Diseño de Programación Orientada a Objetos

Laura Calderón - 202122045

Taller 5: Patrones

Patrón de GoF seleccionado:

- Singleton

Información general del patrón:

- El patrón de diseño Singleton es un patrón de creación que asegura que solo haya una instancia de una clase y proporciona un punto de acceso global a ella. Para el diseño de Singleton es importante que el constructor de la clase Singleton sea privado, además que la clase Singleton debe contener una instancia estática privada de sí misma, y, por último, la clase Singleton debe proporcionar un método estático de acceso que permita al usuario acceder a la única instancia de la clase.

Nombre del proyecto seleccionado:

- Singleton Logger Template

URL del proyecto seleccionado:

- https://github.com/estsetubal-pa-geral/SingletonLoggerTemplate.git

Información general del proyecto:

- El proyecto proporcionado tiene como objetivo implementar un Logger para registrar acciones relacionadas con un juego en un archivo. La finalidad es registrar el momento en que se llevan a cabo ciertas acciones, como el inicio del juego, la adición de un nuevo jugador y las jugadas realizadas. La estructura general del diseño se compone de tres clases principales:
    - Logger: Esta clase representa el logger en sí y se encarga de manejar la creación y escritura en el archivo de log. Implementa el patrón Singleton para garantizar una única instancia y proporcionar acceso global a esta clase.
    - Gammer: Esta clase representa un jugador en el juego y se encarga de almacenar su nombre y la cantidad de juegos realizados. En el constructor de la clase, se registra en el logger la creación de un nuevo jugador.
    - Game: Esta clase representa un juego y se encarga de iniciar el juego y realizar jugadas. Al iniciar el juego y realizar una jugada, se registra la acción correspondiente en el logger.
- En cuanto a los grandes retos de diseño que enfrenta el proyecto, considero que el más grande es la correcta implementación del patrón Singleton en la clase Logger, ya que debe garantizar que solo exista una única instancia del logger en todo el código y que esta instancia sea accesible globalmente. Adicional a esto, otro gran reto sería que el diseño del logger debe abordar de manera eficiente la gestión de recursos, como la apertura y cierre del archivo de log, y garantizar un buen rendimiento, especialmente si se espera un alto volumen de acciones de registro. Finalmente, otro gran reto que se me ocurre es que el diseño debe permitir una fácil mantenibilidad y extensibilidad del logger. Esto implica considerar cómo agregar nuevas funcionalidades y que el proyecto pueda adaptarse a posibles cambios y ampliaciones en el futuro.

-

Información y estructura del fragmento del proyecto donde aparece el patrón:

- En la clase Logger, el fragmento del proyecto en donde se evidencia claramente la utilización del patrón Singleton es:

```
public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;
```

Información del patrón aplicado al proyecto: explicar cómo se está utilizando el patrón dentro del proyecto.

- En este código, el método getInstance() verifica si la instancia de Logger ya ha sido creada (instance == null). Si la instancia no existe, se crea una nueva instancia de Logger. Si la instancia ya existe, se devuelve la instancia existente. Esto garantiza que solo haya una única instancia de Logger en todo el sistema y que se pueda acceder a ella a través de getInstance(), siguiendo así el patrón Singleton.

¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?

- En el proyecto tiene sentido utilizar el patrón Singleton en la clase Logger por varias razones. En primer lugar, al utilizar el patrón Singleton, se asegura que solo existe una única instancia de la clase Logger en todo el sistema. Esto es útil para mantener un único punto de registro de acciones del juego y evitar la creación de múltiples instancias del logger, lo que podría llevar a conflictos o incoherencias en el lado registro de las acciones. Por otro lado, el patrón Singleton proporciona un acceso global a la instancia del Logger, lo que facilita su uso en diferentes partes del código sin necesidad de pasar la instancia como parámetro. Todos los componentes del proyecto pueden acceder al registrador de manera sencilla y consistente a través del método getInstance(), lo que simplifica el registro de acciones en el juego. Además, al utilizar el patrón Singleton, se tiene un control centralizado sobre la creación de la instancia del Logger. Esto permite inicializar los recursos necesarios para el registro, como la conexión con el archivo de registro, en el constructor privado de la clase. De esta forma, también evita la creación necesaria de múltiples instancias, lo que puede ser importante en términos de eficiencia y uso de memoria. Por último, al utilizar el patrón Singleton, se proporciona una única entrada para acceder a la instancia del Logger. Esto facilita la posibilidad de extender o reemplazar el comportamiento del registrador en el futuro sin modificar el código en múltiples lugares. Por ejemplo, si se desea cambiar la implementación del archivo de registro o agregar funcionalidades adicionales al logger, solo se necesita modificar la clase Logger y su método getInstance().

¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

- La primera desventaja que de utilizar el patrón Singleton en el proyecto puede ser un enlace fuerte entre las clases que depende del Logger y la propia implementación del Singleton. Esto puede dificultar la modularidad y flexibilidad del código, ya que cualquier cambio en el Singleton puede tener un impacto en múltiples partes del proyecto. Asimismo, la dependencia de una única instancia global puede complicar las pruebas unitarias. Debido a que el Singleton es accesible desde diferentes partes del código, puede resultar difícil crear pruebas independientes que no dependan de su estado o comportamiento. Las pruebas unitarias pueden requerir configuraciones adicionales para simular el Singleton durante las pruebas. Ahora bien, el uso del patrón Singleton puede tener un impacto en el rendimiento si se utiliza incorrectamente. Por ejemplo, si la instancia del Logger se accede con frecuencia en un entorno con múltiples hilos de ejecución, puede generar problemas de concurrencia. Por último, si se necesita cambiar la implementación del Logger en el futuro, puede resultar difícil debido al mantenimiento de las clases que dependen del Singleton. La modificación de la implementación del Singleton puede requerir cambios en múltiples lugares del código, lo que puede ser costoso y probablemente a errores.

¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

- En el caso particular del proyecto otra forma de solucionar los problemas que resuelve el Singleton es utilizar la técnica de inyección de dependencias para proporcionar una instancia del Logger a las clases que lo necesiten. Esto implica pasar una instancia de Logger como parámetro en el constructor o a través de métodos setter de las clases que lo requieren. Además, se me ocurre que también, para reemplazar el patrón de Singleton se podría hacer uso de patrones de diseño específicos de registro. Existen otros patrones de diseño específicos para el registro de acciones y eventos, como el Observer o el Strategy pattern. Estos patrones permiten una mayor flexibilidad en la gestión y personalización del registro de acciones, permitiendo diferentes comportamientos según los requisitos específicos del proyecto.