





INTRODUZIONE

Ciao Coders,

anche oggi vi parliamo di pattern creazionali dopo l'ottimo riscontro che avete dato su [Builder Pattern](#) e [Factory Method Pattern.](#)

In quasi tutti i progetti ci scontriamo sulla necessità di dover avere una singola istanza di una data classe condivisa da diverse classi. Alcuni esempi sono uno spooler di stampa il quale deve tenere una coda unica, il manager di alcune risorse, una cache , il logger ecc. Il singleton pattern è una possibile soluzione a questa problematica.



1. Il singleton Pattern ha lo scopo di assicurarsi che una classe abbia solo un'istanza e fornire un punto di accesso globale a essa.

Con il termine Singleton si indica quindi una modalità di dichiarazione delle classi che ha lo scopo di garantire la creazione di una sola istanza della classe. La creazione e l'accesso a questa istanza è gestito direttamente dalla classe dichiarata in modalità Singleton.

Gli elementi che caratterizzano un Singleton sono:

- il costruttore privato, per evitare la creazione di oggetti da classi esterne;
- un metodo statico, per accedere all'unica istanza dell'oggetto.

Per chiarirci l'idea riportiamo un esempio di implementazione riportando il caso dello Spooler di una stampante.

IMPLEMENTAZIONE

```
1. public class PrinterSpooler {
```

```
8.     }
9.     return instance;
10.    }
11.   }
12. }
```



Il codice è molto semplice: la classe **PrinterSpooler**

permette l'istanziazione di un unico oggetto, tramite l'invocazione del metodo `getInstance` e non del costruttore il quale è dichiarato volutamente privato. Il metodo `getInstance` si fa quindi carico di creare l'unica istanza dell'oggetto; le successive invocazioni di `getInstance` restituiscono il riferimento allo stesso oggetto. Tale metodo è stato dichiarato `synchronized` per garantire il corretto utilizzo contemporaneamente da più thread. La soluzione mostrata ha ripercussioni sulla performance, perché tutte le invocazioni di `getInstance()` sono rallentate dall'overhead del `synchronized`; se ci pensate è uno spreco perché la race condition può accadere solo sul primo utilizzo del metodo `getInstance`; in letteratura è nota una seconda versione del singleton la quale al costo minimo di un doppio controllo effettuato quando l'istanza non è stata creata, evita che le successive chiamate del `getInstance()` incorrano nell'overhead di sincronizzazione.

```
1. public class PrinterSpooler {
2.     private static volatile PrinterSpooler instance;
3.     private PrinterSpooler() {
```

```
10.         }
11.     }
12. }
13.     }
14.     return instance;
15. }
16. }
17. }
```



CONSIDERAZIONI

Il pattern appena descritto potrebbe sembrare semplicissimo; in effetti lo è ma bisogna considerare i seguenti aspetti e situazioni in cui non è affatto semplice progettare una classe singleton:

- Presenza di Singletons in multiple virtual machines.
- Singletons caricati contemporaneamente da diversi class loaders.
- Singletons distrutti dal garbage collector, e dopo ricaricati quando sono necessari.
- Presenza di multiple istanze come sottoclassi di un Singleton.
- Copia di Singletons come risultato di un doppio processo di deserializzazione.

Ora che abbiamo descritto il singleton e una sua possibile implementazione, non voglio soffermarmi a parlare di possibili varianti e implementazioni di tale pattern (esistono molti libri che già lo fanno) ma voglio parlarvi dei casi di utilizzo e di problemi che potreste scontrarvi quando si abusa del singleton.



LE CRITICHE SULL'UTILIZZO DEL SINGLETON

L'uso di singleton è in realtà un argomento abbastanza controverso nella comunità OOP; quello che una volta era uno schema di progettazione spesso utilizzato viene ora considerato come una pratica da evitare: addirittura un anti-pattern. Le critiche più forti che vengono fatte al singleton sono le seguenti:

1. Il problema principale con il Singleton Pattern è che viene frainteso e spesso utilizzato per introdurre il concetto di variabili globali nel proprio sistema introducendo nell'applicazione lo stato globale nel dominio dell'applicazione. Questo è generalmente negativo perché le variabili globali non si preoccupano della struttura del



singleton in cui ci imbattiamo sono incompatibili con i principi generali di progettazione dell'ingegneria del software. Spesso fungono da aggregati per funzionalità diverse (a volte anche non correlate), introducendo in tal modo varie dipendenze e spesso violando concetti fondamentali come quello di responsabilità singola (single responsibility principle, abbreviato con SRP): ovvero che ogni elemento di un programma (classe, metodo, variabile) debba avere una sola responsabilità, e che tale responsabilità debba essere interamente encapsulata dall'elemento stesso. Infatti una delle abilità uniche di un singleton è che è possibile accedervi ovunque tramite il suo metodo statico disponibile globalmente, consentendo ai programmatore di utilizzarlo all'interno di un metodo senza doverlo passare espressamente attraverso i parametri. Anche se questo può sembrare più semplice per il programmatore, fare affidamento su questa istanza statica significa che le firme dei metodi non mostrano più le loro dipendenze, perché il metodo potrebbe estrarre un singleton "dal nulla". Ciò significa che gli utenti hanno bisogno di conoscere le logiche interne del codice per usarlo correttamente, rendendolo più difficile da usare e testare.



QUINDI ANTI-PATTERN O PATTERN ?

Il lettore a questo punto dell'articolo si farà sicuramente la seguente domanda: siamo di fronte ad un Pattern o Anti-Pattern? La domanda non è semplice, provo a dare il mio pensiero. Io sono d'accordo al 1001 per cento della gravità dei problemi descritti nel paragrafo precedente. La combinazione di questi problemi mostra un altro problema, a mio avviso il più grave: nel mondo odierno di sviluppo orientato ai test e agile, è più importante che mai avere piccoli test che coprono la maggior parte del codice. Una delle cose importanti di questi test è che devono essere in grado di essere eseguiti in qualsiasi ordine (non dipendenti l'uno dall'altro), il che può diventare un problema con l'utilizzo di singleton. Poiché le dipendenze di alcuni metodi dati non sono chiare quando si basano su singleton (ricavandole da getter statici), un tester può non sapere scrivere due test che effettivamente dipendono l'uno dall'altro modificando una risorsa condivisa (il singleton). Questo può produrre test di infima qualità, quelli che passano quando

E' vero però che tutti questi problemi sono conseguenze di un **ABUSO** del singleton; il singleton non è nato per modellare uno stato globale ma bensì per contendere istanze uniche di risorse sharabili nel proprio applicativo: mi viene in mente una coda di stampa, il logger, la connessione al db o verso un altro tipo di dispositivo. Capite quindi che i casi corretti di utilizzo del singleton sono ben pochi ma questo non deve significare che tale pattern non è utile poiché la sua casistica di utilizzo esiste. E' certo però che l'utilizzo al di fuori di queste casistiche diventa un vero e proprio anti-pattern.

A mio avviso esistono altri pattern e tecniche per risolvere casi di cattivo utilizzo del singleton senza cadere nelle problematiche descritte da questo articolo: la Dependency Injection in primis (di cui io abuso) o il Factory. Vi parlerò di questi pattern nei prossimi articoli.

design java OOP pattern singleton



A PROPOSITO DI ME

▲
Torna su



DARIO FRONGILLO



Fondatore di Italiancoders e dell'iniziativa devtalks.
Software architect con un forte background in Java,
Architettura REST, framework Spring , Database
Design, progettazione e sviluppo di SPA e RIA Web
application con framework Javascript. Attualmente
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



Many failed before. Will you complete the trial?

Hero Wars



Nei bilanci salute e risultati: Axa attiva iniziative per garantire l'equilibrio psico-fisico dei 1800 addetti

La Repubblica

Prima di incontrare persone, si raccomanda l'uso di questo prodotto antivirale scientificamente approvato

Agi

Le razze di cani più costose al mondo

Il Mondo dei Cani

Siediti prima di vedere la villa dove vive Maria De Filippi

RetroPages

Il reale patrimonio netto di Valeria Marini ci sciocca profondamente

2 Commenti **ItalianCoders** **normativa sulla privacy** **Accedi** ▾

Consiglia 1

Tweet

Condividi

Ordina dal più recente ▾



Partecipa alla discussione...

ENTRA CON

O REGISTRATI SU DISQUS

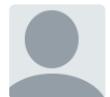
Nome

sia dimenticato un synchronized di troppo:

```
public static synchronized PrinterSpooler getInstance() {  
    if ( instance==null) {  
        synchronized (PrinterSpooler.class) {
```

Altrimenti andresti solo a peggiorare le performance con un blocco sincronizzato in più in fase di prima inizializzazione.

^ | v • Rispondi • Condividi ›



Finta Tartaruga • 3 anni fa

Penso che la frase più importante di tutto l'articolo sia

"E' vero però che tutti questi problemi sono conseguenze di un ABUSO del singleton;"

Nella mia esperienza personale non mi sono trovato spesso ad usare singleton, ma ci sono effettivamente alcune situazioni in cui l'uso del singleton è molto naturale. Tipicamente quando è naturale avere una parte del codice che "offre servizi" al resto del codice, quali appunto code di stampa o servizi di logging. Si osservi anche che in casi come questi non ha molto senso parlare di ereditarietà (cosa dovrei derivare da un servizio di logging?)

Io programmo principalmente in Ada. In questo linguaggio il modo più naturale di implementare un singleton, talmente naturale che non viene neanche visto come qualcosa legato all'OOP: si usa un package che esporta i suoi servizi attraverso funzioni/procedure.



I NOSTRI PARTNER



[Privacy Policy](#)