

Unified Modeling Language, una notazione formale per la specifica, costruzione, visualizzazione e documentazione del modello di un sistema software.
Utile sia per documentazione che nella fase di sviluppo.

6.2 Interaction diagram

Mostra l'ordine in cui le richieste fra oggetti vengono eseguite.

TEMPLATE E STRATEGY

Sono due pattern comportamentali, descrivono come classi e oggetti interagiscono e si distribuiscono le responsabilità.

7.1 Template Method

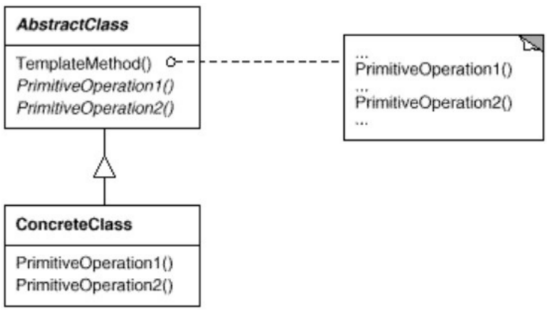
Definisce, in unico punto, lo scheletro di un algoritmo e delega alcuni passi, quelli che possono variare, alle sottoclassi che avranno il compito di ridefinirli senza modificare la struttura dell'algoritmo.
Prendiamo come esempio un framework che fornisce la classe Application e Document responsabili, rspettivamente, di aprire un documento e salvarlo e di rappresentare le informazioni una volta che il documento è in memoria.
Application, intesa come classe astratta, definirà l'algoritmo per aprire e leggere un documento in un metodo openDocument() che chiamerà al suo interno altri metodi di utilità, astratti, che saranno implementati dalle sottoclassi.
Il metodo openDocument() è il nostro templateMethod.

```
public abstract class Application {
    private List<Document> docs;
    // costruttore...

    public final void openDocument(String path) {
        if (!canOpenDocument(path)) {
            // gestisce l'errore
        }
        Document doc = doCreateDocument(path);
        docs.add(doc);
        aboutToOpenDoc(doc);
        doc.open();
        doc.doRead();
    }

    protected abstract void aboutToOpenDoc(Document doc);
    protected abstract Document doCreateDocument(String path);
    protected abstract boolean canOpenDocument(String path);
}
```

e chi intenderà utilizzare il nostro framework dovrà estendere le classi Application e Document, ridefinendo i metodi astratti.
Il templateMethod porta all'*hollywood principle*, ovvero sono le classi parent a chiamare i metodi delle sottoclassi e non l'inverso.

N.B. Sapendo che il templateMethod non cambiarà mai, allora potremmo renderlo final.

7.1.1 Struttura e partecipanti



AbstractClass

- *definisce* le operazioni astratte che rappresentano i passi di un algoritmo;
- *Implementa* il template method (l'algoritmo) che, a sua volta, chiama le operazioni astratte.

ConcreteClass che implementa le operazioni astratte.

7.2 Strategy

Definisce una famiglia di algoritmi, incapsula ogni algoritmo, e li rende intercambiabili.
Lo Strategy nasconde le informazioni interne dell'implementazione di un algoritmo, una classe può avere diversi comportamenti a seconda di qualche condizione.
Prendiamo ad esempio un Shop che può applicare più sconti, ad esempio quello percentuale, assoluto o nessuno sconto.
Prima di applicare lo Strategy saremo in una condizione di questo tipo

```
public class Shop {
    private DiscountType discountType;
        public Shop(DiscountType discountType) {
        this.discountType = discountType;
    }

    public void setDiscountType(DiscountType discountType) {
        this.discountType = discountType;
    }

    public int getTotal(int originalPrice) {
        switch (discountType) {
        case NO_DISCOUNT:
            return originalPrice;
        case ABSOLUTE_DISCOUNT:
            // sottrae un valore predefinito
            // (predefinito dove?)
            return ...;
        case PERCENTAGE_DISCOUNT:
            // applica uno sconto su percentuale
            // (quale percentuale?)
            return ...;
        }
        return ...;
    }
}
```

dove il tipo di sconto può essere modificato a runtime (good) ma i dettagli interni di ogni tipo di sconto devono essere specificati nello Shop (no good) e, se in futuro dovessi aggiungere un nuovo tipo di sconto, dovrei mettere alla classe Shop aggiungengo un nuovo ramo all'if-else, anche se non dovrebbe essere di sua compentenza.
Con lo Strategy avremo

```
public class Shop {

private DiscountStrategy discountStrategy;
    public Shop(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy
    }

    public int getTotal(int originalPrice) {
        return discountStrategy.applyDiscount(originalPrice);
    }
}
```

dove il tipo di sconto può essere modificato a runtime (good) e dettagli interni di ogni tipo di sconto non riguardano Shop (good).
DiscountStrategy è la nostra interfaccia che mette a disposizione il metodo, astratto, applyDiscount(int) che sarà gestito in maniera differente in base alla tipologia di sconto che vogliamo applicare, ad esempio

```java
public interface DiscountStrategy {
    int applyDiscount(int originalPrice);
}
```

```java
public class NoDiscountStrategy implements
     DiscountStrategy {
    @Override
    public int applyDiscount(int originalPrice) {
        return originalPrice;
    }
}
```

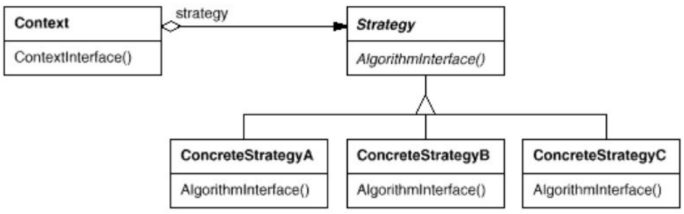
```java
public class AbsoluteDiscountStrategy implements
     DiscountStrategy {
private int discount;

public AbsoluteDiscountStrategy(int discount) {
    this.discount = discount;
}

@Override
public int applyDiscount(int originalPrice) {
    return originalPrice - discount;
}
}
```

```java
public class PercentageDiscountStrategy
     implements DiscountStrategy {
private int percentage;

public PercentageDiscountStrategy(int
     percentage) {
    this.percentage = percentage;
}

@Override
public int applyDiscount(int originalPrice) {
    return originalPrice - (originalPrice *
         percentage / 100);
}
}
```

7.2.1 Conseguenza

I Client devono essere consapevoli che esistono diverse strategie (cioè diverse implementazioni di una strategia).

7.2.2 Struttura e partecipanti



Strategy dichiara l'interfaccia comune a tutti gli algoritmi supportati;
ConcreteStrategy implementa l'interfaccia Strategy e l'algoritmo;
Context usa uno Strategy quando ha bisogno dell'algoritmo che, a runtime, sarà effettivamente implementato da un ConcreteStrategy. Context ha un riferimento a Strategy che sarà configurato assegnandogli un ConcreteStrategy passandolo al costruttore o tramite un setter, inoltre può definire a sua volta un'interfaccia per permettere a uno Strategy di accedere al suo stato.

7.2.3 Differenza con il templateMethod

Sembra simile al templateMethod, invece è completamente l'opposto, col templateMethod l'*implementazione dell'algoritmo è fissa*, cambia solamente l'implementazione di alcuni suoi passi, mentre nello strategy è *fisso il problema che intende risovere* l'algoritmo ma il come deve essere risolto viene implementato dalle sottoclassi.
Prendiamo come esempio l'ordinamento

- con il templateMethod avremo il Bubblesort come algoritmo fisso, dove al suo interno potranno **cambiare alcuni suoi passi**, quindi le sottoclassi estenderanno la classe Bubblesort ed implementeranno i suoi passi astratti;
- con lo Strartegy, si definisce un'interfaccia per l'ordinamento dove l'**algoritmo cambia**, a seconda del tipo di ordinamento, (SelectionSort, InsertionSort, Bubblesort, ecc. . . ..), quindi le sottoclasse implementeranno l'ordinamento desiderato e ridefiniranno l'algoritmo a proprio piacimento.

Template Method sfrutta l'inheritance e l'overriding (meccanismo statico), mentre lo Strategy sfrutta l'object composition e delegation (meccanismo dinamico), infatti l'implementazione di strategy può essere anche modificata a runtime, ad esempio tramite il setter.

7.2.4 Strategy e classi anonime

Si può creare una classe di utilità che crea le strategy con classi anonime

```
public class DiscountStrategies {

    //costruttore privato

    public static DiscountStrategy noDiscount() {
        return new DiscountStrategy() {
            @Override
            public int applyDiscount(int originalPrice) {
                return originalPrice;
            }
        };
    }

    public static DiscountStrategy absoluteDiscount(int discount) {
        return new DiscountStrategy() {
            @Override
            public int applyDiscount(int originalPrice) {
                return originalPrice - discount;
            }
        };
    }

    public static DiscountStrategy percentageDiscount(int percentage) {
        return new DiscountStrategy() {
            @Override
            public int applyDiscount(int originalPrice) {
                return originalPrice - (originalPrice * percentage / 100);
            }
        };
    }
}
```

7.2.5 Strategy e lambda

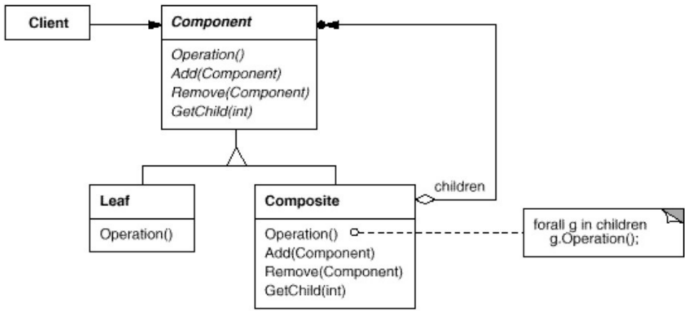
L'interfaccia Strategy di solito ha un singolo metodo astratto, quindi siamo in contesto lambda.

```
public class DiscountStrategies {
    public static DiscountStrategy noDiscount() {
        return originalPrice -> originalPrice;
    }

    public static DiscountStrategy absoluteDiscount(int discount) {
        return originalPrice -> originalPrice - discount;
    }

    public static DiscountStrategy percentageDiscount(int percentage) {
        return originalPrice -> originalPrice - (originalPrice * percentage / 100);
    }
}
```

COMPOSITE

Pattern strutturale (descrive la composizione di classi e oggetti), compone oggetti in strutture ad albero, adatto per tipi di dato ricorsivi.
Le classi client trattano questi oggetti in modo uniforme, indipendentemente dal fatto che si tratti di oggetti semplici o composti, inoltre non sanno, e non dovrebbero sapere, se stanno interagendo con foglie o composti.
Sfrutta sia il meccanismo dell'ereditarietà e sia il meccanismo dell'object composition e delegation, il primo attraverso il fatto che un Composite è un Component, il secondo dove Composite è composto da tanti Component e delega operation() ai figli.
È facile aggiungere nuovi tipi di componenti, basta creare nuove sottoclassi di Leaf o di Composite, questi potranno essere usati immediatamente in strutture esistenti ma è difficile imporre dei vincoli sul tipo dei componenti.
Infatti se un composto deve avere solo un certo tipo di figli, probabilmente il pattern non fa al caso nostro, questo pattern non può imporre vincoli statici, in quanto non possono essere controllati dal compilatore (type system) e quindi dovremmo effettuare controlli a runtime ed eventualmente, sollevare eccezioni (no good).
Un esempio di applicazione del pattern è il FileSystem, avremo la classe astratta FileSystemResource (il nostro Component), la sottoclasse FileSystemFile (Leaf) e FileSystemDirectory (Composite), che contiene al suo interno tanti oggetti di tipo FileSystemResource.

8.1 Struttura



Component

- dichiara l'interfaccia comune a tutti gli oggetti della composizione;
- implementa il comportamento di default comune a tutte le sottoclassi, in questo caso operation();
- dichiara un'interfaccia per accedere a/e gestire i componenti figlio (add(), remove() e get());
- definisce un modo per accedere al componente padre (opzionale).

Leaf

- rappresenta oggetti foglia della composizione;
- non ha figli;
- definisce il comportamento di default dell'interfaccia comune per gli oggetti primitivi della composizione.

Composite

- definisce il comportamento dell'interfaccia comune dei componenti con figli.
- memorizza i componenti figli;
- implementa le operazioni relativi alla gestione dei figli definite nell'interfaccia Component.

Client manipola gli oggetti della composizione attraverso l'interfaccia Component.

Se la richiesta viene inviata ad una foglia, allora viene gestita direttamente, oppure se viene inoltrata ad un oggetto composto, allora viene inoltrata ai figli, facendo, eventualmente, delle operazioni prima e/o dopo l'inoltro.

N.B. Le operazioni di gestione dei figli sono in Component, quindi Leaf li dovrebbe overridare sollevando un'eccezione oppure non facendo nulla.

Per questo motivo abbiamo due versioni di questo pattern, una è quella che abbiamo visto ora, chiamata **design for uniformity**, e l'altra chiamata **design for tape safety** dove i metodi di gestione dei figli sono in Composite.
Nella prima i client, non sapendo se stanno trattando Leaf o Composite, potrebbero chiamare i metodi di gestione dei figli direttamente su Leaf, rischiando di ottenere errori a runtime.
Con l'altra versione il client non può chiamare operazioni per la gestione dei figli su Leaf, evitando così errori a runtime, può farlo solo Composite perdendo però uniformità.
Inolre il client dovrebbe avere un riferimento, di tipo statico, alla radice della struttura Composite, in caso contrario non potrebbe chiamare i metodi per gestire i figli.

ITERATOR

Pattern comportamentale (descrive come classi e oggetti interagiscono e si distribuiscono le responsabilità) usato per accedere ai contenuti di un oggetto aggregato senza esporne la rappresentazione interna e fornisce un modo uniforme per attraversare tale collezione, indipendentemente dal tipo di collezione o dalla sua struttura sottostante.
Supponiamo di avere una classe scuola, avente una collezione privata di studenti permettendo ai client di poter aggiungere alla collezione col metodo add().
Supponiamo inoltre di volerne dare un accesso in sola lettura, attraverso un getter pubblico

```
public class School {
    private ArrayList<Student> students = new ArrayList<>();

    public void addStudent(Student student) {
        students.add(student);
    }

    public ArrayList<Student> getStudents() {
        return students;
    }
}
```

Così facendo si espone ai client un dettaglio implementativo, ArrayList e, se un giorno decidessi di cambiare l'implementazione della collezione, i client avranno problemi di compilazione.
Un getter protected non migliorerebbe la situazione in quanto ai client esterni basterebbe estendere la nostra classe per usare quel getter e quindi stesso problema.
Utilizzando un iteratore

- accediamo ai contenuti di oggetto aggregato senza esporne la rappresentazione interna;
- possiamo supportare diverse strategie di attraversamento degli oggetti aggregati;
- fornire un'interfaccia uniforme per attraversare strutture aggregate diverse.

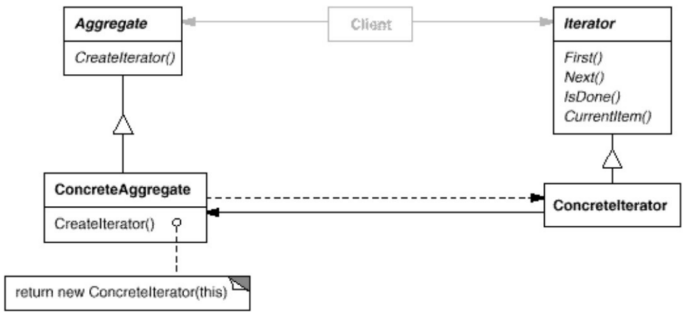
N.B. Un iteratore non conosce il numero di elementi di un aggregato, può solamente attraversarli.
Quindi con l'iteratore avremo

```
public class School {
    private List<Student> students = new ArrayList<>();

    public void addStudent(Student student) {
        students.add(student);
    }

    public Iterator<Student> studentIterator() {
        return students.iterator();
    }
}
```

dove i client usano studentIterator() per "attraversare" sequenzialmente i vari studenti della scuola, ignorando completamente la rappresentazione interna della struttura che li contiene senza poter fare assunzioni sull'ordine durante l'attraversamento.

N.B. Una volta usato un iteratore per un attraversamento completo, ne va usato uno nuovo.

9.1 Struttura



Iterator definisce l'interfaccia per attraversare gli elementi e offre metodi come hasNext() per verificare se ci sono altri elementi da attraversare e next() per ottenere l'elemento successivo;
Aggregate è l'interfaccia o la classe astratta che rappresenta la collezione di oggetti. Fornisce un metodo per ottenere un iteratore che può essere utilizzato per attraversare gli elementi.
ConcreteIterator implementa Iterator e tiene traccia, durante l'attraversamento, dell'oggetto corrente e deve essere in grado di calcolare l'elemento successivo. Per questo ha un riferimento all'aggregato concreto.
ConcreteAggregate implementa Aggregate e crea un opportuno ConcreteIterator.

9.2 Iterator e classi anonime

Un iteratore concreto è tipicamente implementato come classe (privata) interna o direttamente come classe anonima, infatti deve poter accedere direttamente alla rappresentazione interna (privata) dell'aggregato.

```
public class School {
    private Student[] students = new Student[100];
    private int index = 0;

    public void addStudent(Student student) {
        // controlli per la dimensione da fare...
        students[index++] = student;
    }

    public Iterator<Student> studentIterator() {
        return new Iterator<Student>() {
            private int current = 0;

            @Override
            public boolean hasNext() {
                return current < index;
            }

            @Override
            public Student next() {
                if (current >= index)
                    throw new NoSuchElementException();
                return students[current++];
            }
        };
    }
}
```

9.3 Iterator vs Stream

Possono sembrare simili.
Gli stream sono più astratti e possono essere parallelizzati, si basano su un approccio dichiarativo, permettono facilmente di filtrare, raggruppare, partizionare e gestiscono completamente in automatico l'attraversamento dell'aggregato.
Gli iteratori sono molto più semplici, permettono di attraversare un aggregato un elemento alla volta, sono per loro stessa natura sequenziali ma richiedono che hasNext() e next() siano usati correttamente insieme.

PATTERN CREAZIONALI

Questa tipologia di pattern *astrae* il processo di creazione degli oggetti, fornendo diversi modi per associare un'interfaccia (o classe astratta) con un'implementazione e assicura che il programma sia scritto in termini di interfacce e non di implementazione.

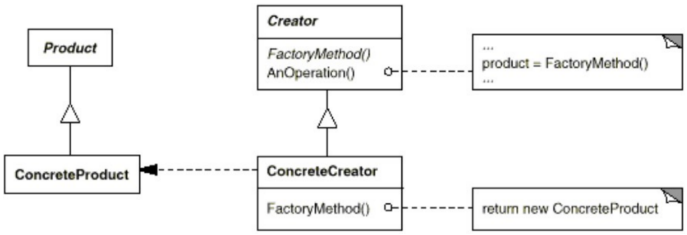
10.1 PERCHE'?

Dal DIP sappiamo che è fondamentale non riferirsi a classi concrete (ma a classi astratte) e questo implica che ogni istruzione new mette a rischio questo principio.
Però, prima o poi, lo statement new deve essere usato ed è fondamentale che questo avvenga in poche e limitate parti del programma su cui abbiamo il controllo di flessibilità.
Se volessimo cambiare il tipo degli oggetti creati, saremmo in grado di farlo in un solo punto e senza troppe conseguenze al resto del programma.

10.2 Factory Method

Definisce un'interfaccia per creare un oggetto di un certo tipo ma lascia decidere alle sottoclassi quale oggetto istanziare.
La creazione di un'instanza avviene in un metodo (Factory method) che può essere astratto per poi essere implementato dalle sottoclassi oppure concreto per poi essere ridefinito dalle sottoclassi.
L'assegnazione dell'instanza viene salvata in una variabile, sempre nella classe base, privata e chi implementa/estende l'interfaccia/classe astratta avrà il compito di implementare/ridefinire il factory method.
Nella classe base il metodo restituirà il tipo dell'interfaccia desiderata, mentre nelle sottoclassi possiamo sfruttare la covarianza.
Nell'esempio di Application e Document, visto nel Template Method, il metodo createDocument() è il nostro factory method, infatti risulta essere astratto.
Factory Method serve a *"parametrizzare"* rispetto alla creazione di un oggetto, basandosi solo su inheritance, ridefinizione di metodo e binding dinamico.
Se il linguaggio permette di "passare" un blocco di codice (lambda) che crea un oggetto, si può anche fare a meno di questo pattern.

10.2.1 Struttura



Product definisce l'interfaccia degli oggetti creati tramite il FactoryMethod di Creator.
Creator dichiara il FactoryMethod, che restituisce un Product, con eventualmente un'implementazione di default, e lo chiama.
ConcreteProduct implementa l'interfaccia Product.
ConcreteCreator implementa o ridefinisce il FactoryMethod di Creator ritornando un ConcreteProduct.

10.2.2 Factory Method vs Template Method

Il factory method è un metodo, eventualmente *astratto* e *protetto*, che verrà poi implementato o ridefinito dalle sottoclassi e viene *richiamato* da diversi metodi concreti della classe base.
Il template method è un metodo *concreto*, eventualmete *final*, che *richiama* operazioni implementate da sottoclassi.
Il primo è pensato per essere *chiamato solo dalla classe base* (che lo dichiara) mentre il secondo è pensato per essere *chiamato dai client*.

10.2.3 Principale utilizzo

Separare la creazione degli oggetti dal loro uso permette di modificare la loro creazione in modo consistente senza produrre una cascata di modifiche nel codice che li usa.
Molto usato nei framework, che generalmente esistono a livello astratto e non dovrebbero conoscere il tipo concreto degli oggetti da istanziare, questa responsabilità è lasciata al Client del framework, che provvederà a estendere il framework con sottoclassi per definire concretamente i factory methods.
Nell'esempio del Template Method, Application e Document sono la parte astratta del framework che si occupa di creare e gestire il framework, mentre la parte bassa è il Client che ridefinisce/implementa, a suo piacimento, i metodi.

10.2.4 Vantaggi e svantaggi

Con questo pattern si elimina la necessità di legare il codice di un'applicazione a classi concrete, il codice lavora con Product, quindi può lavorare con ogni specifico prodotto definito dall'utente.
In compenso, siamo obbligati a creare una sottoclasse di Creator per creare una certa sottoclasse di Product, un'alternativa è quella di usare le lambda (però non stiamo più parlando del pattern factory method).
Nel costruttore passiamo l'interfaccia funzionale Supplier ed usiamo il suo metodo astratto, get, nei metodi in cui assegnamo l'istanza dove invece di una lambda si può passare il constructor reference.

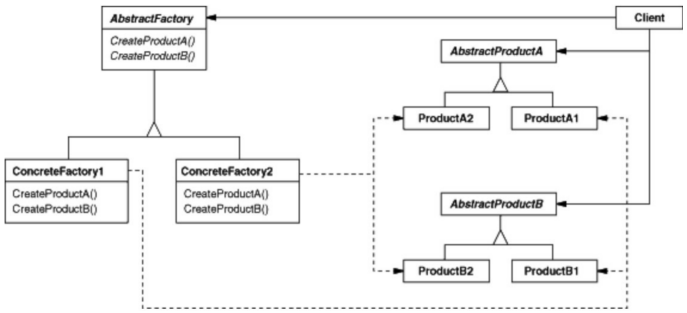
```
public class Client {
    private MyType f;
    private Supplier<MyType> myTypeSupplier;

    public Client(Supplier<MyType> myTypeSupplier) {
        this.myTypeSupplier = myTypeSupplier;
    }
    public void init() {
        f = myTypeSupplier.get();
    }
    public void m() {
        f.n();
    }
    public void reset() {
        f = myTypeSupplier.get();
    }
}
```

```
Client baseClient = new Client(MyImpl::new);
Client derivedClient = new Client(MyOtherImpl::new);
```

10.3 Abstract Factory

Fornisce un'interfaccia per creare famiglie di prodotti correlati o dipendenti senza specificare le loro classi concrete.
Il Client non crea gli oggetti direttamente ma delega la loro creazione a un altro oggetto.

10.3.1 Struttura



AbstractFactory dichiara un'interfaccia per le operazioni di creazione di famiglie di AbstractProduct.
AbstractProduct interfaccia di una singola famiglia di prodotti.
ConcreteFactory implementa le operazioni per creare prodotti della stessa famiglia di un AbstractProduct.
Product implementazione di AbstractProduct.
Client utilizza l'AbstractFactory per la creazione di AbstractProduct.

A run-time avremo una singola ConcreteFactory che crea, in modo consistente, una famiglia di prodotti.

Se volessimo creare un'altra tipologia di prodotti, allora dovremmo instanziare ed usare un'altro tipo di ConcreteFactory, ogni famiglia di prodotti ha una propria implementazione di abstract factory.
Se si volesse aggiungere un nuovo prodotto, bisognerebbe aggiungere un nuovo metodo all'abstractFactory che sarà poi implementato dalle ConcretFactory.

10.3.2 Principale utilizzo

Un sistema deve essere indipendente da come gli oggetti sono creati, composti e rappresentati.
Un sistema che deve essere configurabile con una di tante famiglie di prodotti dove, i prodotti correlati di una stessa famiglia, sono progettati per essere usati insieme.
Si vuole fornire una libreria di oggetti rivelando solo le loro interfacce e non le loro implementazioni.
In quest'ultimo caso, le classi concrete dei prodotti, possono essere rese inaccessibili ai client mettendole in un package private, accessibili solo al package della factory, cosicchè le factory sono le uniche a poter instanziare le classi concrete (un'altra variante è quella di usare classi anonime o interne ma avremo codice meno leggibile).

10.3.3 Accedere e modificare la factory

Per accedere alla factory ci sono due alternative, esiste una sola instanza della factory (tramite uso del Singleton) oppure viene passata ai client attraverso un costruttore o setter.
Cambiare la factory a run-time può essere pericoloso in quanto, dopo il cambiamento, bisognerebbe notificare tutti i client in modo tale da ricreare tutti gli oggetti che sono stati instanziati attraverso la factory precedente.
Se la factory è implementata tramite Singleton, allora cambio l'instanza, se è implementata tramite setter, passo la nuova instanze oppure se è passata tramite costruttore, allora si ricreano le instanze dei client.

10.3.4 AbstractFactory vs Factory Method

Il primo è pensato per essere *usato da altre classi* ed i suoi metodi sono *pubblici*.
Il secondo è pensato per essere *chiamato solo dalla classe base* (che lo dichiara) e *non* deve essere pubblico.

10.4 Static Factory Methods

Sono un'alternativa alla creazione di oggetti con new.
Dato un tipo MyType, invece di scrivere new MyType, scriveremo MyType.create dove il costruttore di MyType è reso privato e viene chiamato all'interno di un metodo statico di nome non necessariamente create.
A differenza dei costruttori, che sono vincolati ad avere lo stesso nome della classe e non ci possono essere più costruttori con la stessa firma, gli static factory method possono avere nomi più significativi.
Supponiamo di voler creare un oggetto TimeSpan specificando i secondi, minuti (entrambi interi).
Con i costruttori sarebbe più complicato perchè, anche cercando di usare l'overloading, avremo errore di compilazione in quanto avrebbero la stessa firma.
Quindi potremmo pensare di avere un unico costruttore, privato, che prende in input entrambi i parametri e creare quanti static factory method vogliamo, che ritornano l'oggetto TimeSpan che fanno due cose differenti, uno per i secondi ed uno per i minuti.

```
public class TimeSpan {
    private TimeSpan(int seconds, int minutes) {...}

    public static TimeSpan ofSeconds(int seconds) {
        return new TimeSpan(seconds, 0);
    }

    public static TimeSpan ofMinutes(int minutes) {
        return new TimeSpan(0, minutes);
    }
}
```

Se volessi creare un TimeSpan con le ore, basterebbe modificare il costruttore senza problemi, perchè privato, aggiungere le ore, creare un nuovo static factory method, chiamare al suo interno il nuovo costruttore e aggiornare gli static factory method esistenti.

N.B. Se il construttore fosse stato pubblico, allora, dopo l'aggiunta del terzo parametro, avremmo avuto problemi di compilazione sia sul nostro codice che su quello dei client esterni.

Quando si invoca un costruttore (tramite un new) viene sempre creato un nuovo oggetto, con gli static factory methods, invece, si possono restituire oggetti condivisi.

10.5 Singleton

Assicurare che una classe abbia una sola istanza e ne fornisce un punto di accesso globale.
Viene implementato tramite uno static factory method (per convenzione si chiama getInstance) che restituisce l'istanzia in un campo statico privato (eventualmente crea l'oggetto la prima volta che viene chiamato).

```
public class MyClass {
    private static MyClass instance = null;

    private MyClass() {...}

    public static MyClass getInstance() {
    if (instance == null) {
        instance = new MyClass();
    }
        return instance;
    }
}
```

10.6 Fluent Interface

Questo design pattern si basa sul Method Chaining, ovvero ogni metodo ritorna un oggetto, in modo da concatenare insieme tante invocazioni in un singolo statement, senza dover salvare i risultati intermedi in variabili locali.
Viene usato per fare il design delle API Object-Oriented, quindi tipico delle librerie e dei framework.
Un tipico caso d'uso è rimpiazzare i setter per rendere la configurazione di un oggetto più facile.

10.7 Builder

Separa la costruzione di un oggetto complesso dalla sua rappresentazione, la configurazione non viene fatta nel costruttore ma viene gestita da un altro oggetto, il builder (si basa su fluent interface).
La costruzione viene preparata passo-passo e infine viene effettivamente creata l'instanza.
Tipicamente la classe builder è una static inner class della classe di cui si vogliono costruire oggetti, il costruttore prende gli argomenti per i campi "richiesti" (non opzionali), fornisce metodi per specificare argomenti per campi opzionali, che non vengono applicati subito ai campi dell'oggetto da creare ed infine fornisce un metodo, build(), che crea effettivamente l'oggetto usando gli argomenti raccolti per i vari campi.
La classe di cui si vuole costruire l'oggetto ha il costruttore e setter privati, in questo modo solo la classe builder vi può accedere e diventa l'unico modo per istanziare oggetti della classe.
L'oggetto, una volta costruito, è sempre valido, eventuali problemi negli argomenti passati vengono rilevati durante la creazione del builder e non dopo.
Supponiamo di avere una classe Person che ha dei campi che deve avere obbligatoriamente ed altri opzionali.
Per applicare il pattern, il costruttore di Person deve essere privato.

```
public class Person {
    private int id; // richiesto
    private String name; // richiesto

    private Integer age; // opzionale con validazione
    private String address; // opzionale senza validazione

    // costruttore privato ed, eventualmente, getter pubblici
}
```

Aggiungiamo la classe builder, che è una static inner class che ha gli stessi parametri della classe ospitante.

```
public class Person {
    ...
    public static class PersonBuilder {
        private int id;
        private String name;
        private Integer age;
        private String address;

        public PersonBuilder(int id, String name) {
            this.id = id;
```

```
            this.name = name;
        }
        ...
    }
    ...
}
```

Aggiungiamo anche i metodi per i campi opzionali (ritorano sempre un builder) e, per convenzione, cominciano con with (non hanno nulla in comune con i withers che invece resituiscono una copia dell'oggetto con qualcosa di diverso).

```
public class Person {
    ...
    public static class PersonBuilder {

        //dichiarazione dei parametri

        //costruttore dei parametri obbligatori

        public PersonBuilder withAge(Integer age) {
            if (age <= 0)
                throw new IllegalArgumentException("age must be positive: " + age);
            this.age = age;
            return this;
        }
        public PersonBuilder withAddress(String address) {
            this.address = address;
            return this;
        }
    }
    ...
}
```

Infine il metodo build che costruisce l'oggetto e che ritorna il tipo della classe ospitante.

```
public class Person {
    ...
    public static class PersonBuilder {

        //dichiarazione dei parametri

        //costruttore dei parametri obbligatori

        //metodi per i parametri opzionali

        public Person build() {
            Person person = new Person();
            person.setId(id);
            person.setName(name);
            person.setAge(age);
            person.setAddress(address);
            return person;
        }
    }
    ...
}
```

Per creare un oggetto Person, bisognerà accerdere prima al costruttore del builder, chiamare i metodo opzionali ed infine usare il metodo build (sembra uno stream, infatti anche loro si basano su fluent interface).
Se il build va a termine, significa che l'oggetto creato è valido, in caso contrario avremo errori prima della sua effettiva creazione (vedi il metodo withAge).

```
Person person = new Person.PersonBuilder(1, "John")
                          .withAge(10)
                          .withAddress("an address")
                          .build();
```

OBSERVER

E' un design pattern comportamentale utilizzato per gestire le relazioni uno a molti quando lo stato di uno degli oggetti cambia e si richiede l'aggiornamento automatico di altri oggetti dipendenti.

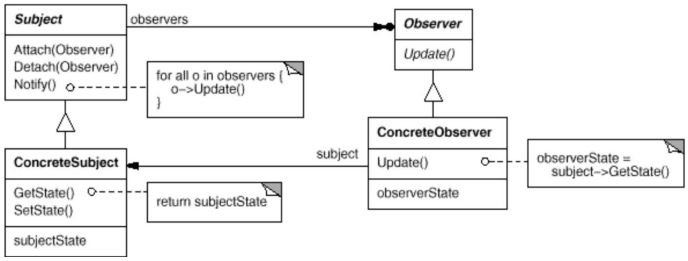
Classico esempio è quello di excel dove abbiamo una cella caratterizzata da una formula che fa riferimento a più celle "satellite" ed ogni volta che modifichiamo il valore di questi satelliti, la cella caratterizzata dalla formula cambia di conseguenza.

Questo pattern è basato sul concetto di un oggetto (**subject**) che tiene traccia dei suoi osservatori (**observers**).
Tutti gli observer vengono notificati ogni volta che il subject cambia il suo stato (è il subject che notifica e non sono gli osservatori che controllano ogni tot tempo).
A quel punto ogni observer può ispezionare lo stato del subject e aggiornare il proprio stato.

N.B. Il subject non conosce quali observer lo osservano, lui si mantiene una lista di observer ma non conosce la loro implementazione concreta (**accoppiamento astratto** o abstract coupling).
Quando notifca, lui non sa a chi sta veramente notificando.

Questo pattern favorisce la riduzione della dipendenza tra il soggetto e gli osservatori, permettendo di aggiungere o rimuovere osservatori senza modificare il soggetto stesso.
Inoltre, favorisce un design flessibile e scalabile, permettendo di gestire in modo efficiente le relazioni tra gli oggetti in un sistema software.

11.1 Struttura



Subject fornisce un'interfaccia per aggiungere, rimuovere e notificare gli Observer, ne conosce solamente l'interfaccia e può essere osservato da qualsiasi Observer.
Observer definisce l'interfaccia per gli oggetti che devono essere notificati riguardo il cambiamento del Subject.
ConcreteSubject mantiene uno stato da fornire ai ConcreteObserver.
ConcreteObserver ha un riferimento a ConcreteSubject ed un proprio stato che deve essere consistente al ConcreteSubject.

11.2 Interazione

ConcreteSubject notifica i suoi osservatori quando avviene un cambiamento che potrebbe rendere lo stato degli Observer inconsistente col proprio.
Durante la notifica il ConcreteObserver può ispezionare il Subject per ottenere le informazioni per aggiornare il proprio stato, in modo da renderlo consistente con quello del Subject.

11.3 Conseguenze

Observer non si riferisce a Subject ma è Subject che si riferisce agli Observer notificandoli.
ConcreteObserver ha un riferimento diretto a ConcreteSubject, mentre ConcreteSubject non lo ha, lo fa indirettamente chiamando il metodo di notifica ereditato da Subject.
Il Subject invia una notifica, in modalità broadcast a tutti i suoi Observer e spetterà all'Observer capire se la notifica è di suo interesse oppure no.

11.4 Notifica

Se la notifica parte dal Subject, allora i client non sono responsabili della notifica ma se il client cambia in sequenza più campi dello stesso Subject, allora avremo numerose notifiche consecutive (inefficienza).
Se la notifica viene chiamata dal client, ad esempio dopo vari cambiamenti chiama la notifica, è lui il responsabile della chiamata ma almeno si evitano le numerose notifche continue (sperando che il client si ricordi di chiamare la notifca).

11.5 Svantaggio

Il pattern è sincrono, ovvero, durante la notifica, aspetta che tutti i suoi Observer aggiornino il proprio stato.
Uno di questi Observer potrebbe essere maldestro, nel senso che ci mette troppo tempo, bloccando tutti gli altri, oppure maligno, magari lanciando un'eccezione.

PUBLISH SUBSCRIBER

Pattern basato su messaggi, abbiamo un Publisher che invia il messaggio ed un Subscriber che si sottoscrve ad un certo tipo di messaggio.

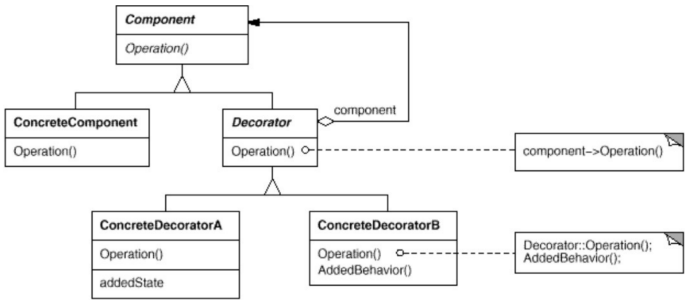
12.1 Publish subscribe vs Observer

I Publisher non inviano direttamente il messaggio ai Subscriber (mentre nel Observer è prorpio così) ma sono affidati ad un intermediario (**broker**) che si occupa di consegnare il messaggio al Subscriber appropriato (nell'Observer sono loro a decidere se è di loro interesse o meno).
Quindi Publisher e Subscriber sono completamente scorrelati (mentre nel Observer abbiamo una specie di accoppiamento astratto).
Questo pattern è asincrono (mentre l'altro è l'opposto), ovvero una volta che il Publisher consegna il messaggio al broker, lui continua a svolgere i propri compiti ed i mesaggi saranno consegnati, non subito, ai Subscriber appropriati (thread paralleli) ed è alla base dei sistemi ad evento (java swing) dove il messaggio è l'evento e i subscribe sono chiamati listener che si registrano e rimangono in ascolto di un determinato evento.
Questo pattern permette di avere un filtraggio dei messaggi, i subscriber dichiarano interesse per un certo tipo di messaggio e il broker si occuperà di consegnare ai subscriber solo i messaggi appropriati, invece, in Observer, spetta agli osservatori occuparsi manualmente del filtraggio.

DECORATOR

Pattern strutturale, ovvero descrive come classi e oggetti vengono composti per formare strutture più ampie, è basato su object composition e delegation e il suo intento è quello di aggiungere, dinamicamente, responsabilità ad un oggetto senza influenzare altri oggetti.
Fornisce una soluzione più flessibile riguardo l'utilizzo dell'ereditarietà con la quale si avrebbe un'esplosione di sottoclassi perchè, per cambiare una decorazione, dovremmo creare una nuova instanza di un'altra classe.
L'idea di base è quella di racchiudere l'oggetto (**decorato**) in un altro oggetto (**decoratore**) che delega i metodi all'oggetto decorato aggiungendo del comportamento prima o dopo la inoltro.
Il decoratore ha la stessa interfaccia dell'oggetto decorato, quindi la sua presenza è trasparente per i client e ciò permette di avere tanti decoratori annidati ricorsivamente.

13.1 Struttura



Component definisce l'interfaccia per gli oggetti a cui è possibile aggiungere, dinamicamente, nuove responsabilità;
ConcreteComponent implementa Component;
Decorator ha un riferimento a un oggetto Component e definisce un'interfaccia conforme a Component;
ConcreteDecorator aggiunge responsabilità all'oggetto Component.

Il Decorator inoltra, di default, le richieste al suo oggetto Component e i ConcreteDecorator aggiungono il comportamento prima o dopo l'inoltro.

N.B.L'inoltro ci deve sempre essere, altrimenti rompiamo la catena di decorazione.

Per l'interface conformance, decorator e ConcreteComponent devono avere una superclasse comune e tale interfaccia, Component, deve essere il più leggera possibile, deve concentrarsi solo sul definire un'interfaccia e non uno stato in quanto mettere troppe funzionalità in Component porterà, le sottoclassi, ad avere funzionalità che non gli competono.

13.2 Conseguenza

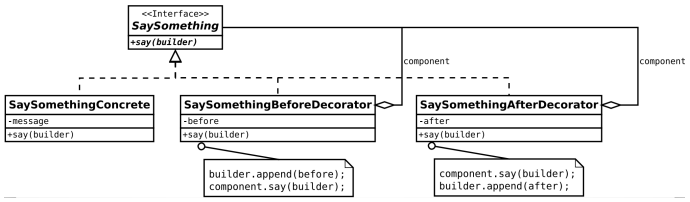
Un sistema basato su questo pattern avrà a che fare con tanti piccoli oggetti che differiscono per il modo in cui sono iterconnessi e composti.
Questi sistemi sono flessibili e facili da customizzare ma sono difficli da comprendere e debuggare.
Se esiste un solo tipo di decorazione, la classe astratta Decorator non serve.
Uno stesso oggetto Decorator non può essere usato per decorare due componenti.
Potremmo cambiare il componente da decorare se aggiungiamo un setter nel decorator ma lo stesso oggetto decoratore può decorare un solo componente alla volta.

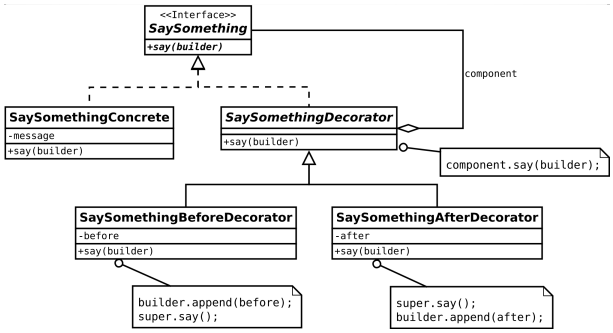
13.3 Decorator vs Strategy

Il Decorator rispetta il principio *"Changing the skin of an object versus changing its guts"* (cambiare la pelle invece delle viscere), cambia la pelle di un oggetto avvolgendolo in un decoratore, invece lo Strategy cambia le viscere, ovvero l'oggetto delega allo Strategy il comportamento da seguire.
Component non è a conoscenza dei suoi decoratori, il client utilizza, *senza saperlo*, il Decorator più esterno che farà qualcosa prima o dopo l'inoltro al Component che, a sua volta, se sarà un altro decoratore farà la stessa cosa, altrimenti applicherà il comportamento di default.
Nello Strategy, il client *è consapevole* delle diverse implementazioni (ha un riferimento allo Strategy).
Il Decorator ha la stessa interfaccia del suo Component mentre lo Strategy ha la sua interfaccia con diverse implementazioni concrete.

13.4 Implementazione senza e con l'entità AbstractDecorator

Sono due versione del pattern Decorator, nella prima avremo l'interfaccia SaySomething implementata direttamente sia da SaySomethingConcrete che da SaySomethingBeforeConcreteDecorator e SaySomethingAfterConcreteDecorator dove i due decoratori avranno un riferimento a SaySomething e chiameranno direttamente l'inoltro su quest'ultimo, aggiungendo del comportamento prima o dopo l'effettivo inoltro.
Nella seconda versione, SaySomething verrà implementata da SaySomethingConcrete e da SaySomethingDecorator che verrà estesa dai decoratori visti prima.
A differenza della prima versione, sarà SaySomethingDecorator ad avere un riferimento a SaySomething e ci chiamerà l'inoltro, mentre i due decoratori si occuperanno di aggiungere del comportamento prima o dopo l'inoltro, inoltre, non avendo un riferimento diretto a SaySomething, non potranno chiamare l'inoltro su SaySomething, ma lo delegheranno a SaySomethingDecorator attraverso il super.





Aggiungere nuovi tipi di decorator è facile, basta implementare/estendere e poi avvolgere SaySomething.
Il difficile è rimuovere un decorator in quanto la catena delle decorazioni deve essere ricreta, dove l'oggetto SaySomething finale non deve essere re-istanziato, mentre i decoratori intermedi si.

N.B. Quando passiamo il componente da decorare in un decorator, il componente non deve essere null (deve essere documentato nel javadoc), infatti occorre fare un controllo la prima volta che ci viene fornito un componente da decorare null (bisognerà anche testare questo scenario).

13.5 L'inoltro

In questo pattern potrebbe capitare di dimenticarsi di inoltrare al Component (non deve succedere).
Per evitare ciò, potremmo rendere il metodo di inoltro, nella classe Decorator, final, e di definire due metodi, astratti, che aggiungono del cambiamento prima e dopo l'inoltro.
In questo modo le sottoclassi non potranno dimenticarsi di inoltrare al componente, anche perchè non possono ridefinirlo essendo final, ma avranno il compito di implementare i metodi astratti definiti in Decorator.
Il fatto rendere il metodo final e lasciare alcuni suoi passi astratti è un esempio di template method, molti pattern sono usati insieme.

13.6 Lambda

Il pattern Decorator sfrutta il meccanismo dell'ereditarietà, object composition e delegation per sopperire alla mancanza della composizione di funzioni.
Se il linguaggio supporta le lambda, allora, invece di usare i tre meccanismi, possiamo usarle per avere lo stesso risultato ma, a quel punto, non parliamo più di Decorator.
Le lambda si usano con le interfacce funzionali (un solo metodo), quindi è applicabile solo al Component che ha uno ed un solo metodo da decorare.
Quindi, nell'esempio visto prima, SaySomething è un'interfaccia funzionale, quindi eleggibile all'utilizzo delle lambda.
Una stessa lambda può essere usata più volte all'interno di una composizione (invece con i decorator dovevamo creare obbligatoriamente due instanze per la stessa funzione), inoltre la composizione di una lambda è *lineare* (fluent interface), mentre la composizione del Decorator è *annidata* (la linearità è più leggibile).

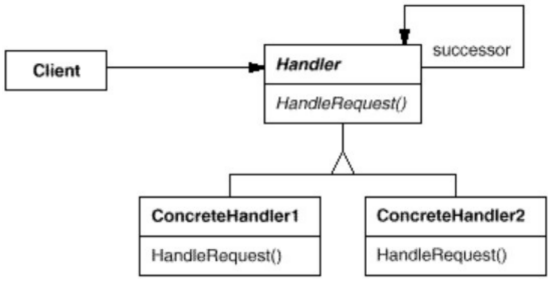
CHAIN OF RESPONSABILITY

Pattern comportamentale avente lo scopo di evitare di associare il mittente di una richiesta al suo destinatario, dando a più di un oggetto la possibilità di gestire la richiesta.
Il primo oggetto della catena che riceve la richiesta, la gestisce oppure la inoltra al prossimo candidato che si comporterà nello stesso modo.
Il client che effettuerà la richiesta, non dovrà preoccuparsi di chi dovrà gestirla, anche perchè non sa chi la gestirà, quindi è fondamentale che gli oggetti della catena abbiamo la stessa interfaccia.

14.1 Applicabilità

Quando più oggetti possono gestire una richiesta ma non è noto a priori chi lo farà oppure il caso in cui l'insieme di oggetti che possono gestire una richiesta devono essere specificati dinamicamente.

14.2 Struttura



Handler definisce l'interfaccia per gestire le richieste ed eventualmente implementa il riferimento al successore nella catena;
ConcreteHandler gestisce le richieste per cui è responsabile, può accedere al successore e se può gestire la richiesta, lo fa, altrimenti la inoltra al successore;
Client invia la richiesta chiamando un metodo sul primo elemento della catena, ignorandone il tipo effettivo.
Il successore di un Handler può essere passato direttamente al costruttore oppure tramite un setter.
Nel primo caso, dopo che abbiamo creato la catena, non la possiamo più modificare, mentre nel secondo caso è consentito stando però attenti al non creare cicli.

14.3 Conseguenze

Reduce coupling, Client e ConcreteHandler sono scollegati, liberiamo il Client dal dover sapere (e non deve sapere) chi gestirà la sua richiesta.
Un oggetto della catena non conosce la struttura della stessa ma conosce solamente il suo successore.
Maggiore flessibilità nell'assegnazione delle responsabilità agli oggetti dove la catena viene costruita a run-time e le responsabilità possono essere aggiunte o rimosse modificando la stessa.
Ricezione non garantita, ovvero, siccome la richiesta non ha un gestore esplicito, non c'è garanzia che venga gestita (la richiesta scorre tutta la catena).

14.4 L'inoltro

Come nel Decorator, per evitare di dimenticarsi l'inoltro, possiamo definire la logica dell'inoltro nella classe base (metodo final), chiamare al suo interno un metodo astratto, che conterrà la logica della gestione della richiesta che sarà poi implementata dalle sottoclassi.

14.5 Decorator vs Chain of responsability

Entrambi i pattern sembrano simili ma

ha due classi principali astratta;	ha una classe astratta;
tutte le classi gestiscono la richiesta;	la richiesta sarà gestita da *una ed una sola* classe oppure non viene gestita;
il componente da decorare non deve essere null;	l'ultimo componente della catena è null;
pattern strutturale	pattern comportamentale

14.6 Lambda

Come il Decorator, anche questo pattern simula la composizione di funzioni.
Quindi possiamo pensare di usare una lista di lambda, scorriamo questa lista che potrà terminare perchè troveremo una lambda che riesce a gestire la richiesta oppure perchè lista si esaurisce in quanto non abbiamo una lambda in grado di gestire la richiesta.
Però non stiamo più parlando del pattern Chain of Responsability.

Visitor

E' un pattern comportamentale che consente di definire una nuova operazione senza modificare le classi degli oggetti su cui opera.
Questo pattern è utile quando si ha una gerarchia di classi e si desidera eseguire diverse operazioni su di esse senza dover aggiungere nuovi metodi a ciascuna classe.
Le operazioni variano a seconda del tipo di classe concreta.

15.1 Funzionamento

Avremo un'interfaccia comune, AbstractVisitor, che definisce metodi del tipo visitXXX per ogni tipo **concreto** della struttura e le sottoclassi concrete che la estenderanno (implementeranno tutti i visitXXX).
Nella classe base/interfaccia della gerarchia aggiungiamo il metodo accept(...) che prende in input un AbstractVisitor e le classi concrete della gerarchia implementeranno questo metodo chiamando il corrispettivo metodo visitXXX passando se stesso come input.
Questo pattern sopperisce alla mancanza del **double dispatch** (overloading dinamico).
Supponiamo di prendere in considerazione l'esempio dell'esercitazione Expression.
Quindi avremo la nostra interfaccia ExpressionVisitor con due metodi, visitConstant e visitSum che prendono in input, rispettivamente, un Constant ed un Sum e una sua implementazione concreta, ParenthesisRepresentationVisitor che la implementa.
Nell'interfaccia Expression avremo il metodo accept(...) che prende in input ExpressionVisitor ed infine avremo le classi Constant e Sum che implementeranno accept(...) chiamando rispettivamente visitConstant e visitSum.

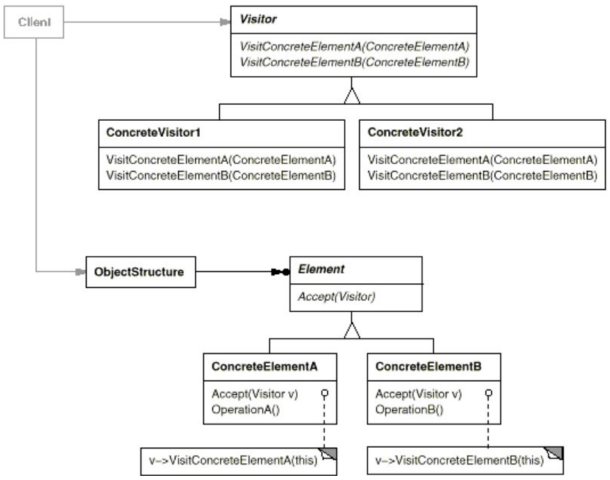
15.2 Double dispatch

Supponiamo di avere il seguente test

```
@Test
public void testReprImproved() {
    ExpressionVisitor r = new ParenthesisRepresentationVisitor();
    Expression exp = new Sum(
                      new Constant(10),
                      new Constant(5)
                      );
    assertEquals("(10 + 5)", exp.accept(r));
}
```

Quando chiamiamo exp.accept(r), exp, staticamente, è di tipo Expression ma a runtime è di tipo Sum e quindi, per il binding dinamico, viene chiamato visitSum.
VisitSum, staticamente, prende in input un oggetto di tipo ExpressionVisitor che, a runtime, è di tipo ParenthesisRepresentationVisitor, quindi per il binding dinamico, viene chiamato ParenthesisRepVisitor.visitSum.
Ecco perchè double dispatch, ovvero abbiamo usato due volte il meccanismo del binding dinamico.

15.3 Collaborazioni



Un client crea un ConcreteVisitor e visita la struttura a oggetti.
Quando un elemento viene visitato chiama il metodo del visitor che corrisponde alla sua classe, passando se stesso come argomento, in questo modo il visitor può accedere allo stato dell'oggetto, se necessario
Aggiungere una nuova classe alla gerarchia, come Multipliction, comporterà aggiungere un nuovo metodo visitXXX, in ExpressionVisitor, che prenderà in input Multipliction ed implentare accept(...) in Multipliction.
L'aggiunta di un nuovo metodo in ExpressionVisitor, comporterà ad errori di compilazione nei visitor concreti, quindi occorrerà implementarli.

15.4 Varianti Visitor

Supponiamo di voler rimuovere il metodo eval() da Expression e di implementare la valutazione di un espressione tramite il EvalVisitor, tale che EvalVisitor<:ExpressionVisitor.

15.4.1 Visitor generico

Dall'esempio precendete, noi sappiamo che in ExpressionVisitor i metodi visitXXX ritornano una stringa, ma a noi servirebbe che i metodi visitXXX di EvalVisitor ritornino un integer, quindi rendiamo ExpressionVisitor generico e, a sua volta, i metodi visitXXX e accept sono generici.
Andremo a specificare l'argomento di tipo solamente quando andremo ad implementare Expression, ParenthesisRepresentationVisitor e EvalVisitor e nei test, in quanto non possiamo instanziare in tipo generico, instanziamo direttamente il visitor concreto.

15.4.2 Visitor void

I metodi visitXXX e accept sono void e per questo motivo ogni visitor concreto si deve mantere uno stato, che viene usato per accumulare i risultati durante la visita, ed infine fornire un metodo per ottenere il risultato finale di tutta la vista.

15.4.3 Coseguenze

Affinchè i visitor possano agire sugli elementi, l'interfaccia degli elementi deve fornire operazioni pubbliche di accesso e questo compromette l'incapsulamento.
E' facile aggiungere nuove operazioni, basta aggiungere un visitor completo.
E' difficile aggiungere un nuovo elemento da visitare, bisogna definire un nuovo metodo in visitor che poi dovrà essere implementato dalle classi concrete e il nuovo elemento da visitare deve implementare accept.

N.B. Quando aggiungiamo il metodo visitXXX per la nuova classe da visitare al Visitor, i visitor concreti non compileranno più fino a quando non implementeremo il nuovo visitXXX in ogni visitor concreto.

Per questo motivo, il visitor andrebbe usato quando la gerarchia degli oggetti è *stabile*.
Una soluzione a questo, potrebbe essere quella di fornire un'implementazione dei metodi visitXXX in una classe astratta che implementa Visitor, cosicchè i client che implementeranno Visitor, estenderanno la classe astratta e ridefiniranno i metodi di loro interesse (questo però è un altro pattern).
Simulando il binding dinamico, facciamo a meno di instanceof e downcast.
Inoltre, se il linguaggio permettesse l'overloading dinamico, allora non ci sarebbe bisogno del metodo accept nelle classi da visitare ma comunque, quest'ultime, dovrebbero sempre fornire punti di accesso alla classe.

ADAPTER

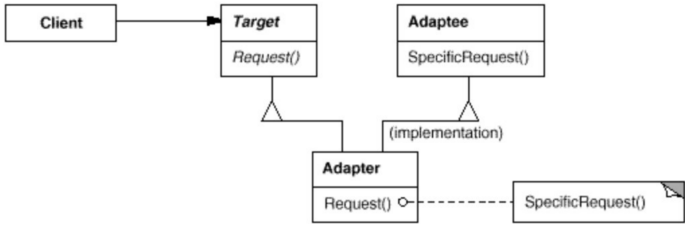
Il pattern Adapter è un design pattern strutturale che consente a due interfacce incompatibili di lavorare insieme, nello specifico consente a una classe di funzionare da adattatore tra queste due interfacce, consentendo loro di collaborare senza dover modificare il loro codice sorgente.

16.1 Applicabilità

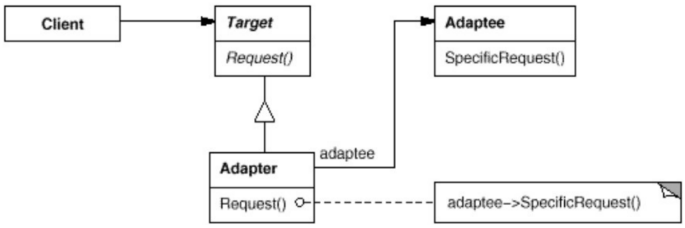
Vogliamo usare una classe esistente ma la sua interfaccia non è compatibile con l'interfaccia che ci serve oppure vogliamo creare una classe riusabile che collaborerà con classi scorrelate.

16.2 Struttura

Abbiamo due varianti di questo pattern, una con ereditarietà ed una con object composition e delega.
Nella prima variante abbiamo una classe, Adapter, che implementa l'interfaccia che ci serve, estendendo un'altra classe esistente, Adaptee, cosicchè i metodi dell'interfaccia chiameranno i metodi della superclasse Adaptee.



Nella seconda variante, Adapter implementa l'interfaccia che ci serve delegando ad un Adaptee e i metodi dell'interfaccia chiamarenno dei metodi sull'oggetto Adaptee.



Target definisce l'interfaccia del dominio usata dai Client;
Client collabora con gli oggetto conformi all'interfaccia Target;
Adaptee l'interfaccia esistente da adattare;
Adapter adatta l'interfaccia Adaptee a Target.

Il Client, a runtime, chiama operazioni su oggetto Adapter, staticamente di tipo Target, che a sua volta gestisce le operazioni chiamando in modo opportuno delle operazioni di Adaptee.

16.3 Conseguenze

Nella versione basata su ereditarietà

- la classe Adapter adatta un certo tipo concreto di Adaptee, quindi se volessi adattare una sottoclasse di Adaptee, allora dovremmo creare un nuovo adapter;
- possiamo ridefinire i metodi Adaptee in quanto è superclasse di Adapter;
- non introduciamo un nuovo livello di indirezione perchè non c'è object composition.

Nella versione basata su object composition

- si può adattare qualsiasi istanza di Adaptee;
- possiamo aggiugere del comportamento prima o dopo le operazioni di Adaptee;
- introduciamo un ulteriore di livello di indirezione.

N.B. In linguaggi tipo java, in cui ereditarietà implica subtyping, bisogna usare la versione object composition.

16.4 Altro utilizzo di Adapter

Esiste un altro caso d'uso di questo pattern.
Supponiamo di avere un'interfaccia con tanti metodi astratti che sono correlati tra loro (altrimenti si violerebbe ISP) e di avere una classe che la implementa.
Implementare un'interfaccia significa implementare tutti metodi dell'interfaccia anche se a noi iteressano determinati metodi.
Ecco che interviene l'Adapter, che fa da ponte tra l'interfaccia, dando un'implementazione di default a tutti i metodi dell'interfaccia (tipicamente vuota), e la nostra classe, che estende Adapter e reimplementa solo i metodi che gli servono.

N.B.L'implementazione di default si applica facilmente solo ai metodi void.

16.5 Adapter vs default methods

Questo pattern è un'alternativa ai default methods in quanto invece di stabilire una volta per tutte l'implementazione di default nell'interfaccia, con l'Adapter, noi non la tocchiamo e lasciamo più libertà allo sviluppatore.

FACADE

Pattern comportamentale (descrive come classi e oggetti interagiscono e si distribuiscono le responsabilità) con l'intento di fornire un'interfaccia unificata ad un set di interfacce di un sottosistema complesso in modo tale da semplificarne l'utilizzo, nascondendo i dettagli interni del sottosistemadi interfacce e delegando alle stesse, rendendo il codice più pulito.
Il suo intento è quello di fornire vista semplificata di default, di alto livello disaccoppiando i client dai dettagli del sottosistema e facilitanto la strutturazione a strati dove uno strato comunica col sottostante attraverso il facade.
Usando i pattern e seguendo i principi realizziamo sistemi complessi costituiti da tanti piccoli componenti, disaccoppiati il più possibile e facilmente intercambiabili (riusabili), eventualmente anche a runtime.
Chi vuole, può manipolare e interagire direttamente coi singoli componenti tramite tipi astratti oppure può interagire con la "facciata" semplificata dei Facade.

17.1 Esempio Visitor con Expression

Per visitare un'espressione avevamo a disposizione due visitor, ExpressionTypeSystemVisitor ed ExpressionEvaluatorVisitor.
Tutti i client, per avere il tipo di un'espressione, dovevano creare un visitor e chiamare il metodo accept su un'Expression

```
    Expression e = new Multiplication(new Constant(1), new Constant(2));
    Class<?> result = e.accept(new ExpressionTypeSystemVisitor());
```

Con il facade basta aggiungere un intermediario, ExpressioneTypeSystem ed ExpressionEvaluator, che attraverso un metodo, delegano al visitor appropriato

```
    public class ExpressionTypeSystem {
        public Class<?> computeType(Expression e) {
            return e.accept
                    (new
                        ExpressionTypeSystemVisitor());
        }
    }
```

```
    Expression e = new Multiplication(new
        Constant(1), new Constant(2));
    Class<?> result = new ExpressionTypeSystem()
                        .computeType(e);
```

Stessa cosa con ExpressionEvaluator dove il metodo, in questo caso eval(), farà accept di un ExpressionEvaluatorVisitor su un'Expression.
Nell'esempio del visitor, il valutatore dava per scontato che le espressioni fossero già state controllate dal type system e quindi i client dovevano usare i due componenti nell'ordine giusto, prima il type system e poi, se non ci sono errori di tipo, il valutatore

```
Expression e = new Multiplication(new Constant(1), new Constant(2));
Class<?> type = new ExpressionTypeSystem().computeType(e);
// in case catch exception
Object result = new ExpressionEvaluator().eval(e)
```

Quindi, a questo possiamo nascondere ulteriormente, ovverro aggiungere un nuovo facade del tipo

```
public class ExpressionInterpreter {
    public Object interpret(Expression expression) throws ExpressionInterpreterException {
        try {
            // ignore type result, as long as it's well-typed
            new ExpressionTypeSystem().computeType(expression);
            return new ExpressionEvaluator().eval(expression);
        } catch (TypeSystemException e) {
            throw new ExpressionInterpreterException("Expression not well-typed", e);
        }
    }
}
```

cosicchè alla fine i Client dovranno solamente scrivere

```
Expression e = new Multiplication(new Constant(1), new Constant(2));
// in case catch exception
```

```
Object result = new ExpressionInterpreter().interpret(e);
```

N.B. I Client possono comunque usare direttamente tutto quello che c'è dietro.

17.2 Altro caso di utilizzo

Supponiamo che il client sia il nostro codice ed il sottosistema siano librerie esterne.
Il Facade farà da tramite fra il nostro codice e la libreria esterna cosicchè se un giorno decidessimo di cambiare la librearia, dovremo solo modificare il Facade.