5. Singleton

(GoF pag. 117)

5.1. Descrizione

Specifica i tipi di oggetti a creare, utilizzando un'istanza prototipa, e crea nuove istanze tramite la copia di questo prototipo.

5.2. Esempio

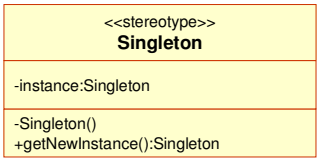
Un applicativo deve istanziare un oggetto che gestisce una stampante. Questo oggetto deve essere unico, vale dire, deve esserci soltanto una sola istanza di esso, altrimenti, potrebbero risultare dei problemi nella gestione della risorsa.

Il problema è la definizione di una classe che garantisca la creazione di un'unica istanza all'interno del programma.

5.3. Descrizione della soluzione offerta dal pattern

Il "*Singleton*" pattern definisce una classe della quale è possibile la istanziazione di un unico oggetto, tramite l'invocazione a un metodo della classe, incaricato della produzione degli oggetti. Le diverse richieste di istanziazione, comportano la restituzione di un riferimento allo stesso oggetto.

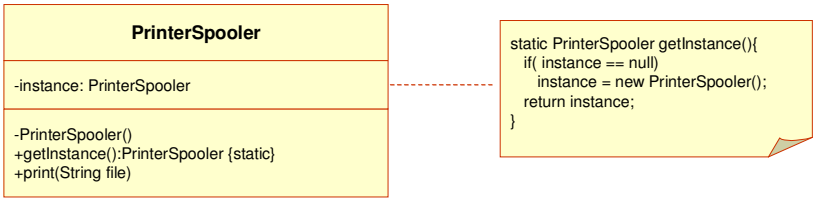
5.4. Struttura del pattern



5.5. Applicazione del modello

Schema del modello

Si presenta di seguito una delle implementazioni più semplici del Singleton:



Partecipanti

- **Singleton**: classe PrinterSpooler.
  - Definisce un metodo getInstance che restituisce un riferimento alla unica istanza di se stessa.
  - E' responsabile della creazione della propria unica istanza.

Descrizione dl codice

Si propongono adesso tre diverse implementazioni del *Singleton* pattern in Java. Si discutono le caratteristiche di ognuna di esse.

a) Singleton come classe statica

E' il modello più semplice, ma che in realtà non è un vero e proprio **Singleton**, perché soltanto si lavora con una classe statica, non un oggetto [14], [17]. Questa classe statica ha metodi statici che offrono i servizi richiesti.

```
public static class PrinterSpooler {

  private PrinterSpooler() {

   }

  public static void print (String msg) {
     System.out.println( msg );
  }

}
```

Si noti che il costruttore è dichiarato privato, per evitare l'istanziazione di oggetti della classe.

L'invocazione all'oggetto sarà una istruzione simile a:

```
PrinterSpooler.print( somethingToPrint );
```

Questa versione di Singleton è chiamato *Booch utility*, perché è stato Grady Booch a identificarlo [8]. Uno dei problemi principali che ha questa versione è la necessità di conoscere a tempo di caricamento delle classi tutta l'informazione necessaria per creare il **Singleton**. Un altro problema è l'impossibilità di implementare interfacce tramite questo approccio.

b) Singleton creato da un metodo statico

Il **Singleton** è implementato come una classe che ha un metodo statico (`getInstance`) che deve essere chiamato per restituire l'istanza del **Singleton**. L'oggetto **Singleton** verrà istanziato solo la prima volta che il metodo sia invocato, in modo che eventuali informazioni necessarie per creare il **Singleton** possano essere fornite in tempo di esecuzione. Le veci successive sarà restituito un riferimento allo stesso oggetto.

```
public class PrinterSpooler {

 private static PrinterSpooler instance;

   private PrinterSpooler() {
```

```
    }

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

  public void print (String msg) {
     System.out.println( msg );
  }

}
```

In questa implementazione il costruttore continua ad essere statico, per costringere all'utilizzo del metodo `getInstance()`, per ricavare gli oggetti:

```
...
PrinterSpooler theUnique = PrinterSpooler.getInstance();
theUnique.print( somethingToPrint );
//if we try this:
PrinterSpooler mayBeOther = PrinterSpooler.getInstance();
//then...
if( theUnique != mayBeOther )   → false!!!
...
```

c) Singleton *multi thread*

La implementazione precedente non va bene nel caso di una esecuzione *multithread*. Si pensi al caso nel quale un *thread* fa l'invocazione, per prima volta, al metodo `getInstance` e dopo di testare se l'istanza è già stata creata con l'istruzione `if( instance == null)`, viene sospeso da un altro *thread* che invoca lo stesso metodo. Dato che il **Singleton** non è ancora creato, questo secondo *thread* potrebbe crearlo. Dopo, quando il controllo è restituito al primo *thread*, questo crea una nuova istanza di Singleton.

La soluzione e sincronizzare il metodo `getInstance`:

```
public class PrinterSpooler {

 private static PrinterSpooler instance;

   private PrinterSpooler() {

   }

   public static synchronized PrinterSpooler getInstance() {
      if ( instance==null) {
         instance = new PrinterSpooler();
      }
      return instance;
   }


  public void print (String msg) {
     System.out.println( msg );
  }

}
```

La soluzione presentata può essere considerata inefficiente perché ogni volta che si fa una invocazione al metodo `getInstance` deve essere

acquisito un *lock*. Una soluzione scorretta in Java è la suggerita da Doug Schmidt e presentata da Holub [8]. Questa soluzione utilizza il lock soltanto se l'istanza non è già stata creata (strategia chiamata "*double-checked locking*"):

```
public class PrinterSpooler {

 private static PrinterSpooler instance;

   private PrinterSpooler() {

   }

   public static PrinterSpooler getInstance() {
      if ( instance == null) {
             synchronized( PrinterSpooler.class ) {
                    if( instance == null )
                           instance = new PrinterSpooler();
             }
      }
      return instance;
   }


  public void print (String msg) {
     System.out.println( msg );
  }

}
```

Bacon et. al. [1] spiegano come i criteri di ottimizzazione, al momento di implementare la Java Virtual Machine, possono far generare più di una istanza al programma presentato.

Osservazioni sull'esempio

Si è voluto presentare un esempio banale, che serve a esemplificare diverse implementazioni di **Singleton**. Data la semplicità dell'esempio, non si presenta l'esecuzione di un *main program*, come nei casi dei pattern precedenti.

5.6. Osservazioni sull'implementazione in Java

Altre considerazioni da studiare al momento di proporre l'implementazione di un *Singeton* sono descritti da Fox [5]. Aspetti da tenere in conto sono:

- 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.