BUILDER PATTERN UN OTTIMA ALTERNATIVA AL COSTRUTTORE

5 Comments / Di Dario Frongillo / In Design Pattern, Java / 13 Dicembre 2017


Di Dario Frongillo
13 Dicembre 2017

IL PROBLEMA DEI COSTRUTTORI

Quando una classe in Java comincia ad avere molti campi, la creazione di un oggetto tramite costruttore diventa un qualcosa di infernale per il programmatore. Facciamo un esempio per chiarirci le idee:

```
1.  public class Animal {
2.      public enum Sex{
3.          MALE,
4.          FEMALE
5.      }
6.
7.      /*
8.      pedigreeCode campo obbligatori e non modifica
9.      */
10.     private final String id;
11.
12.     private String name;
13.     private String pedigreeName;
14.
15.     private String owner;
16.     private String race;
17.     private String residence;
```


Di Dario Frongillo
13 Dicembre 2017


```
18.     private Boolean isVaccinated;
25.     //getter e setter
26.
27.     public Animal(String name, String pedigreeName, String id,
        String owner, String race, String residence, Boolean
        isVaccinated, Boolean isChampion, List sons, Sex sex, Double
        weight, Double height) {
28.         this.name = name;
29.         this.pedigreeName = pedigreeName;
30.         this.id = id;
31.         this.owner = owner;
32.         this.race = race;
33.         this.residence = residence;
34.         this.isVaccinated = isVaccinated;
35.         this.isChampion = isChampion;
36.         this.sons = sons;
37.         this.sex = sex;
38.         this.weight = weight;
39.         this.height = height;
40.     }
41.
42.     public Animal(String id,String name, String pedigreeName)
        {
43.         this.name = name;
44.         this.pedigreeName = pedigreeName;
45.         this.id = id;
46.     }
47.
48.     public Animal(String id, String owner, String race, String
        residence) {
49.         this.id = id;
50.         this.owner = owner;
51.         this.race = race;
52.         this.residence = residence;
53.     }
54.
55.     public Animal(String id){
56.         this.id = id;
57.     }
58. }
```

Inanzitutto notiamo che se alcuni campi sono opzionali nella

creazione di un oggetto le alternative sono due:

come valore per i campi non usati.

2. Creare diverse versioni di costruttori con un differente set di campi da specificare. Soluzione con meno margine di errore nell'utilizzo ma più costosa per il programmatore in fase di sviluppo di una classe.

In entrambe le soluzioni la probabilità di errore è altissima.

```
1.  Animal pluto = new Animal("pluto", "123", "PlutoSecondo",
2.                            "labrador","Marco Rossi","Via
x",true, false,
3.                             null, Animal.Sex.MALE,40.5,30.0);
```

Nell'esempio precedente abbiamo specificato labrador come owner e Marco Rossi come razza! Effettivamente, utilizzando il costruttore per posizione dei parametri, fare un errore è facilissimo, per non parlare in fase di lettura del codice: è difficilissimo dire, a colpo d'occhio, cosa vogliano dire i dodici parametri utilizzati nel costruttore in fase di instansazione di un oggetto, a meno di andare a consultare il javadoc (se presente) o direttamente il codice sorgente della classe, con conseguente

perdita di tempo.
Un'alternativa può essere quella di trasformare la classe Animal in un JavaBean con un costruttore senza parametri e un setter per

ciascun parametro obbligatorio o opzionale. Inoltre nulla assicura

Di Dario Frongillo
13 Dicembre 2017

Questo articolo pone una possibile soluzione molto utile da utilizzare, in alternativa al costruttore, quando una classe ha molti parametri: il **builder pattern**.

BUILDER PATTERN - TEORIA

Il builder pattern è uno dei più importanti pattern, meglio conosciuti come **GoF design pattern** (per i novelli del mestiere i Gof sono gli autori del libro **Design Patterns: Elements of Reusable Object-Oriented Software** ... se non avete letto questo libro correte subito a comprarlo...la bibbia del programmatore!).

Nella programmazione ad oggetti tale pattern è molto di voga poiché separa la costruzione di un oggetto complesso dalla sua rappresentazione, cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. In questo modo l'algoritmo per la creazione di un oggetto complesso è indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate. Ciò ha l'effetto immediato di rendere più semplice la classe, permettendo a una classe builder separata di focalizzarsi sulla corretta costruzione di un'istanza e lasciando che la classe originale si

concentri sul funzionamento degli oggetti. Questo è

Di Dario Frongillo
13 Dicembre 2017

costruire un oggetto passo-passo, cosa che si può verificare quando si fa il parsing di un testo o si ottengono i parametri da un'interfaccia interattiva. Per maggiori dettagli e spiegazioni dal punto di vista formale e teorico vi rimando alla lettura di **Design Patterns: Elements of Reusable Object-Oriented Software.**

BUILDER PATTERN - PRATICA

La versione di BuilderPattern che proporro' in questo articolo e' quella descritta da **Joshua Bloch** nel suo libro **Effective Java**: una versione migliorata del modello di builder originale la quale si presente come una soluzione chiara e altamente leggibile facendo un uso di fluent design.

Proviamo ad implementare il builder pattern sulla classe Animal mostrata ad inizio articolo.

```
1.  public final class AnimalBuilder {
2.
3.   private String id;
4.   private String name;
5.   private String pedigreeName;
6.   private String owner;
```

Di Dario Frongillo
13 Dicembre 2017

5 Comments

```
7.  private String race;
14.  private Double height;
15.
16.  private AnimalBuilder(String id) {
17.  this.id = id;
18.  }
19.
20.  public static AnimalBuilder newBuilder(String id) {
21.  return new AnimalBuilder(id);
22.  }
23.
24.  public AnimalBuilder name(String name) {
25.  this.name = name;
26.  return this;
27.  }
28.
29.  public AnimalBuilder pedigreeName(String pedigreeName) {
30.  this.pedigreeName = pedigreeName;
31.  return this;
32.  }
33.
34.  public AnimalBuilder owner(String owner) {
35.  this.owner = owner;
36.  return this;
37.  }
38.
39.  public AnimalBuilder race(String race) {
40.  this.race = race;
41.  return this;
42.  }
43.
44.  public AnimalBuilder residence(String residence) {
45.  this.residence = residence;
46.  return this;
47.  }
48.
49.  public AnimalBuilder isVaccinated(Boolean isVaccinated) {
50.  this.isVaccinated = isVaccinated;
51.  return this;
52.  }
53.
54.  public AnimalBuilder isChampion(Boolean isChampion) {
55.  this.isChampion = isChampion;
```

```
56.    return this;
63.
64.    public AnimalBuilder sex(Animal.Sex sex) {
65.    this.sex = sex;
66.    return this;
67.    }
68.
69.    public AnimalBuilder weight(Double weight) {
70.    this.weight = weight;
71.    return this;
72.    }
73.
74.    public AnimalBuilder height(Double height) {
75.    this.height = height;
76.    return this;
77.    }
78.
79.    public Animal build() {
80.    return new Animal(name, pedigreeName, id, owner, race,
       residence, isVaccinated, isChampion, sons, sex, weight,
       height);
81.    }
82.   }
```

Adesso un oggetto di tipo Animal può essere istanziato anche nel seguente modo:

```
1.   Animal pluto2 = AnimalBuilder.newBuilder("0000001")
2.    .name("0000001")
3.    .pedigreeName("PlutoSecondo")
4.    .owner("Marco Rossi")
5.    .race("labrador")
6.    .residence("Via x")
7.    .isVaccinated(true)
8.    .isChampion(false)
9.    .sons(null)
10.   .sex(Animal.Sex.MALE)
11.   .weight(40.5)
12.   .height(30.0)
13.   .build();
```

Builder, evitando equivoci ed errori nel passaggio dei valori. Il risultato è quello di ottenere un codice chiamante più facile da leggere e da scrivere.

Un altro vantaggio di questo pattern è la possibilità di istanziare oggetti cloni o simili a quello appena creato, minimizzando il codice da scrivere. Supponiamo di voler instanziare due animali identici e un terzo animale identico, ma di sesso opposto. Senza utilizzare un nuovo builder, è sufficiente chiamare ancora `build` dopo aver modificato i parametri del nuovo oggetto, come dimostrato dal seguente codice:

```
1.  AnimalBuilder animalBuilder =
AnimalBuilder.newBuilder("0000001")
2.          .name("0000001")
3.          .pedigreeName("PlutoSecondo")
4.          .owner("Marco Rossi")
5.          .race("labrador")
6.          .residence("Via x")
7.          .isVaccinated(true)
8.          .isChampion(false)
9.          .sons(null)
10.         .sex(Animal.Sex.MALE)
11.         .weight(40.5)
12.         .height(30.0);
13.
14.  Animal animal3A = animalBuilder.build();
15.  Animal animal3AClone = animalBuilder.build();
16.  Animal animal3B =
animalBuilder.sex(Animal.Sex.FEMALE).build();
```

principio nel nostro esempio inserendo i check di validazione all'interno del metodo `build` prima della creazione di `Animal`; dato che questo metodo è l'unico punto della classe adibito alla creazione dell'oggetto:

```
1.  public Animal build() {
2.      if(weight>200){
3.          throw new IllegalArgumentException("Animale troppo
    pesante");
4.
5.      }
6.      if(!isVaccinated){
7.          throw new IllegalArgumentException("Animale non
    vaccinato");
8.
9.      }
10.     return new Animal(name, pedigreeName, id, owner, race,
    residence, isVaccinated, isChampion, sons, sex, weight,
    height);
11.  }
```

BUILDER COME INNER CLASS

Non è strettamente necessario che la Builder class sia una classe definita in un file a parte. È possibile definire la builder class come static inner class della classe di definizione dell'oggetto. Trovo

personalmente più confusionario questo approccio, pertanto non

Di Dario Frongillo
13 Dicembre 2017

SVANTAGGI DEL BUILDER

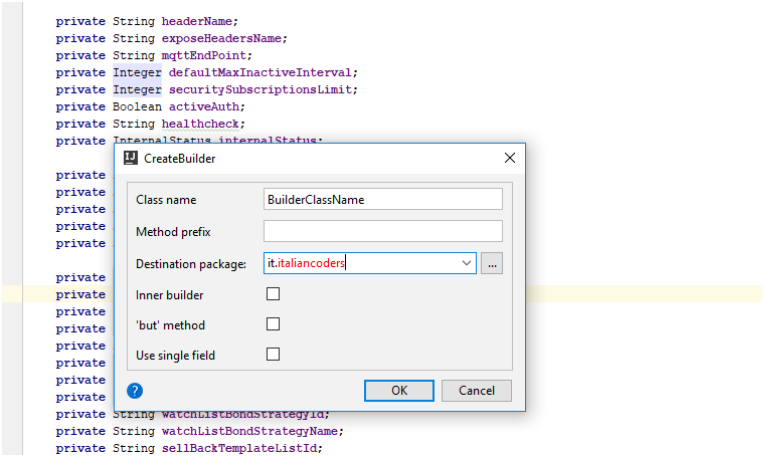
Parliamo adesso dei svantaggi nell'utilizzare il builder nei propri progetti. L'unico reale svantaggio è quello di dover definire una builder class per classe con conseguente incremento del tempo di sviluppo. A mio avviso il tempo investito nella preparazione del builder viene sempre ripagato, velocizzando la comprensione del codice client e la manutenzione. Esistono però possibili soluzioni per automatizzare la creazione delle builder class. Riporto le tecniche più utilizzate oggi.

BUILDER IDE PLUGIN

Gli IDE più utilizzati oggi (ad es. Eclipse, Intellij, ecc) offrono dei fantastici plugin per automatizzare la creazione di Builder Class. Io personalmente utilizzo Builder Generator, facilmente scaricabile

dal menu Plugins di Intellij. Accedendo agli shortcut di Intellij con ALT+INS puoi autogenerare in un secondo la classe di Builder della classe corrente scegliendo:

- package di destinazione
- inner builder: se spuntata questa opzione il plugin creerà il builder all'interno della classe corrente come inner builder class.



BUILDER CON LOMBOK

Un altra ottima alternativa per automatizzare la creazione del builder pattern con Lombok. Per sapere di più su cos'è Lombok

leggetevi questo esaustivo nostro articolo: LINK.

Utilizzando la seguente annotation sopra la classe

programmatore non ci sarà nessun side-effect: riuscirete ad utilizzare il builder method senza aver scritto una riga di codice!

```
AnimalLombok plutoLombok = AnimalLombok.newBuilder()
        .id("0000001")
        .name("0000001")
        .pedigreeName("PlutoSecondo")
        .owner("Marco Rossi")
        .race("labrador")
        .residence("Via x")
        .isVaccinated(true)
        .isChampion(false)
        .sons(null)
        .sex(Animal.Sex.MALE)
        .weight(40.5)
        .height(30.0)
        .build();
```

CONCLUSIONI

Vi consiglio altamente di utilizzare il builder pattern nei grossi progetti per i motivi che ho citato in questo articolo: io non posso

più farne a meno!
Trovate gli esempi di questo articolo sul mio GITHUB

Di Dario Frongillo
13 Dicembre 2017

- Design Patterns: Elements of Reusable Object-Oriented Software (Autori: Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm)
- Effective Java, Joshua Bloch

builder java OOP pattern

A PROPOSITO DI ME

DARIO FRONGILLO

Fondatore di Italiancoders e dell iniziativa devtalks.
Software architect con un forte background in Java,


Di Dario Frongillo
13 Dicembre 2017

5 Comments

Mi occupo di sviluppo soluzioni software in ambito Banking e Finance: in particolare progettazione e sviluppo di applicativi web based per la realizzazione di sistemi di trading, interfacce con i mercati finanziari e di servizi real time.

ALTRO

Magallo • 2 anni fa

Salve a tutti. Volevo far notare un dubbio che ho. Ma è normale che

nel builder ci sia una duplicazione di tutti i membri interni della

Di Dario Frongillo
13 Dicembre 2017

5 Comments

1 • Rispondi • Condividi ›

Glody Fimpa • 3 anni fa

A me Lombok non funziona, dà un errore che il metodo newBuilder() non esiste su la classe ch usa Lombok per produrre il Builder. Provando ad eseguirlo come Java Application non compila proprio. Io sto usando Eclipse. L'errore è nel Main, non nella classe Bean. Tu come fai ad usarlo ? Grazie !!!

• Rispondi • Condividi ›

Dario Frongillo admin ➚ Glody Fimpa • 3 anni fa

Ciao Glody ti manca il plugin .. prova a installare il plugin e fammi sapere se ti funziona

• Rispondi • Condividi ›

Glody Fimpa • 3 anni fa

Dario, qual'è il plugin che usi in IntelliJ e se c'è una versione per Eclipse ?

• Rispondi • Condividi ›

Dario Frongillo admin ➚ Glody Fimpa • 3 anni fa

Di Dario Frongillo
13 Dicembre 2017

5 Comments

I NOSTRI PARTNER

NEEDFORNERD

Privacy Policy