TALLER 5 – SINGLETON PATTERN.

1.ACERCA DEL PATRÓN

El patrón de diseño Singleton nos permite asegurarnos de que una clase tenga una única instancia y al mismo tiempo proporciona un punto de acceso global a dicha instancia. Asimismo, esta catalogada como patrón creacional, lo que concierne al proceso de creación de objetos.

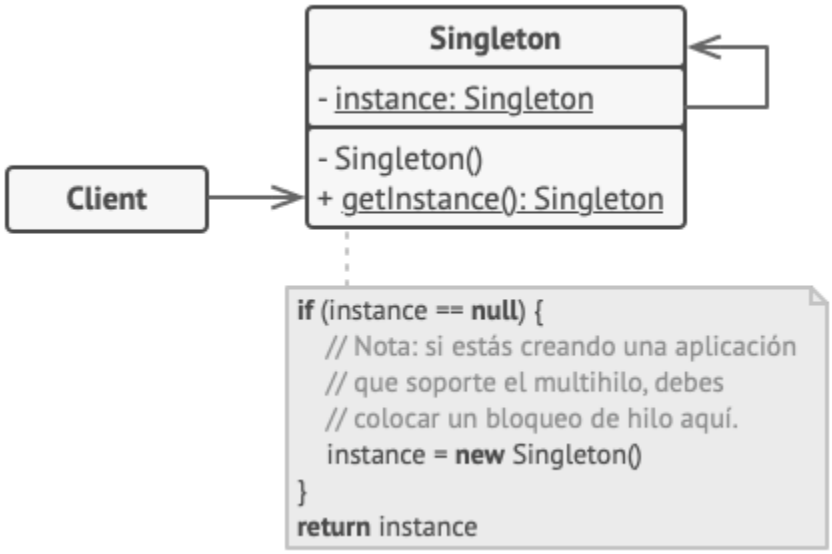
1.1 Aplicabilidad

El patrón Singleton resuelve dos problemas al mismo tiempo, violando el Principio de responsabilidad única: Garantizar que una clase tenga una única instancia y Proporcionar un punto de acceso global a dicha instancia. En el contexto de juegos, esto significa que el patrón Singleton puede garantizar que solo haya un juego y que todos los objetos del juego tengan acceso a él sin recurrir a variables o funciones globales.

1.2. Estructura

Se conforma de un contexto (Context), es decir el objeto principal que mantiene una instancia de cualquiera de los estados. También tiene una interfaz de estado (State), que permite encapsular los comportamientos e implementar los estados concretos (ConcreteStateA, ConcreteStateB)

Figura 1



Tomada de: https://refactoring.guru/es/design-patterns/singleton

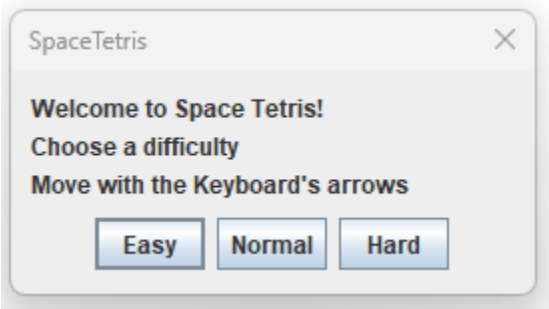
2. ACERCA DEL PROYECTO

Este proyecto es un juego que lleva por nombre *Space tetris*. El videojuego te convierte en una nave espacial la cual esta rondando por el espacio y una horda de asteroides se acercan a ti. Entre más asteroides puedas esquivar mayor será la distancia que recorras y por ende aumentará tu puntuación. El juego tiene 3 niveles de dificultad (fácil, medio y difícil) donde cada vez el espacio para que la nave cruce disminuye. El juego acaba cuando chocas con un asteroide.

De la misma manera, este juego hace uso de la librería swing para modelar los gráficos de la interfaz de usuario. Por último, cabe reconocer que Singleton no es el único patrón que utiliza este proyecto. El patrón iterator también se encuentra y se complementa con Singleton.

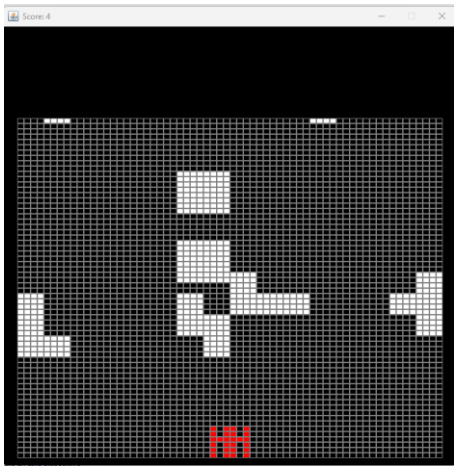
Enlace del repositorio: https://github.com/Gabelonio/Space-Tetris.git

Figura 2



Nota: *Acá se puede seleccionar el nivel de dificultad*

Figura 3



Nota: *vista de como se ve el juego.*

2.2 Grandes retos del Diseño

Analizando el proyecto se pueden identificar 3 grandes retos para los programadores a la hora de diseñar el proyecto. El primero es modelar cada uno de los asteroides, su forma, velocidad y frecuencia con la que salen. En segundo lugar, modelar cada uno de los movimientos de la nave. La posición, los controles y el tipo de dirección. Por último, está la creación de la interfaz gráfica. Donde se modelan las casillas, la consola y los threads (encargados de mostrar asteroides, movimientos del jugador y puntaje en el campo de visión del usuario).

2.2 Diseño

La estructura principal consta de dos actividades que se dividen en paquetes de origen, que muestran el juego y configuran su lógica: Los paquetes lógicos son "logicaAsteroide" y "logicaNave". El paquete gráfico es "grafica".

2.3. Paquete visualización

En este paquete hay:

- El modelo de la nave y la invocación de los asteroides.
- Una clase lógica breve ("Casilla") que determina la colisión y el espacio destinado a la nave o los asteroides.
- La configuración de la dificultad.
- La consola principal donde se ejecuta el juego.
- El movimiento de la nave y los asteroides.
- La lógica del puntaje.
- Casi todas las clases se gestionan con hilos (threads), excepto las clases "Casilla", "Consola" y "MainThread".

2.4 Paquete lógico

En los paquetes destinados al núcleo lógico del juego encontramos:

- La forma y rotación de los asteroides.
- El creador de los asteroides.

- La lógica de movimiento de la nave.
- Las coordenadas de la nave.

Comenzando con el paquete "logicaAsteroide”, tiene las siguientes especificaciones:

1. La forma de cada asteroide se gestiona con una matriz de valores booleanos.
2. La forma de cada asteroide se define mediante clases individuales.
3. La fábrica de asteroides se gestiona con un patrón Singleton.
4. La forma y rotación de los asteroides se generan aleatoriamente.

Continuando con el paquete "logicaNave", tiene las siguientes especificaciones:

1. La lógica de los movimientos se gestiona con un patrón Iterator.
2. La posición de la nave está compuesta por dos coordenadas: X e Y.
3. Se centra en dos clases de movimiento, positivo (derecha y abajo) y negativo (izquierda y arriba).

3. El patrón en el juego:

El uso del patrón Singleton dentro del juego Space tetris es bastante simple pero poderoso a la vez. Dentro del paquete lógicaAsteroide existe una clase llamada FabricaAasteroide que posee un solo atributo. Este Singleton se encarga de crear una única instancia de la clase fabrica.

3.1. Ventajas

- Garantizar una única instancia: El patrón Singleton asegura que solo haya una única instancia de la fábrica de asteroides en todo el juego. Esto puede ser importante para evitar la creación de múltiples fábricas y mantener el control sobre la generación de asteroides.

- Acceso global: Al utilizar un Singleton, se proporciona un punto de acceso global a la fábrica de asteroides desde cualquier parte del juego. Esto facilita la creación y gestión de los asteroides en diferentes componentes del juego, como la lógica de juego, la visualización y el control del jugador.

- Centralización de la lógica de creación: Al tener una única fábrica de asteroides, se centraliza la lógica de creación y se evita la dispersión de código relacionado con la generación de asteroides en diferentes partes del juego. Esto facilita el mantenimiento y la escalabilidad del juego a medida que se agregan nuevas características.

- Control sobre la generación de asteroides: Al utilizar el patrón Singleton, se tiene un mayor control sobre la generación de asteroides. Se pueden implementar restricciones adicionales, como limitar la cantidad máxima de asteroides generados o controlar la generación en momentos específicos del juego.

3.1. Desventajas

- Acoplamiento fuerte: El uso del patrón Singleton puede introducir un acoplamiento fuerte entre diferentes partes del código que dependen de la fábrica de asteroides. Esto puede dificultar la modificación o extensión de la lógica de generación de asteroides sin afectar otras partes del juego.

- Dificultad para realizar pruebas unitarias: Debido a que el patrón Singleton crea una instancia global accesible desde cualquier parte del juego, puede ser complicado realizar pruebas unitarias aisladas en componentes que dependen de la fábrica de asteroides. Esto se debe a que no se puede instanciar una versión alternativa de la fábrica durante las pruebas.

- Limitaciones en la concurrencia: Si el juego se ejecuta en un entorno con múltiples hilos o si se desea escalar la generación de asteroides en paralelo, el patrón Singleton puede generar problemas de concurrencia. Se deben aplicar mecanismos adicionales, como bloqueos o sincronización, para garantizar un acceso seguro y evitar condiciones de carrera.

- Dificultad para reemplazar la implementación: Si en algún momento se necesita reemplazar la lógica de generación de asteroides por una implementación diferente, cambiar la instancia única del Singleton puede ser complicado. Esto puede requerir modificaciones extensas en el código que depende del Singleton y puede afectar la estabilidad del juego.

- Dependencia oculta: El uso del patrón Singleton puede ocultar las dependencias y acoplamientos entre diferentes componentes del juego. Esto puede dificultar la comprensión y el mantenimiento del código a largo plazo, ya que las interacciones entre los componentes pueden no ser evidentes.

4. Solución Alternativa a Singleton:

El patrón Singleton es único en su capacidad para garantizar una única instancia de una clase. Sin embargo, en algunos casos, se pueden considerar otros patrones de diseño como alternativas al Singleton. Algunas posibles alternativas incluyen:

Patrón Monostate (Estado Monótono): Este patrón permite que múltiples objetos compartan el mismo estado interno, lo que puede proporcionar comportamiento similar a un Singleton sin la restricción de tener una única instancia de la clase.

Patrón Dependency Injection (Inyección de Dependencias): En lugar de tener una única instancia global, se pueden inyectar dependencias en las clases que las necesitan. Esto permite una mayor flexibilidad y facilita la prueba y la sustitución de componentes.

Patrón Registry (Registro): En lugar de tener una única instancia de una clase, se puede utilizar un registro centralizado para almacenar y acceder a las instancias de varias clases. Esto proporciona un punto centralizado de acceso, pero permite la creación de múltiples instancias.

4.1. Solución especifica

Un patrón de diseño que podría funcionar para modelar la fábrica de asteroides en lugar del patrón Singleton es el patrón Factory Method (Método de Fábrica). Debido a que proporciona una interfaz para crear objetos, pero delega la responsabilidad de la creación de objetos a las subclases, permitiendo así una mayor flexibilidad y extensibilidad.

Paso a paso

1. Definir una interfaz común para todos los tipos de asteroides que se pueden generar.
2. Crear una clase base abstracta, como "AsteroidFactory", que declare un método abstracto llamado "createAsteroid" para crear objetos de asteroides.
3. Implementar subclases de "AsteroidFactory" para cada tipo de asteroide específico. Cada subclase será responsable de crear y devolver una instancia del asteroide correspondiente.
4. En el juego, se utilizará una instancia de la subclase adecuada de "AsteroidFactory" para crear asteroides según sea necesario.

5. REFERENCIAS

Gamma, E., Helm, R., Johnson, R., Vlissides, J. M. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN: 0201633612

Refactoring Guru. (2023). State – Behavioral Patterns – Design Patterns. https://refactoring.guru/design-patterns/state