Taller 5 - Singleton

David Burgos - 201818326

Wyo Hann Chu Méndez - 202015066

Juan Daniel Sepúlveda - 202113067

Link del repositorio: https://github.com/thevarunjain/elevator-system

**Información del patrón**

El patrón singleton se caracteriza por permitir una sola instancia de una clase, esto con el objetivo de que no se malgaste la memora teniendo diferentes instancias de una clase que hacen el mismo proceso. Usualmente se controla esta única inicialización con un método getClass() y con un método constructor privado, el método getClass() revisa internamente si ya se creó la instancia de la clase y en el caso que no la encuentre la inicializa y la retorna, por otro lado, el método constructor privado hace que la misma clase sea la única que tenga control de si se inicializa o no.

Un ejemplo de este patrón lo podemos encontrar en un loggin de una aplicación, si en un proyecto existe una clase que se encargue de guardar la información del usuario no es necesario que se creen una instancia de esta clase en todos los elementos del proyecto, es suficiente con que se inicialice una vez y que cada clase la llame y la utilice de la misma forma. Otra situación en donde es común ver este patrón es en las interacciones con redes, a veces varias clases necesitan crear una request o cargar una imagen, pero el que cada una tenga su propia clase que maneje esta interacción puede generar un gran costo en memoria por lo que todas comparten la misma instancia de la clase.

Además, Singleton es considerado por muchos un anti patrón, puesto que, para hacer uso del patrón Singleton es necesario el hacer uso de estados globales en la aplicación (variables globales) que son generalmente innecesarias. Además, viola el principio de responsabilidad singular, ya que, la clase no solamente se encarga de sí misma si no también compromete el comportamiento de otras clases.

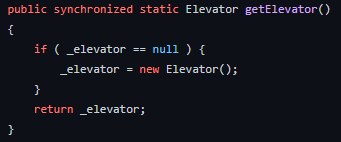
**Información del Proyecto**

Este proyecto representa un elevador que recorre diferentes pisos de un edifico, este puede subir o bajar pisos según el input del usuario, este comportamiento se logra mediante las 3 clases principales.

* La clase Elevator (en la que encontraremos el patrón Singleton) guarda la información como el piso actual y su dirección, también contiene 2 PriorityQueue las cuales guardan las solicitudes para subir o bajar de piso.
* La clase ElevatorHandler añade las diferentes paradas del elevador a las PriorityQueue según la dirección del mismo.
* La clase ElevatorServer se encarga del movimiento del elevador, esta tarea incluye: revisar si hay una nueva parada, cambiar la dirección en que se está moviendo el elevador y cambiar el piso en que se encuentra

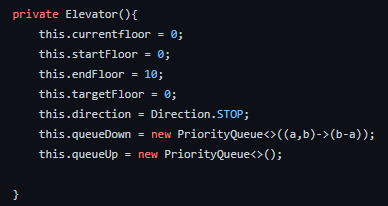
**Análisis del patrón en el proyecto**

Inicialmente, podemos revisar la clase Elevator, esta solo se va a crear una vez y va a retornar esta misma instancia, esto lo podemos ver en el método getElevator()



En este método se revisa la creación en el ( \_elevator == null ), en el caso de que no exista se crea, pero en el caso que ya exista se retorna el que ya existe.

Otro elemento de este patrón lo podemos encontrar en el método constructor

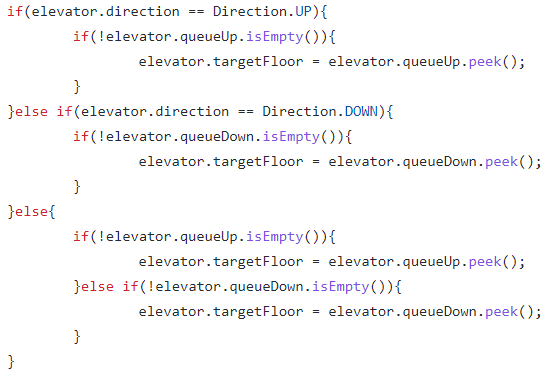


Primero, este método es privado para asegurarnos que solo existe una instancia la misma clase. Además, se añaden los valores iniciales de los atributos.

La función getElevator() es utilizada a lo largo del código para facilitar la comunicación entre las diferentes clases y el elevador, especialmente la podemos ver en la clase Main y ElevatorServer



Esta instancia se utiliza principalmente en la clase ElevatorServer para cambiar los elementos del elevator, un ejemplo es cuando se revisan las colas para cambiar el piso objetivo



**Análisis de la implementación**

Se utilizó este patrón para este punto del proyecto, ya que, todas las clases giran en torno al elevador y lo que este hace. Por lo tanto, es más fácil tener una clase que contenga a su mismo objeto, pues todas las clases que necesiten hacer uso de este objeto solamente tendrán que llamar a la función .get en vez de tener que generar nuevos elevadores para cada una de las clases.

No obstante, esto cuenta con un gran problema y es que viola en todo sentido el principio de responsabilidad singular, y es que todas las clases dependen de este objeto central conocido como “elevador”. En caso de que el programa fuera más grande y se necesitara modificar el objeto elevador para agregar nuevos métodos o funcionalidades, todas las clases tendrían que cambiar para que el programa funcione correctamente. Además, el hecho de que solo haya una misma instancia de la variable elevador hace que sea mucho más difícil correr pruebas como lo son las pruebas por unidad, debido a que, será necesario correr el constructor de la clase elevador para hacer pruebas en cualquier otra clase.

Para solucionar este problema que se genera al utilizar el patrón Singleton se pueden hacer varias cosas, una de las más fáciles es en el main crear un objeto privado de tipo elevador que contenga al elevador que todas las clases van a utilizar. Así, se pierde la necesidad de hacer el .get de todas las clases y solamente se utiliza en el constructor el objeto elevador que se genera en el main sin la necesidad de que todas las clases modifiquen una variable dentro de otra clase.