International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)
Web Site: www.ijettcs.org Email: editor@ijettcs.org
Volume 5, Issue 6, November - December 2016 **ISSN 2278-6856**

The Issues in Singleton Design Pattern

Madonna Lamin

[1]Department of Computer Science and Engineering, ITM Universe, Vadodara

Abstract

To speed up software development process there have been immense contribution by one and all. Out of the many the discovery of Design Patterns, which are standard solutions to problems in object-oriented software engineering has proven fruitful in the software design unit. These patterns guarantee better quality software and at the same time a novice designer feels at home when it comes to understanding the system functionality. Till date there are many design patterns that solves specific software design problems. One of which is the Singleton Pattern which deals with the object creation mechanism. This paper highlights the various issues in this pattern and throws light on the solution to the same for further refinement and proper usage.

Keywords- Design Pattern, Singleton

1. INTRODUCTION

Design Patterns are a way of implementing a common solution to a common problem in object-oriented software [1]. They describe the proven solutions to recurring problems [2]. One of the advantages of using Design Patterns is that novice developers find it easy to develop the software as they are now provided with a ready-made solution which they can implement without any worries as it has been used earlier and proved to be efficient in similar situations [3].

They are categorized into Creational, Behavioral and Structural Patterns[4]. The creational patterns include Factory patterns and Singleton pattern. The behavioral includes The Strategy pattern, The Template Method pattern, The Observer pattern, The MVC pattern, The Command pattern and Chain of Responsibility pattern. The structural patterns are The Composite pattern, The Adapter pattern, The Decorator pattern, The Façade pattern and The Flyweight pattern.

The various advantages of Design Pattern are- they provide ready-made solutions to various problems and hence speed up the development process; they capture the knowledge and expertise of many professionals and hence increase the quality of software being made and since they are revised and improved over time they are more reliable than the patterns prepared by the developer himself which might have some problems.
This paper discusses and focuses on the various issues encountered in the usage of Singleton Pattern and probable solutions to the same.

2. THE SINGLETON PATTERN

The Singleton Pattern which is a class of creational design pattern deals with ways to create instances of classes. In many cases, the exact nature of how the object is created could vary with the needs of the program. The best practice is to abstract the object initialization process into a class. The Singleton Pattern assures that there is one and only one instance of a class created, and provides a global point of access to it.

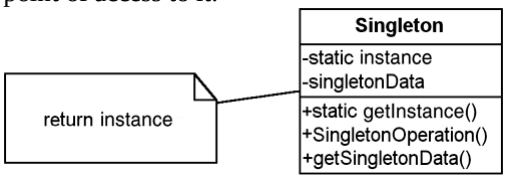

Figure 1 Standard, Simplified View of the Singleton Pattern

A simple example of Singleton class is shown below:

```
public class ClassicSingleton {
  private static ClassicSingleton instance = null;
  protected ClassicSingleton() {
    // Exists only to defeat instantiation.
  }
  public static ClassicSingleton getInstance() {
    if(instance == null) {
      instance = new ClassicSingleton();
    }
    return instance;
  }
}
```

There are several interesting points concerning the ClassicSingleton class. First, the ClassicSingleton employs a technique known as *lazy instantiation* to create the Singleton and as a result, the Singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that the singleton instances are created only when needed.

There are various issues in the Singleton pattern. Some of them are enlisted and describe with probable solutions to tackle:

Issue 1- If we create a Singleton class and create a subclass of that then the subclass constructor is call implicitly through super().

```
public class myClasss extends ClassicSingleton{
        myClass(){
          super();
          //because of this its create superclass
          //object over here.
          //so its kill Singleton philosophy which does not
        work here
          }
}
```

The solution is to create a Singleton class as *final*. Hence no subclass for that class can be created.

Issue 2- The constructor is not protected.
If the constructor is protected or public then a class, which is in the same package, can create object of Singleton class which violates the Singleton criteria.

The solution for preventing the creation of the new object of Singleton class is to create a private constructor in this class. If the constructor is declared private then no class can create the object of Singleton class except the class itself. In addition the subclass also cannot create since in the subclass the constructor which calls the super method cannot find the constructor of the super class.

Another solution is to create a default constructor and no other class lands in the same package. Hence no class can create the object of Singleton class because its constructor is default.

Issue 3 - ClassicSingleton.getInstance () method is not thread-safe because of the following code:

```
1: if(instance == null) {
2:    instance = new Singleton();
3: }
```

If a thread is preempted at Line 2 before the assignment is made, the instance member variable will still be null, and another thread can subsequently enter the if block. In that case, two distinct singleton instances will be created. Unfortunately, that scenario rarely occurs and is therefore difficult to produce during testing.

The solution is to use double locking system to solve multi threaded issue.

```
public static Singleton getInstance() {

  if(singleton == null) {
    synchronized(Singleton.class) {
      singleton = new Singleton();
    }
  }
  return singleton;
}
```

The method also can be synchronized but it affects the performance since when one thread obtains the lock of that object and at the same time no another thread use this method this affects the performance.

Therefore, double locking is a better option. What happens if two threads simultaneously access getInstance ()? Imagine Thread 1 enters the synchronized block and is preempted. Subsequently, a second thread enters the if block. When Thread 1 exits the synchronized block, Thread 2 makes a second check to see if the Singleton instance is still null. Since Thread 1 set the Singleton member variable, Thread 2's second check will fail, and a second Singleton will not be created. Or so it seems.

Issue 4 - Because multiple classloaders are commonly used in many situations—including servlet containers—we can end up with multiple singleton instances no matter how carefully we have implemented the Singleton classes. To ensure that the same classloader loads the Singletons, we must specify the classloader ourselves, for example;

```
private static Class getClass(String classname)
              throws ClassNotFoundException {
    ClassLoader classLoader=
      Thread.currentThread().getContextClassLoader();
       if(classLoader== null)
         classLoader= Singleton.class.getClassLoader();
       return (classLoader.loadClass(classname));
     }
   }
```

The preceding method attempts to associate the classloader with the current thread. If that classloader is null, the method uses the same classloader that loaded a Singleton base class. The preceding method can be used in place of Class.forName () method.

Issue 5 - If we serialize a Singleton and then deserialize it twice, there will be two instances of the singleton, unless we implemented the readResolve () method as shown below:

```
import org.apache.log4j.Logger;
public class Singleton implements java.io.Serializable {
  public static Singleton INSTANCE= new Singleton();
  protected Singleton(){
    //Exists only to thwart instantiation.
```

```
  }
    private Object readResolve() {
       return INSTANCE;
    }
}
```

The previous Singleton implementation returns the lone Singleton instance from the readResolve() method, therefore, whenever the Singleton class is de-serialized, it will return the same Singleton instance.

7 APPLICATIONS OF SINGLETON PATTERN

Hardware interface access- Practically the Singleton pattern can be used in case of external hardware resource usage is limited. For example, hardware printers where the print spooler can be made a Singleton in order to avoid multiple concurrent accesses and creating deadlock.

Logger- A Singleton is a good mechanism to use in the log files generation. Let's say we have an application where the logging utility has to produce one log file based on the messages received from the users. If there exists multiple client applications that are using this logging utility class there is a chance that they might create multiple instances of the same class. This may lead to certain issues during concurrent access to the same logger file. Thus, it is advisable to use the logger utility class as a Singleton and provide a global point of reference.

Configuration File- The Singleton pattern suits best for this as it prevents multiple users to repeatedly access and read the configuration file or properties file.

Cache – The cache is a potential Singleton object as it can have a global point of reference for all future calls to the cache object that the client application will be using in the in-memory object.

8.Conclusion

The five issues discussed facilitate the software designer to be aware and avoid the pitfalls and loopholes well before a full-fledged product is developed. Proper and strict adherence to the pattern ensures the focal point that it is meant to obtain.

References

[1] Alex Blewitt, Alan Bundy, Ian Stark, "Automatic Verification of Design Patterns in Java", School of Informatics, University of Edinburgh, UK
[2] Taibi, T., Ling,D.N.C.,"Formal specification of design patterns, a comparison", Computer Systems and Applications, 2003.Book of Abstracts, ACS/IEEE International Conference on 14-18 July 2003.
[3] Harpreet Singh,"Term Paper on Design Patterns", Advanced Topics in Software Engineering,CSC 532
[4] Andy C.H.Law,"Introducing Design Patterns in First Year Object-Oriented Programming",Kwatlen University College.
[5] Alan Shaloway, James R.Trott,"Design Patterns Explained",Addison Wesley Publication,ISBN-0-201-71594-5
[6] Javed Mulla,"Singleton Pattern", Article in LinkedIn, July 15, 2015.
[7] [Online] Available: http://en.wikipedia.org/wiki/Singleton_pattern. (General Internet Site)

AUTHOR

Madonna Lamin, Assistant Professor Computer Science and Engineering, received the B.E. degree in Computer Science and Engineering from The M.S. University of Baroda in 2000 and M.E. degree in Computer Science and Engineering from Gujarat Technological University in 2013. Her area of interest are Design Thinking, Agile Methodology.