

TC3003: Diseño y Arquitectura de Software

Dr. Juan Manuel González Calleros

Email: jmgonzale@itesm.mx

Twitter: [@Juan_Gonzalez](https://twitter.com/Juan_Gonzalez)

Facebook: [Juan Glez Calleros](https://facebook.com/JuanGlezCalleros)

Reuniones pedir cita



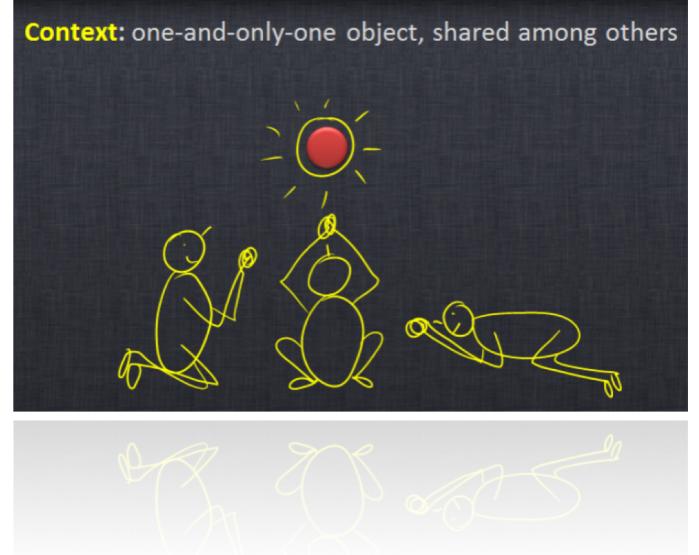
Singleton Pattern

One of a Kind Objects(Objetos únicos)

El Patrón Singleton es el más simple de todos en términos de su diagrama de clases, ya que su diagrama ¡Sólo tiene una clase!

-¿De qué sirve eso?

Hay muchos objetos de los que sólo necesitamos uno de su tipo, por ejemplo: grupos de subprocessos, cachés, cuadros de diálogo, objetos que manejan configuraciones de registro, objetos utilizados para iniciar sesión y objetos que actúan como controladores de dispositivos(impresoras, tarjetas gráficas).



-¿ Y no podría simplemente hacerlo por convención o variables globales?

El Patrón Singleton es un método probado para asegurar que sólo se cree un objeto y sólo uno para una clase determinada.

Nos da un punto de acceso global, igual que una variable global pero sin las desventajas.

-¿Cuáles desventajas?

Por ejemplo: Si tu asignas un objeto a una variable global, tienes que crear ese objeto desde el inicio de tu aplicación(depende de la implementación). ¿Qué pasaría si ese objeto consume muchos recursos y tu aplicación nunca termina usándolo?

Pues con el Patrón Singleton podemos crear nuestros objetos sólo cuando son necesarios.

Y por muy simple que suene, algo que debes preguntarte es: ¿Cómo puedo prevenir que más de un objeto sea instanciado?

¿Cómo se crea un objeto?	<code>new MyObject();</code>
¿Qué pasaría si otro objeto quisiera crear un MyObject?	No habría ningún problema.
Siempre que tengamos una clase, ¿podemos instanciarla una o más veces?	Sí, siempre y cuando sea una clase pública.
¿Y si no lo es?	Sólo las clases en el mismo paquete pueden instanciarla. Pero aún pueden instanciarla más de una vez.

-¿Qué significa lo siguiente?

```
public MyClass {  
    private MyClass() {}  
}
```

Es una clase que no puede ser instanciada porque tiene su constructor privado.

-¿Hay algún objeto que pueda utilizar el constructor privado?

Se entendería que el código que está en MyClass sería el único que podría llamarlo, pero esto no tendría mucho sentido.

-¿Por qué no tiene sentido esto?

```
public MyClass {  
    private MyClass() {}  
}
```

Porque se tendría que tener una instancia de la clase para llamarla, pero no se puede tener una instancia porque ninguna otra clase puede instanciarla.

Puedo usar el constructor de un objeto de tipo MyClass, pero nunca puedo crear una instancia de ese objeto porque nadie más puede usar “new MyClass()”.

-¿Qué hay de hacer esto?

```
public MyClass {  
    public static MyClass getInstance() {  
    }  
}
```

MyClass es una clase con un método estático. Se puede llamar al método estático de la siguiente manera:

```
MyClass.getInstance();
```

-¿Ahora ya se puede instanciar a MyClass?

```
public MyClass {  
    private MyClass() {}  
  
    public static MyClass getInstance() {  
        return new MyClass();  
    }  
}
```

Diseccionando la implementación del Patrón Singleton clásico

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

La fábrica de chocolate

Las fábricas modernas tienen calderas de chocolate con controlador por computadora. El trabajo de la caldera es tomar chocolate y leche, llevarlos a ebullición y luego pasarlos a la siguiente fase de hacer barras de chocolate.

En el siguiente código notará que han tratado de tener mucho cuidado para asegurarse de que no sucedan errores, cómo drenar 500 galones de mezcla sin hervir o llenar la caldera cuando ya está llena, o hervir es una caldera vacía



```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

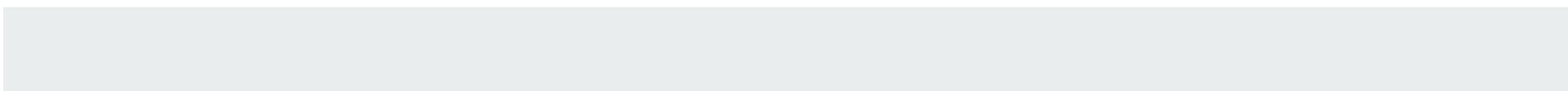
    public boolean isBoiled() {
        return boiled;
    }
}
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.

To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.



Choc-O-Holic, ha hecho un buen trabajo para garantizar que no ocurran cosas malas,, ¿No crees?

Por otra parte, probablemente sospeche que si se pierden dos instancias de chocolate, pueden ocurrir algunas cosas malas

¿Cómo podrían salir las cosas si se crea más de una instancia de chocolate en una aplicación?

¿Puedes ayudar a Choc-O-Holic a mejorar su clase ChocolateBoiler convirtiéndola en Singleton?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
      
    ChocolateBoiler() {
        empty = true;
        boiled = false;
    }
      
    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

Puntos a destacar del patrón Singleton

Comencemos con la definición concisa del patrón: El patrón Singleton garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global.

¿Qué está pasando realmente aquí? estamos tomando una clase y permitiéndole administrar una sola instancia de sí misma. También estamos evitando que cualquier otra clase cree una nueva instancia por sí misma. Para obtener una instancia, debes pasar por la clase misma

También proporcionamos un punto de acceso global a la instancia: cada vez que necesite una instancia, solo consulte la clase y le devolverá la instancia única. Como has visto, podemos implementar esto para que el Singleton se cree de manera sencilla, lo cual es especialmente importante para el objeto de uso intensivo de recursos

Okay, let's check out the class diagram:

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

Singleton
static uniqueInstance
// Other useful Singleton data...
static getInstance()
// Other useful Singleton methods...

The uniqueInstance class variable holds our one and only instance of Singleton.

↑ A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Lidiando con sucesos múltiples (Multithreading)

Muchos de los problemas multihilos son resueltos de manera casi trivial convirtiendo métodos getInstance() a sincronizados.

Sin embargo, es algo muy “costoso” en términos de ejecución.

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.



¿Cómo se mejora esto?

En la mayoría de los trabajos en Java, es necesario asegurarnos que el patrón Singleton siga presente en nuestro trabajo. Pero si el uso de una sincronización parece muy pesado, hay algunas opciones para corregir y mejorar los métodos.

1.- No hacer nada si el rendimiento del getInstance() no es crítico para la aplicación.

En la mayoría de los trabajos en Java, es necesario asegurarnos que el patrón Singleton siga presente en nuestro trabajo. Pero si el uso de una sincronización parece muy pesado, hay algunas opciones para corregir y mejorar los métodos.

2.- Mover el método de una instancia pasiva a una activa.

```
public class Singleton {  
    private static Singleton uniqueInstance = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return uniqueInstance;  
    }  
}
```



Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!



We've already got an instance, so just return it.

3.- Hacer un chequeo doble para reducir el uso de sincronización en getInstance()

Lo que se realiza en esta opción es verificar la creación de una instancia, si la hay, se hace la sincronización.

```
public class Singleton {  
    private volatile *static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

Check for an instance and
if there isn't one, enter a
synchronized block.

Note we only synchronize
the first time through!

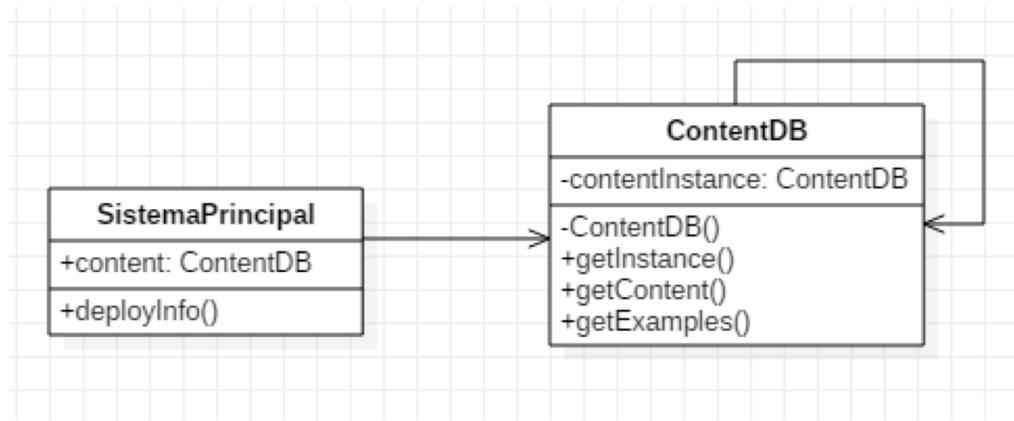
Once in the block, check again and
if still null, create an instance.

* The volatile keyword ensures that multiple threads
handle the uniqueInstance variable correctly when it
is being initialized to the Singleton instance.

Otra aplicación del patrón de diseño

La empresa InternationalPMI posee un sistema para conocer y aprender de una manera interactiva el PMBOK, su sistema principal realiza funciones como mostrar el contenido y ejemplos. Su problema es que quieren trasladar toda la información a una BD la cual va a contener métodos como la conexión y la recuperación del contenido con sus ejemplos. Mientras tanto el sistema principal es responsable de crear las funciones respectivas para tomar esa información y mostrarla.

Diagrama de clases - InternationalPMI





Tarea

APLICAR ESTE PATRÓN A TU PROYECTO FINAL.