

Indice

1	Lambda expression	2
1.1	Intro	2
1.2	Lambda expression	3
1.3	Method references	4
1.4	Scope di una lambda	4
2	Stream	5
2.1	Creazione di uno stream	5
2.1.1	Stream infiniti	5
2.2	Operazioni intermedie	6
2.3	Operazione di riduzione di uno stream	6
2.3.1	Optional	6
2.4	Operazione di raccolta di uno stream	6
3	INFORMATION HIDING	8
3.1	Design Pattern VS Design Principle	9
4	ACRONIMO SOLID	10
4.1	Single Responsibility Principle (SRP)	10
4.2	Open-Closed Principle (OCP)	10
4.3	Liskow Substitution Principle (LSP)	12
4.4	Interface Segregation Principle (ISP)	14
4.5	Dependency Inversion Principle (DIP)	14
5	COVARIANZA E CONTROVARIANZA	16
6	DESIGN PATTERN	19
6.1	Notazione Uml	19
6.2	Interaction diagram	19
7	TEMPLATE E STRATEGY	20
7.1	Template Method	20
7.1.1	Struttura e partecipanti	20
7.2	Strategy	21
7.2.1	Struttura e partecipanti	21
7.2.2	Differenza con il templateMethod	22
8	COMPOSITE	23
8.1	Struttura	23
9	ITERATOR	25
9.1	Struttura	25
9.2	Iterator vs Stream	25
10	PATTERN CREAZIONALI	26
10.1	PERCHE' ?	26
10.2	Factory Method	26
10.2.1	Struttura	26
10.2.2	Factory Method vs Template Method	26
10.2.3	Principale utilizzo	27
10.2.4	Vantaggi e svantaggi	27
10.3	Abstract Factory	27
10.3.1	Struttura	27
10.3.2	Principale utilizzo	27
10.3.3	Accedere e modificare la factory	28
10.3.4	AbstractFactory vs Factory Method	28
10.4	Static Factory Methods	28
10.5	Singleton	28
10.6	Fluent Interface	29
10.7	Builder	29

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 restituisce 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'istanza 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 effettivamente confrontare gli oggetti (*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);
```

- `Class::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 purché 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' sì.

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 precedenti 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 generare 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 precedente

```
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 funzione 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'istanza 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'istanza, privati, che però vi si possono 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 instazziassi 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 vincolo 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 funzionare, sarà colpa loro.

Più dettagli interni nascondiamo, più avremo disaccoppiamento e più sarà facile testare singole componenti in isolamento.

Se dichiarassimo qualcosa come public/protected, come le API, e volessimo effettuare dei cambiamenti, 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 rompe 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 pattern 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 Responsibility 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 ricompilazione 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'ereditarietà 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 implementativi 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.

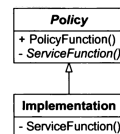
Questo porta al concetto della **fragile base class problem**, ovvero una classe è considerata fragile se piccole modifiche corrette e sicure alla classe base, possono portare le classi derivate a non funzionare più correttamente.

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 chiusa 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 configurabile rispetto al log ma non è aperta ad estensioni del log in quanto la sua gestione dipende da un if-else/switch e quando 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'interfaccia 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, ovvero 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 sottotipo 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 sottotipo 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 principio 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'interfaccia, Shape, con all'interno la definizione del metodo area.

Così facendo, entrambe le classi concrete, implementeranno Shape e ridefiniranno, 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 Shape, 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 contenente 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.

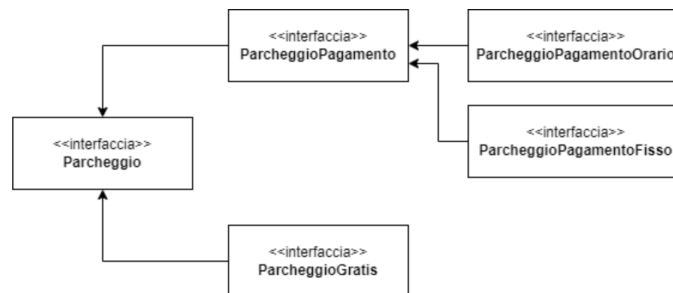
Prendiamo ad esempio il sistema di pagamento di un parcheggio

```
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 parcheggio 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, implementerà 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.

Tradizionalmente, 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'ultimo, 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.

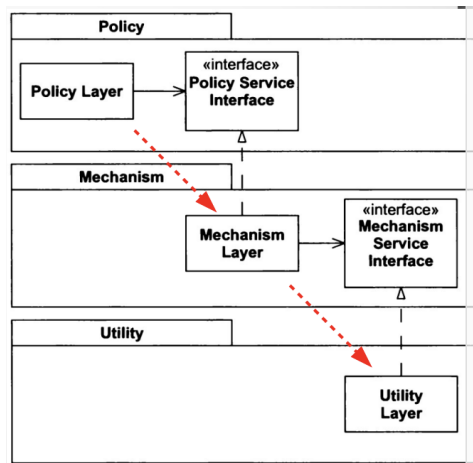
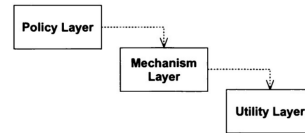
Il layer di alto livello usa solo il layer sottostante ma usare significa che dipende dal layer sottostante.

Inoltre 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 $f: D \rightarrow C$, può essere sostituita da una funzione g , tale che $g: D' \rightarrow C'$, sse $C' \leq C$ (**covariante** sul tipo di **ritorno**) e $D \leq D'$ (**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 $\text{Manager} \leq \text{Employee}$.

Employee ha un metodo `setSalary()` e Manager ha il metodo `setBonus(int)`.

Prendiamo altre due classi, EmployeeDB e ManagerDB, tale che $\text{ManagerDB} \leq \text{EmployeeDB}$.

La prima ha al suo interno un metodo `find(int)` che restituisce 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 eseguiassi 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` restituisce 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());
```

Come detto prima, 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 genererà 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'approccio 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 sì, 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 riutilizzabile 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 riutilizzabile.

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 sopprimono alle mancanze di funzionalità/caratteristiche del linguaggio di programmazione (ad esempio il Visitor sopprime 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.

7.1 Template Method

Definisce, in unico punto, lo scheletro di un algoritmo e delega alcuni passi, quelli che possono variare da oggetto ad oggetto, alle sottoclassi che avranno il compito di ridefinirli.

Prendiamo come esempio un framework che fornisce la classe `Application` e `Document` responsabili, rispettivamente, di aprire un documento e salvarlo e di rappresentare le informazioni di un documento una volta 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 overrideati dalle sottoclassi.

N.B. Il template method non sono i metodi astratti ma è il metodo che li chiama che definisce lo scheletro dell'algoritmo.

```
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();
    }

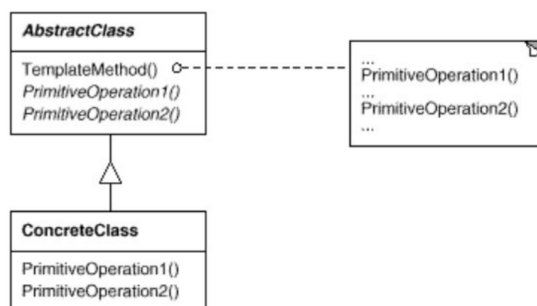
    protected abstract void aboutToOpenDoc(Document doc);
    protected abstract Document doCreateDocument(String path);
    protected abstract boolean canOpenDocument(String path);
}
```

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 cambierà mai, allora potremmo renderlo final.

7.1.1 Struttura e partecipanti



AbstractClass

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

ConcreteClass che implementa le operazioni primitive.

7.2 Strategy

Definisce una famiglia di algoritmi, incapsula ogni algoritmo e li rende intercambiabili, nascondo le informazioni interne dell'implementazione.

I Client devono essere consapevoli che esistono diverse strategie, avremo un overhead di comunicazione tra Context e Strategy e numero di oggetti nel sistema sarà maggiore.

Il pattern deve essere usato solo quando la *variazione del comportamento* è fondamentale per i client.

Prendiamo ad esempio un Shop che può applicare più sconti, ad esempio quello percentuale, assoluto o nessuno sconto.

```
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);  
    }  
}
```

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.

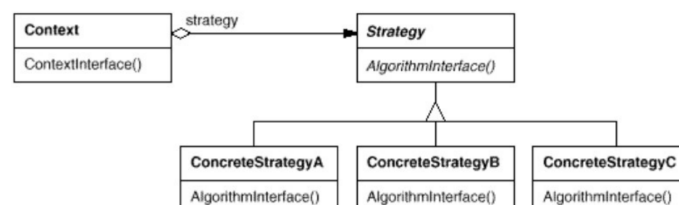
```
public interface DiscountStrategy {  
    int applyDiscount(int originalPrice);  
}
```

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

```
public class AbsoluteDiscountStrategy  
    implements DiscountStrategy {  
    private int discount;  
  
    public AbsoluteDiscountStrategy(int  
        discount) {  
        this.discount = discount;  
    }  
  
    @Override  
    public int applyDiscount(int originalPrice) {  
        return originalPrice - discount;  
    }  
}
```

```
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 Struttura e partecipanti



Strategy dichiara l'interfaccia comune a tutti gli algoritmi supportati.

ConcreteStrategy implementa l'interfaccia Strategy e implementa l'algoritmo .

Context usa uno Strategy quando ha bisogno dell'algoritmo che, a runtime, sarà effettivamente implementato da un ConcreteStrategy.

7.2.2 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 risolvere* l'algoritmo, 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 strategy, 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 Ordinamento 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.

COMPOSITE

Pattern Strutturale, 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.

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.

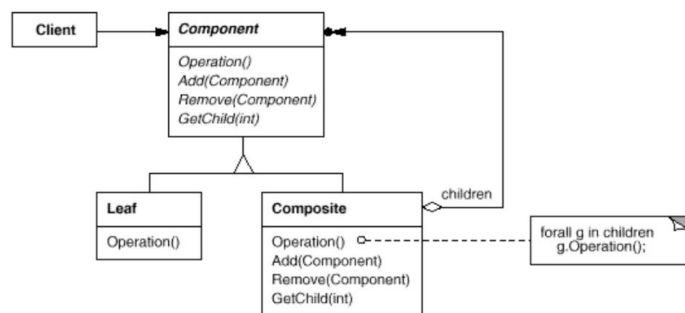
Se un composto deve avere solo un certo tipo di figli, probabilmente il pattern non fa al caso nostro.

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 relative 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.

Si nota che le operazioni di gestione dei figli sono in **Component**, quindi **Leaf** li dovrebbe overriding 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.

Nella variante type safe il client dovrebbe avere un riferimento alla radice della struttura di tipo statico Composite, in caso contrario non potrebbe chiamare i metodi per gestire i figli.

ITERATOR

Pattern comportamentale 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 di studenti.

Se restituissimo la collezione con un classico getter pubblico, esporremmo troppi dettagli interni ai client permettendogli di manipolare direttamente la collezione.

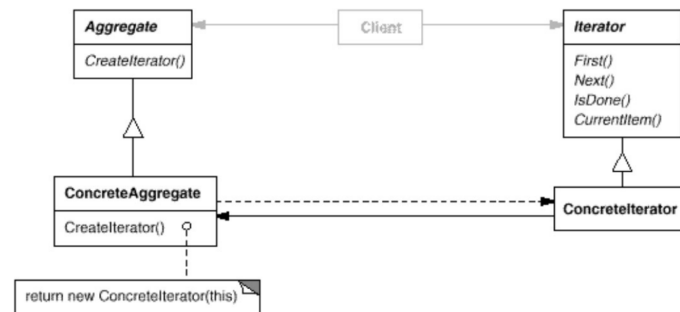
Se un giorno decidessi di cambiare l'implementazione della collezione, i client ne risentiranno.

Un getter protected non migliorerebbe la situazione in quanto ai client esterni basterebbe estendere la nostra classe per usare quel getter.

Invece facendo tornare un iteratore, non forniamo un punto di accesso alla collezione, infatti se decidessi di passare da collezione ad un set, non avrei problemi di compilazione (col getter classico li avrei avuti).

N.B. Un iteratore non conosce il numero di elementi di un aggregato, può solamente attraversarli.

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 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'istanza avviene in un metodo (Factory method) che può essere astratto per poi essere implementato dalle sottoclassi oppure concreto per poi essere ridefinito dalle sottoclassi.

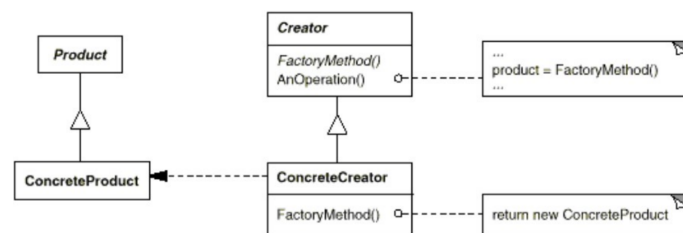
L'assegnazione dell'istanza viene salvata in una variabile, sempre nella classe base, privata e chi implementa/estende l'interfaccia/classe astratta avrà il compito di implementare/ridefinire il factory method.

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.

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.

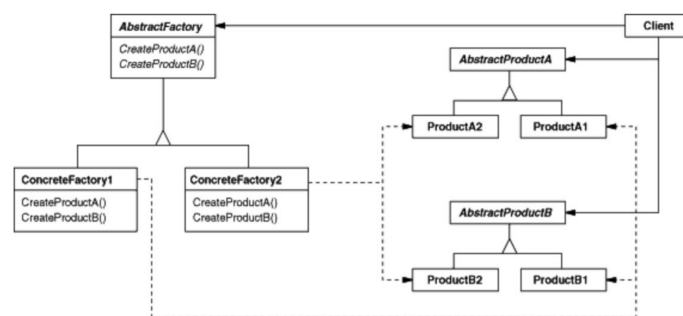
In compenso, siamo obbligati a creare una sottoclasse di Creator per creare una certa sottoclasse di Product, un'alternativa è quella di usare le lambda (però non stiamo più parlando del pattern factory method).

Nel costruttore passiamo l'interfaccia funzionale Supplier ed usiamo il suo metodo astratto, get, nei metodi in cui assegniamo l'istanza.

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 istanziare 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 ConcreteFactory.

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, cosicché le factory sono le uniche a poter istanziare 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 istanza 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 istanziati attraverso la factory precedente.

Se la factory è implementata tramite Singleton, allora cambio l'istanza, se è implementata tramite setter, passo la nuova istanze oppure se è passata tramite costruttore, allora si ricreano le istanze 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 costruttore 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'istanza 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'istanza.

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 (ritornano 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 accendere 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();

```