Singleton Pattern Design

Objetivo y Aplicación

Este patrón creacional recibe su nombre debido a **que sólo se puede tener una única instancia para toda la aplicación de una determinada clase**, esto se logra restringiendo la libre creación de instancias de esta clase mediante el operador *new* e imponiendo un constructor privado y un método estático para poder obtener la instancia.

La intención de este patrón es garantizar que **solamente pueda existir una única instancia de una determinada clase** y que exista una referencia global en toda la aplicación, por lo tanto solo es aplicable en este tipo de proyectos.

Se intenta resolver la instanciación excesiva de objetos de una misma clase, idénticos entre sí, en distintos puntos de nuestro software

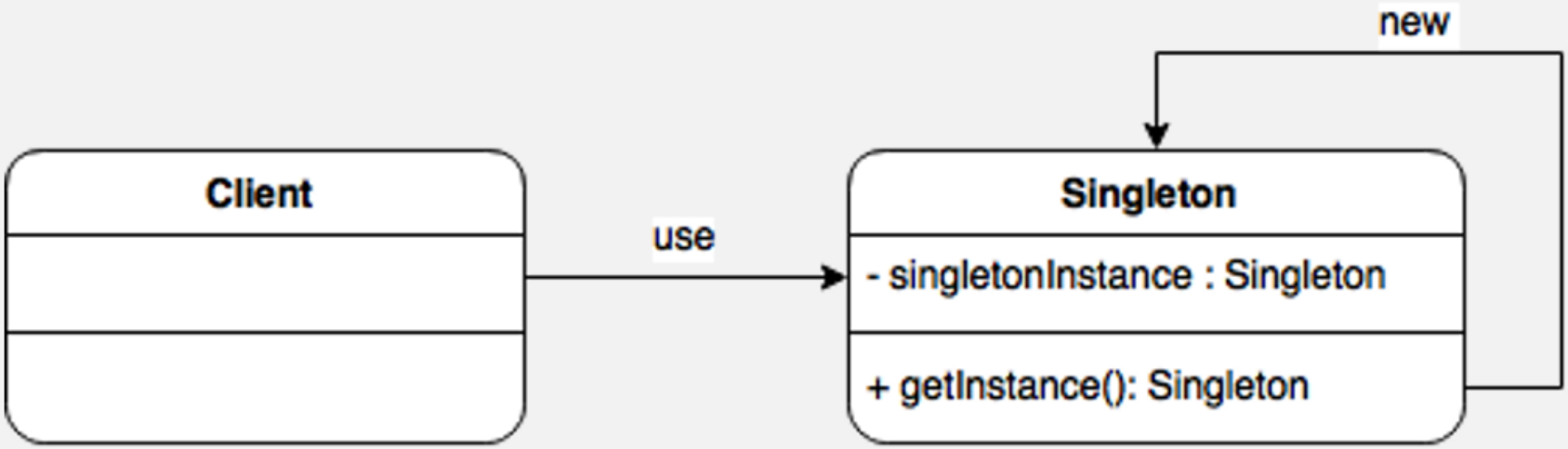
Estructura

Los componentes que conforman el patrón son los siguientes:

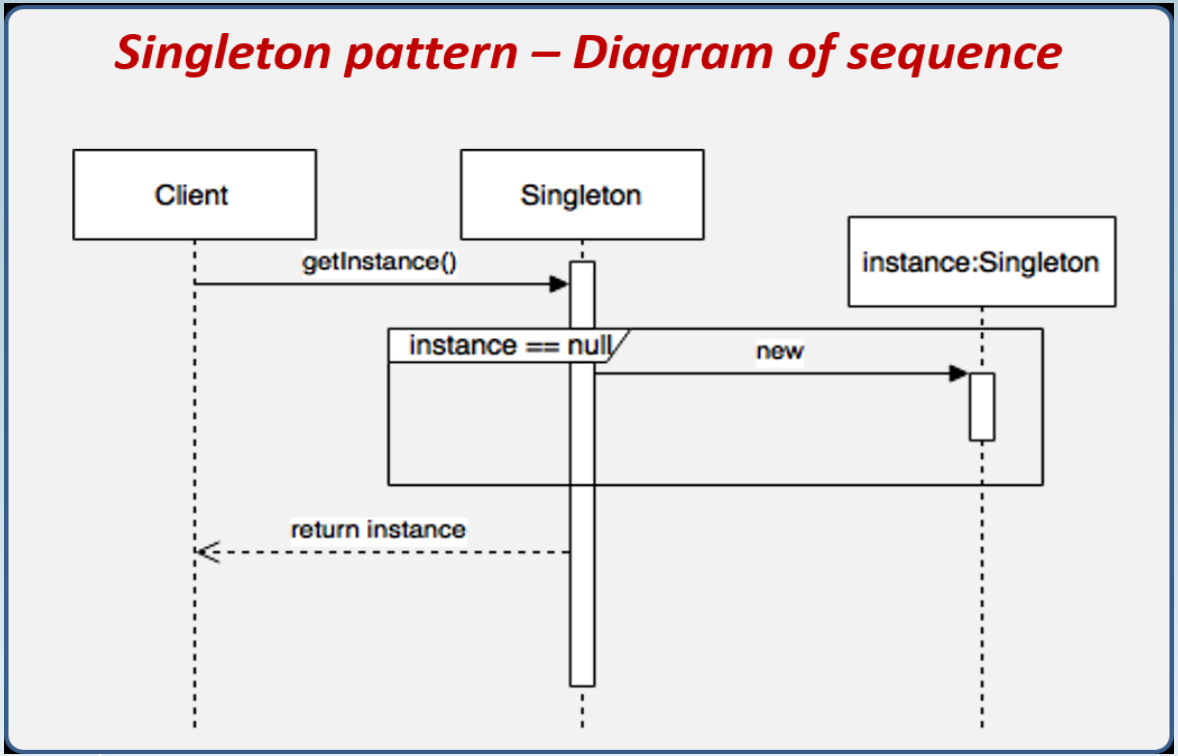
Client:
Componente que desea obtener una instancia de la clase Singleton.

Singleton:
Clase que implementa el patrón Singleton, de la cual únicamente se podrá tener una instancia durante toda la vida de la aplicación.

Singleton pattern – Class diagram



Singleton pattern – Diagram of sequence



1.El cliente solicita la instancia al Singleton mediante el método estático getInstance.

2.El Singleton validará si la instancia ya fue creada anteriormente, de no haber sido creada entonces se crea una nueva.

3.Se regresa la instancia creada en el paso anterior o se regresa la instancia existente en otro caso.

Consecuencias

Ventajas

• **Reduce el espacio de nombres.**
El patrón es una mejora sobre las variables globales. Ya no se reservan nombres para las variables globales, ahora solo existen instancias

• **El control de acceso a la instancia única es** ***controlado***, porque la clase Singleton encapsula la única instancia.

•Permite el **refinamiento** de las operaciones y la representación.

•Permite un numero variable de instancias. El patrón es **fácilmente configurable** para permitir más de una instancia.

•**Flexible** en cuanto a las operaciones contenidas en una clase

Desventajas

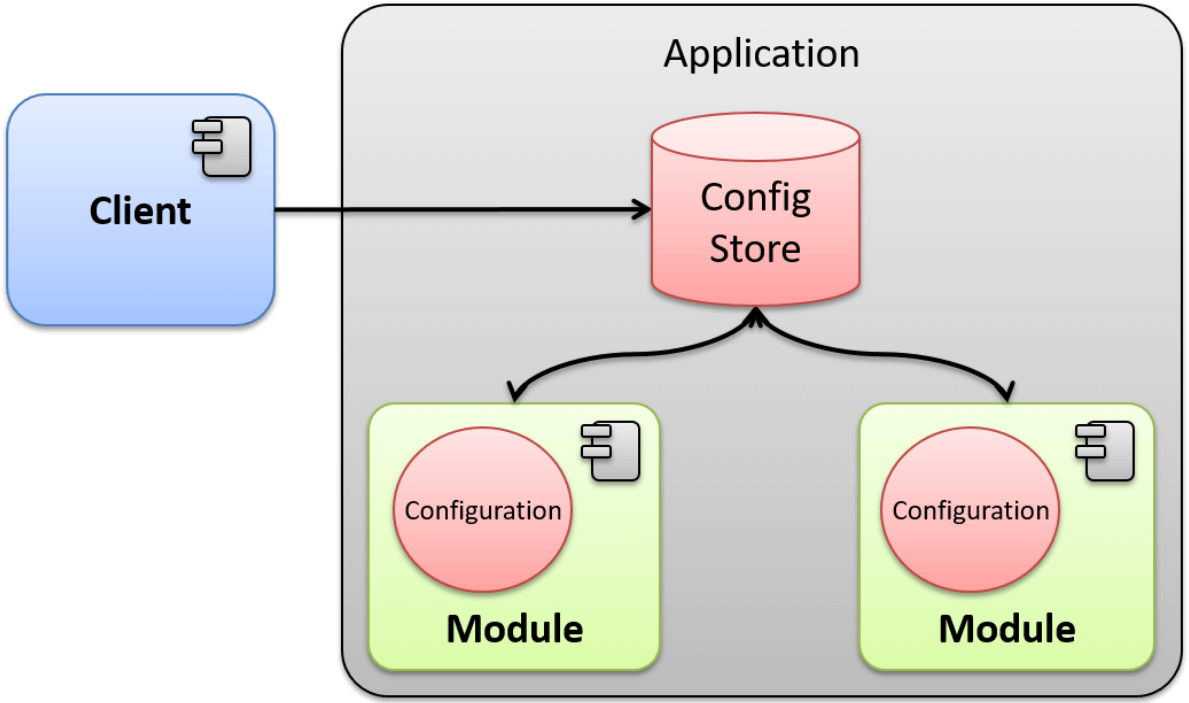
•Opera de forma **síncrona**, lo cuál puede ser perjudicial si nuestro software requiere de concurrencia de llamados a métodos de este objeto.

•Viéndolo de forma genérica, el mayor enemigo de este patrón es el **acceso de forma concurrente** a este objeto singleton, problema que de hecho no existiría si no se aplicara el patrón.

•Corresponde a un **recurso compartido** dentro de nuestro software, por lo que debemos tener especial cuidado en el manejo de su estado, ya que si un segmento de nuestro software altera su estado, esto puede afectar el funcionamiento de otros segmentos de software que utilicen a este objeto.

Ejemplos

Ejemplo



Mediante la implementación del patrón de diseño Singleton crearemos una aplicación que permite gestionar la configuración del sistema desde un único punto centralizado.
Así, cuando la aplicación inicie, cargara la configuración inicial y está disponible para toda la aplicación.

```java
// Java program implementing Singleton class
// with getInstance() method
class Singleton {
  // static variable single_instance of type Singleton
  private static Singleton single_instance = null;

  // variable of type String
  public String s;

  // private constructor restricted to this class itself
  private Singleton() {
    s = "Hello I am a string part of Singleton class";
  }

  // static method to create instance of Singleton class
  public static Singleton getInstance() {
    if (single_instance == null)
      single_instance = new Singleton();
    return single_instance;
  }
}
```

```java
// Driver Class
class Main {
  public static void main(String args[]) {
    // instantiating Singleton class with variable x
    Singleton x = Singleton.getInstance();

    // instantiating Singleton class with variable y
    Singleton y = Singleton.getInstance();

    // instantiating Singleton class with variable z
    Singleton z = Singleton.getInstance();

    // changing variable of instance x
    x.s = (x.s).toUpperCase();

    System.out.println("String from x is " + x.s);
    System.out.println("String from y is " + y.s);
    System.out.println("String from z is " + z.s);
    System.out.println("\n");

    // changing variable of instance z
    z.s = (z.s).toLowerCase();

    System.out.println("String from x is " + x.s);
    System.out.println("String from y is " + y.s);
    System.out.println("String from z is " + z.s);
  }
}
```

Output:

```
String from x is HELLO I AM A STRING PART OF SINGLETON CLASS
String from y is HELLO I AM A STRING PART OF SINGLETON CLASS
String from z is HELLO I AM A STRING PART OF SINGLETON CLASS


String from x is hello i am a string part of singleton class
String from y is hello i am a string part of singleton class
String from z is hello i am a string part of singleton class
```

x ---------->
y ---------->

z ---------->

```
private static Singleton single_instance = null;

public String s;

---------------------------------------------------

private Singleton();

public static Singleton getInstance();
```

Explicación:

En la clase Singleton, cuando llamamos por primera vez al método getInstance (), crea un objeto de la clase con el nombre single_instance y lo devuelve a la variable. Dado que single_instance es estático, se cambia de nulo a algún objeto. La próxima vez, si intentamos llamar al método getInstance (), dado que single_instance no es nulo,
se devuelve a la variable, en lugar de crear instancias de la clase Singleton nuevamente. Esta parte se realiza por si condición.

En la clase principal, instanciamos la clase singleton con 3 objetos x, y, z llamando al método estático getInstance (). Pero en realidad después de la creación del objeto x, las variables y y z apuntan al objeto x como se muestra en el diagrama.
Por lo tanto, si cambiamos las variables del objeto x, eso se refleja cuando accedemos a las variables de los objetos y y z. Además, si cambiamos las variables del objeto z, eso se refleja cuando accedemos a las variables de los objetos x e y.