**Nombre: Nicolás Miguel Murillo Cristancho**

**Código: 202213891**

**Diseño y Programación: Taller 5**

1. **Información del proyecto:**

El proyecto elegido es un videojuego programado en java, publicado en enero de 2023. El juego se ejecuta en una interfaz gráfica en la cual controlamos a un tigre que debe comerse a sus presas mientras evita ser tocado por otros animales. Específicamente, en un plano bidimensional, el videojuego permite modificar la posición del tigre y detectar la posición de las presas y depredadores, para saber cuándo sumar puntos o terminar el juego.

Este videojuego podría ser usado con fines educativos o de entretenimiento.

El proyecto se pude consultar en el siguiente link: <https://github.com/Kalzud/javasmallgame.git>

**1.1 Estructura del proyecto:**

El proyecto se divide en 6 paquetes

Aplication: Tiene la clase principal desde la cual se ejecuta el método main y se llaman a las demás clases

Gameobjects: Tiene los objetos que representan a los animales, una clase por cada animal

Factory: Clase encargada de crear los gameobjects

Gameassets: Contiene los recursos de sonido e imágenes que usa el juego

Strategy : contiene las clases que se ocupan de renderizar el juego

GameState: contiene las clases necesarias para iniciar los distintos estados posibles del juego (inicio, pausado, etc.)

**1.2 Retos de diseño que enfrenta el proyecto:**

Las relaciones entre las clases y las jerarquías de las mismas son un gran reto ya que hay una cantidad considerable de muchas clases con distintos propósitos, por lo tanto, se requirieron muchas clases que orienten el control del programa y dirijan a las demás.

La asignación de errores también fue un gran reto debido a la gran cantidad de responsabilidades distintas que se debían repartir

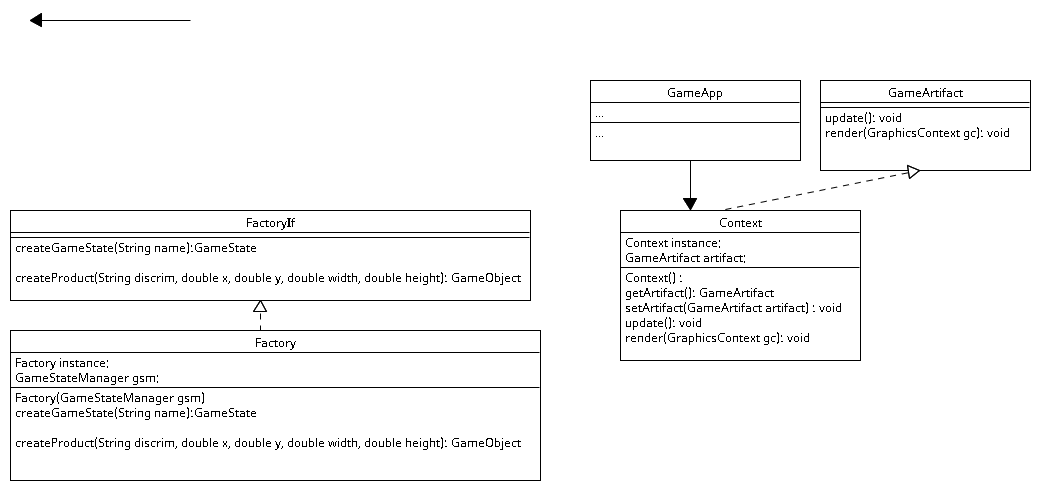
1. **Información y estructura del fragmento del proyecto donde aparece el patrón.**

**2.1 Factory**

En el paquete factory se tiene la interfaz FactoryIf, que se encarga de que las clases que la implementen (En este caso, solo Factory) tengan los métodos createGameState y createGameObject, estos dos métodos sirven para crear uno de los estasos del juego y un gameObject(instancia de un animal) respectivamente.

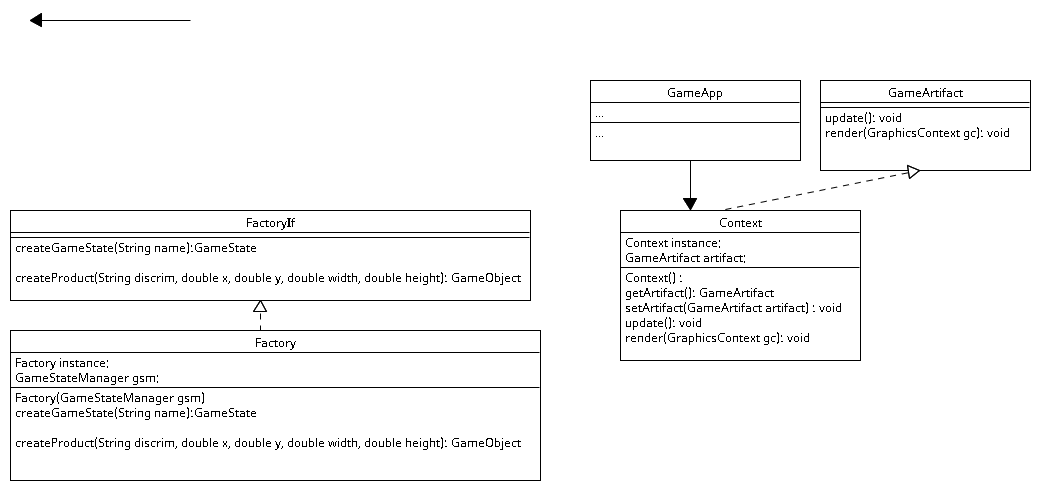
Como se mencionó, la clase factory implementa FactoryIf. Esta clase es el primer ejemplo del patrón singleton que encontramos en el código. Es la encargada de crear los animales, para los cuales recibe su nombre (así sabe a qué tipo de clase llamar para crear un tipo de animal específico) y sus dimensiones. También crea los estados de juego, recibiendo únicamente su nombre.

Solo se ocupa de crear y retornar este tipo de objetos, que deben ser usados por otras clases.



**2.2 Context**

Esta clase se ocupa de renderizar y actualizar el contenido gráfico en su respectivo estado de juego. Esta clase implementa la interfaz GameArtifact que se asegura de que quienes la implementen tengan los metodos de update() y render(), este último es de vital importancia para que el juego se visualice ya que recibe como parámetro un objeto del tipo Graphics context.



1. **Información general sobre el patrón Singleton:**

Este patrón se utiliza para garantizar que solo exista una instancia de una clase en un programa, o lo que es lo mismo, que su constructor solo sea usado una vez, y que en toda la aplicación solo exista un objeto de esta clase.

Normalmente este patrón es usado para aprovechar varias ventajas que pueden ser útiles si se adaptan al contexto del programa.

Por ejemplo, es ventajoso en casos donde se requiere un punto de acceso único a un recurso compartido, como una base de datos, un archivo de configuración o, en este caso, un creador de objetos. Pues proporciona un control centralizado sobre la instancia de la clase, lo que evita la creación accidental de múltiples instancias y garantiza que todas las demás partes del programa accedan a la misma instancia.

Otra ventaja importante es que el Singleton permite un uso eficiente de los recursos del sistema. Esto es especialmente útil en situaciones en las que la creación de múltiples instancias puede ser costosa en términos de memoria o tiempo de ejecución.

1. **Información del patrón aplicado al proyecto**

**4.1 Factory:**

En las clases que hacen uso de un patrón singleton es supremamente común (incluso necesario) contar con 2 características, la primera que su constructor se privado y no público, y la segunda, que tenga un método llamado getInstance (o un nombre similar) que funcione al mismo tiempo como constructor y como un “getter” y al mismo tiempo compruebe si ya existe una instancia para impedir que se cree más de una.

private static Factory *instance //la clase tiene un único objeto de su propia clase como atributo*

private Factory(GameStateManager gsm){ //el método constructor es privado

super();

this.gsm = gsm;

}

public static Factory getInstance(GameStateManager gsm) {

if(*instance* == null)

*instance* = new Factory(gsm);

return *instance*;

//Se asegura de entregar la única instancia de la clase que existe, y si no existe, la crea.

}

**4.2 Context**

Esta clase cumple exactamente con las mismas características mencionadas en la clase anterior.

private static Context *instance* = null;

private Context() {

super();

artifact = null;

}

public static Context getInstance() {

if(*instance* == null)

*instance* = new Context();

return *instance*;

}

1. **¿Por qué se usó el patrón Singleton en esta parte del proyecto?**

**5.1 Factory**

En cuanto a la clase Factory, no tendría ningún sentido crear varias instancias pues lo único que nos interesa de ella es crear objetos de otras clases, esta clase no tiene atributos significativos que pudieran variar de una instancia a otra. Es decir, si se crearan varias instancias todas las harían absolutamente lo mismo y no se distinguirían entre sí. Es por ello que se aplicó el patrón singleton y se restringió su constructor.

La principal ventaja se ve a nivel de memoria y optimización, pues se necesitan crear muchos gameObjects en repetidas ocasiones y en muchas partes del código. Si no se usara el patrón singleton se terminarían creando muchas factorías a lo largo de la ejecución, desperdiciando memoria y recursos.

**5.2 Context**

Es fundamental que todas las clases y métodos que utilizan a Context accedan a la misma instancia, ya que si por accidente se creara otro objeto de este tipo, que contenga información diferente o vacía, se perdería por completo la secuencialidad del programa,

Además, context se actualiza periódicamente por lo cual es necesario preservar toda su información, la cual se perdería si se llamaran a diferentes instancias.

1. **¿Qué desventajas tuvo usar el patrón Singleton en esta parte del proyecto?**

**6.1 Factory**

Es posible que la implementación de este patrón pueda ser confusa, sobre todo al inicio, ya que normalmente los constructores suelen ser públicos, el hecho de forzar a que solo se pueda crear una instancia de la clase requiere código adicional y aumenta las probabilidades de que ocurran errores.

Así mismo, desde otras clases se debe usar el método getInstance() en vez del constructor tradicional.

Más allá de ello, no hay desventajas evidentes. Suele ser común que en se implementen este tipo de patrones en circunstancias en las que, a largo plazo, es necesario tener más de una instancia de una clase, pero en este caso no ocurre ya que la clase cumple perfectamente su función.

**6.2 Context**

Al igual que en factory, la implementación puede ser más confusa, pero fuera de ello no representa desventajas significativas.

1. **¿De qué otras formas se pudieron haber solucionado los problemas que resuelve el patrón singleton?**

**7.1 Factory**

Una de las soluciones pudo haber sido simplemente eliminar la factoría, y, por consiguiente, no se desperdiciaría memoria en objetos de dicha clase. Se tomaría la decisión de crear los objetos consultando a cada clase de gameObject directamente. Sin embargo, esto hubiera desorganizado la jerarquía y aumentado excesivamente las relaciones entre clases. Lo más costoso de esta solución es que hubiera disminuido el acoplamiento, haciendo no solo que el código fuera menos legible si no que los errores fueran más costosos de corregir.

**7.2 Context**

Es difícil imaginar una solución que usara varias instancias de context, en el mejor de los casos, se tendría que copiar la información del anterior cada vez que se fuera a usar uno nuevo. Sin embargo, esto no representaría ningún beneficio, únicamente hubiera dificultado la implementación.