Reflexión sobre el Patrón de Diseño Singleton

Autores: - Joao Castillo — 25776 - Kenett Ortega — 25777

Repositorio: https://github.com/bcastillo-2022474/uvg-estructuras-hoja-4

¿Qué es el patrón Singleton?

El patrón Singleton garantiza que una clase tenga **una única instancia** durante toda la ejecución del programa, y proporciona un punto de acceso global a ella. Se implementa con un constructor privado y un método estático `getInstance()` que crea la instancia solo la primera vez que se invoca.

```
public class Calculator {
    private static Calculator instance;

    private Calculator() {}

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

Ventajas del patrón Singleton

1. **Instancia única garantizada:** Es imposible crear accidentalmente múltiples instancias de la clase, lo que evita inconsistencias de estado cuando el objeto administra un recurso compartido.
2. **Punto de acceso global controlado:** A diferencia de una variable global ordinaria, el Singleton encapsula su instancia dentro de la propia clase. El acceso siempre pasa por `getInstance()`, lo que permite agregar lógica de inicialización, validaciones o logging en ese punto.
3. **Inicialización diferida (lazy initialization):** La instancia se crea solo cuando se necesita por primera vez, no al arrancar el programa. Esto puede ahorrar recursos si el objeto es costoso de construir y no siempre se utiliza.
4. **Facilita la coordinación:** Cuando múltiples partes del sistema necesitan acceder al mismo objeto (por ejemplo, un logger, una conexión a base de datos, o un calculador), el Singleton evita tener que pasar la instancia de módulo en módulo.

Desventajas del patrón Singleton

1. **Comportamiento similar a una variable global:** Al ser accesible desde cualquier punto del código mediante `Calculator.getInstance()`, introduce un estado global implícito. Esto hace que las dependencias entre clases sean difíciles de detectar, ya que no aparecen en las firmas de los métodos ni en los constructores.
2. **Dificulta las pruebas unitarias:** Un Singleton retiene su estado entre pruebas. Si una prueba modifica el estado de la instancia, puede afectar a las pruebas siguientes de formas inesperadas. Para mitigarlo se requieren mecanismos de reset o el uso de mocks, lo que agrega complejidad.
3. **Viola el principio de responsabilidad única:** La clase asume dos responsabilidades: su lógica de negocio *y* el control de su propio ciclo de vida (cuándo y cómo se crea).
4. **Problemas en entornos multihilo:** La implementación básica no es thread-safe. Si dos hilos llaman a `getInstance()` simultáneamente antes de que la instancia exista, pueden crear dos instancias. Se necesita sincronización (`synchronized`) o una variante como *double-checked locking* para resolverlo.
5. **Acoplamiento fuerte:** El código que invoca `Calculator.getInstance()` queda acoplado directamente a esa clase concreta, lo que dificulta sustituirla por una implementación diferente (por ejemplo, para pruebas o extensión del sistema).

¿Es adecuado su uso en este programa?

El uso del Singleton para la clase `Calculator` en este programa es **aceptable dentro del contexto académico**, pero presenta limitaciones que vale la pena reconocer.

A favor

- La `Calculator` no mantiene estado entre evaluaciones: cada llamada a `operate()` recibe su propia pila como parámetro y trabaja de forma independiente. Esto elimina el principal riesgo del Singleton (estado compartido contaminado), ya que el objeto es efectivamente sin estado.
- En este programa de un solo hilo, los problemas de concurrencia no aplican.
- Tiene sentido conceptualmente que exista una sola "calculadora" en el sistema.

En contra

- Dado que `Calculator` no tiene estado, el Singleton no aporta un beneficio real: crear múltiples instancias sería igualmente correcto y más fácil de probar.
- La restricción de instancia única es artificial aquí. El patrón se justifica mejor en objetos que administran recursos costosos o compartidos (conexiones de red, archivos de configuración, pools de hilos), lo cual no es el caso de un evaluador de expresiones.
- El acoplamiento que introduce complica la extensión futura del sistema (por ejemplo, una calculadora con historial o configuración distinta por usuario).

Conclusión

El Singleton es adecuado cuando realmente se necesita controlar el acceso a un recurso único y compartido. En este programa cumple el requisito del enunciado y funciona correctamente, pero si el proyecto creciera, sería preferible usar **inyección de dependencias**: recibir la instancia de `Calculator` como parámetro en lugar de acceder a ella globalmente. Esto haría el código más modular, testeable y flexible sin sacrificar ninguna funcionalidad.