ICI2241 **Proyecto Videojuego (GM)**

Nombres: Julián Guerrero, Isidora Osorio y Martina Sandoval

Repositorio: Repositorio GitHub

(La rama más actualizada es Isidora)

GM1.1 Realizar una explicación del juego a realizar desde la perspectiva del jugador. Tutorial detallado de qué trata el juego (ej. objetivos, narrativa, elementos, y controles de movimiento/acción utilizando para ello maquetas o pantallazos de las interfaces gráficas.

Objetivo y explicación del juego

El objetivo del juego es eliminar todas las naves enemigas en cada nivel para avanzar. Cada nivel aumenta en dificultad, con más enemigos y algunos que requieren varios disparos para ser destruidos. Si pierdes tus tres vidas por los ataques enemigos, el juego termina y debes empezar de nuevo desde el primer nivel. (En el avance solo existe un nivel,

Nave del jugador

Nave enemiga

¿Cómo se juega?

- Para jugar es bastante simple, al iniciar el juego verás tu nave y las naves enemigas, por lo que debes moverte para que no te alcancen sus disparos. Para moverte deberás utilizar las flechas del computador.

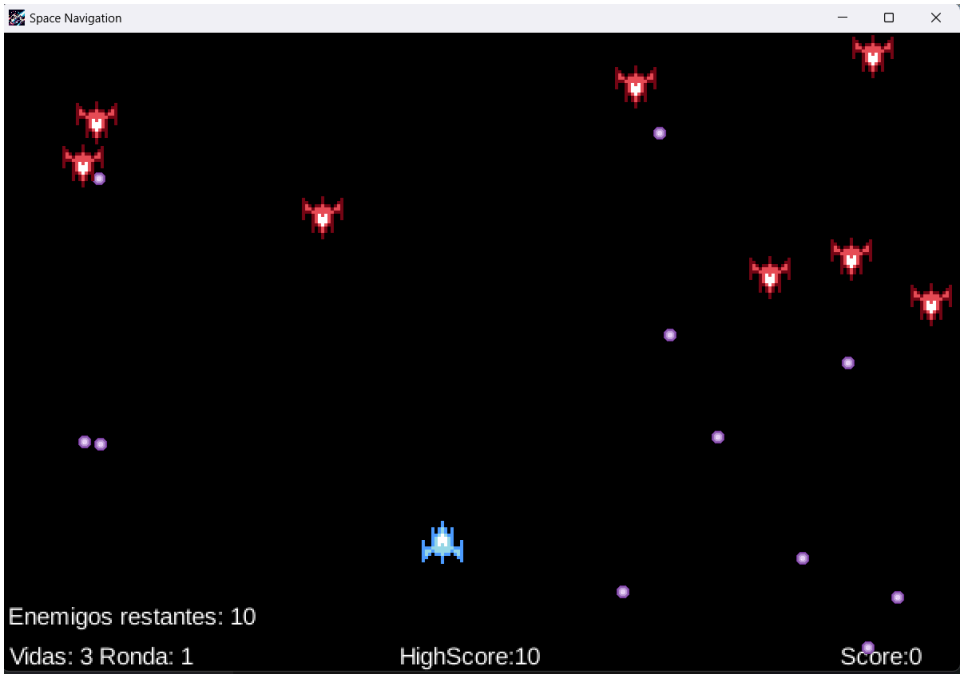
- Para eliminar las naves enemigas debes de dispararles, para disparar aprieta la tecla “espacio”. Esta tecla lanza una “bala” hacia arriba. Algunos enemigos necesitarán varios disparos para ser destruidos.Y para poder moverte debes usar las “flechas” del teclado.

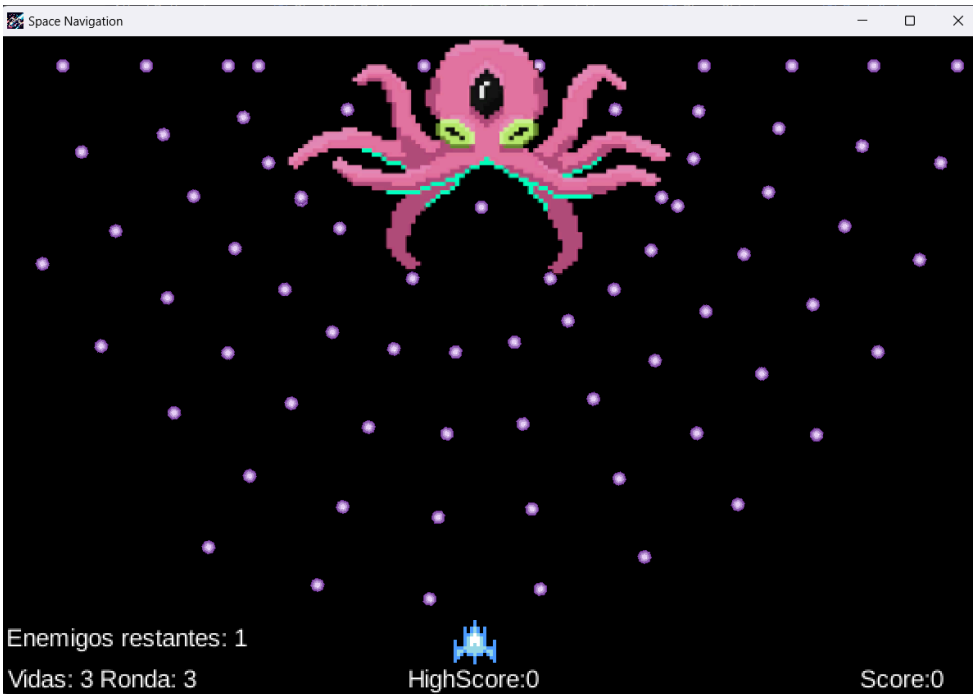
Elementos en pantalla

Al iniciar el juego se mostrarán las naves enemigas y tu propia nave. También en las esquinas de la pantalla se mostrarán el puntaje que llevas hasta el momento,

el puntaje más alto que hayas tenido, las vidas que te quedan, la ronda en la que te encuentras y la cantidad de enemigos que te quedan por derrotar.

También existe una pantalla de menú, que se muestra antes de comenzar el juego. Y otras pantallas que se muestran cuando eres derrotado (GameOver) o terminas el juego (Final).





GM1.2 Análisis del juego a realizar (perspectiva del Ing. de software), especificando el juego a usar de base y las modificaciones que se le pretende realizar en términos de funcionalidad y estructura.

El juego base que se escogió fue SpaceNav, este juego base se desarrolla en el espacio y el objetivo es destruir los asteroides que van rebotando por la pantalla. Este juego también está dividido en niveles, cada vez que avanzas a otro nivel la cantidad de asteroides aumenta.

Las principales modificaciones en cuanto a funcionalidad son:

- En las clases que tratan con las naves del juego, se agregó un cooldown para sus disparos, esto significa que entre disparos hay un pequeño espacio de tiempo, esto ayuda que el disparo sea más regularizado y no se pueda “spammear”.
- Las colisiones son entre el disparo del enemigo y la nave del enemigo.
- La nave del jugador va a poder lanzar un poder especial, el cual se debe recargar (esto aún no está implementado en el avance).

Las modificaciones en cuanto a estructura son:

- Se creó la clase BossShip, la cual corresponde al jefe final del juego, la diferencia con las naves del enemigo y la nave del jugador es que tiene distintos patrones de disparo.

- Existe la clase SpaceShip, que hereda a todas las naves (nave del jugador, enemiga y el jefe), esta es una clase abstracta.
- Se creó una interfaz que controla los patrones de ataque del jefe (AttackPattern).

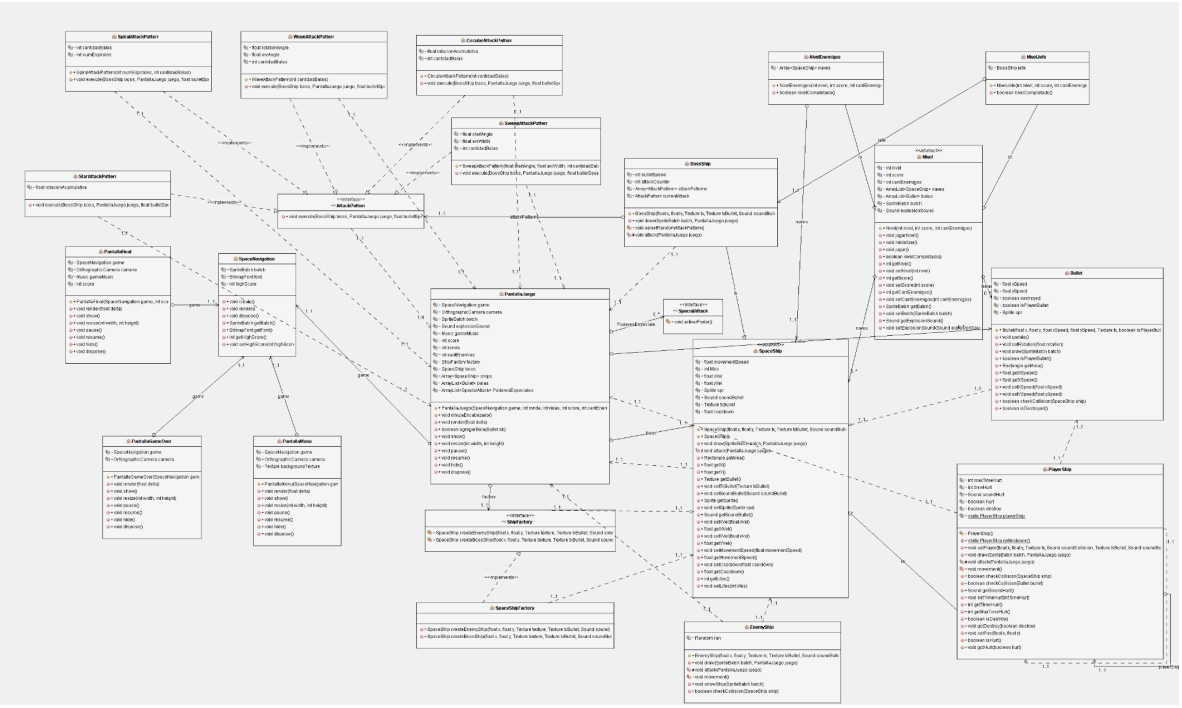
En este proyecto, todos los sprites del juego base fueron reemplazados, los disparos, las naves (con la inclusión del jefe) y se eliminó los asteroides. También se cambió la música de fondo a una pista “retro”, manteniendo sin cambios los efectos de sonido. Para integrar estos elementos, sustituimos directamente los archivos de imagen y sonido en la carpeta assets.

Estas modificaciones están relacionadas con la clase PantallaJuego, ya que en esta clase se inicializan las naves del jugador y los enemigos, así como la música y los efectos de sonido. La clase carga los archivos de assets y modifica los elementos en pantalla, aplicando los nuevos sprites y la música.

GM1.3 Diseño de diagrama UML con clases del Dominio del juego (clases base LibGDX + propias) y su código en Java (modificar)

Diagrama UML

(En el paquete, está el png para que se vea mejor)



GM1.4 Diseño y codificación de 1 (una) clase abstracta que sea padre de al menos 2 (dos) clases. La clase abstracta debe ser utilizada por alguna otra clase (contexto)

En el juego se utiliza la clase abstracta “SpaceShip” la cual hereda a las clases “PlayerShip”, “EnemyShip” y “BossShip”. Esta clase fue implementada de manera abstracta, ya que nos permite tener métodos concretos que se compartirán entre las clases sin necesidad de cambios, y también métodos abstractos los cuales serán utilizados por las clases que la extienden de manera distinta en cada caso, en este proyecto se utilizó especialmente, ya que cada nave tiene las mismas “bases” de movimiento, renderizado y ataque, pero implementadas de distinta forma, compartiendo atributos como vidas, velocidad de movimiento entre otras, además pudiendo añadir nuevos enemigos de bajo nivel de manera más rápida.

La clase abstracta “SpaceShip” consta de 8 atributos principales, los cuales son velocidad de movimiento, vidas, velocidad en el eje x, velocidad en el eje y, imagen de la nave, sonido de balas, textura de balas, cooldown, estos atributos son utilizados por las clases concretas para codificar el movimiento de las naves, la forma en la que atacan entre otras.

Esta clase cuenta con un constructor principal en el que se pasan por parámetro la posición en el eje x e y, las texturas de la nave, las texturas de las balas y el sonido de estas. Además de este constructor esta clase consta de 18 métodos de los cuales 2 son abstractos, con el resto de estos compuestos de setters y getters para los distintos parámetros.

Los métodos abstractos son “draw” y “attack” estos se encargan de mostrar la nave y del ataque de esta.

Para las clases concretas que extiende de esta tenemos:

PlayerShip

- Esta cuenta con 5 atributos adicionales a los de la clase abstracta, los cuales son tiempo máximo estando herido, tiempo estando herido, sonido cuando se hiere la nave, si está herida o no y por último si fue destruida. Además de estos atributos contamos con un constructor, que tiene los mismos parámetros que la clase abstracta más un atributo adicional que es el sonido cuando se hiere la nave. Para los métodos de esta clase contamos con las implementaciones de ambos métodos abstractos y además con una sobrecarga del método “checkCollision”, en la primera implementación de este método se tiene como parámetro una clase de tipo

“SpaceShip” y se encarga de revisar las colisiones entre la nave del jugador y las naves enemigas, también se encarga de modificar los parámetros adecuados cuando ocurre un choque como lo puede ser las vidas del jugador y si está herido, entre otras, para la segunda implementación del método tenemos como parámetro una clase de tipo “Bullet”, en esta implementación se revisan las colisiones entre balas enemigas y la nave del jugador, realizando modificaciones a los parámetros necesarios de la misma manera que la implementación anterior. Además de esta sobrecarga de métodos tenemos un último método concreto que se encarga del movimiento de la nave mediante la entrada del jugador, este se llama “movement”. Ahora el primer método abstracto que se implementa en esta clase es el de “draw” el cual se encarga del dibujado de la nave y del llamado de métodos como el “movement” y el “attack”, además se encarga de evitar que el jugador sobrepase los límites del área jugable, por último se encarga de posicionar al jugador y de evitar que este se mueva si este herido. El segundo método implementado es el de “attack” este se encarga del ataque por parte de la nave del jugador creando una bala, añadiendo sonido y verificando si el jugador puede volver a disparar pasado un tiempo, evitando que se dispare repetidamente en un periodo de tiempo corto.

EnemyShip

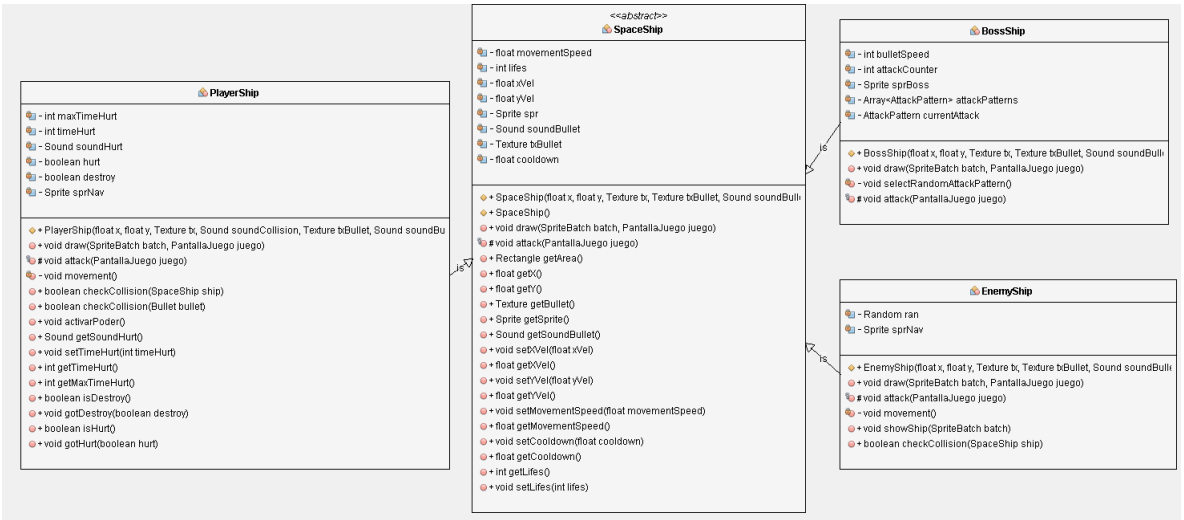
- Esta cuenta con un atributo adicional llamado “ran”, que se utiliza para obtener valores aleatorios, además de los atributos esta clase cuenta con un constructor que toma los mismos atributos que el constructor de la clase abstracta. En cuanto a métodos esta clase cuenta con 3 métodos adicionales a los métodos heredados, los cuales son “movement” este método se encarga del movimiento horizontal de los enemigos, y también se encarga de evitar que sobresalgan del área jugable, luego tenemos el método “showShip” el cual se encarga de dibujar/mostrar la nave enemiga este método se utiliza en la clase “PantallaJuego” para detener el movimiento de los enemigos cuando el jugador es herido. Finalizando los métodos concretos tenemos el método “checkCollision” el cual se encarga de revisar las colisiones entre naves y realizar un cambio de dirección cuando estas colisionan, produciendo así un rebote. Para la implementación de “draw” que es la primera clase abstracta, tenemos que esta se encarga nuevamente de dibujar la nave, del movimiento y del ataque de esta, aunque esta vez, a diferencia de la implementación en la clase “PlayerShip” no tenemos el código que se encarga de manera lo que pasa cuando la nave esta heria. Para la implementación para el método

“attack” que es el segundo método abstracto, tenemos que este nuevamente se encarga de restringir que tan seguido pueden disparar, también se encarga de crear una bala, rotar la textura de la bala y reproducir un sonido de disparo.

BossShip

- Esta cuenta con 4 atributos adicionales los cuales son velocidad de las balas, contador de ataque, un arreglo de patrones de ataque y un ataque actual, estos dos últimos atributos utilizan una interfaz llamada “AttackPattern”, además esta clase cuenta con un constructor el cual tiene los mismos parámetros que el constructor de la clase abstracta. En cuanto a métodos esta clase cuenta con los dos métodos abstractos, más una clase concreta, esta última, llamada “selectRandomAttackPattern”, se encarga de elegir un patrón de ataque que utilizara el jefe. La clase abstracta “draw” se encarga del dibujado del jefe y además de llamar a la otra clase abstracta “attack” la cual se encarga del ataque del jefe esta además llama a la clase concreta “selectRandomAttackPattern”, lo que hace que cada cierto tiempo el jefe utilice un nuevo patrón de ataque seleccionado desde el arreglo “attackPatterns” lo que añade complejidad a la batalla final.

Diagrama UML



GM1.5 Diseño y codificación de 1 (una) interfaz que sea implementada por al menos 2 (dos) clases. La interfaz debe ser utilizada por alguna otra clase (contexto)

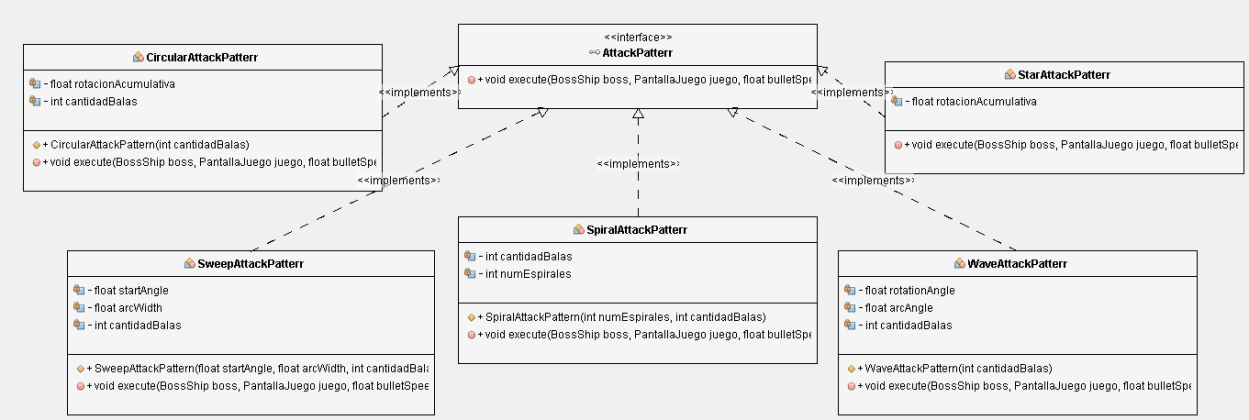
La interfaz AttackPattern se diseñó para gestionar de manera eficiente los diferentes patrones de ataque del jefe en el juego. Su propósito es facilitar la implementación de ataques al proporcionar un método. La interfaz define el método execute, que permite a cada clase de ataque tener su propia implementación. Esto añade dinamismo al juego, ya que el jefe puede seleccionar aleatoriamente el patrón de ataque, haciendo que el jugador no sepa cuál será el siguiente movimiento.

La interfaz se implementa en varias clases, cada una con un tipo específico de ataque. Por ejemplo, SpiralAttackPattern ejecuta un ataque en espiral, lanzando disparos que giran a su alrededor. StarAttackPattern dispersa los disparos en varias direcciones desde el centro del jefe, creando un efecto de estrella. Por otro lado, SweepAttackPattern permite que el jefe barra disparos de un lado a otro, mientras que CircularAttackPattern lanza proyectiles en un patrón circular, rodeándolo por completo. Finalmente, WaveAttackPattern utiliza una lógica que genera disparos en trayectorias curvas, imitando un movimiento ondulante.

En el contexto del juego, el jefe elige aleatoriamente uno de estos patrones de ataque y llama al método execute de la clase correspondiente, lo que permite que el combate sea menos predecible y más desafiante para el jugador. Esta estructura modular no solo mejora la jugabilidad, sino que también facilita la incorporación de nuevos patrones de ataque si es que se desea.

Diagrama UML

- **Interfaz AttackPattern**



GM1.6 Se debe aplicar encapsulamiento y principios OO

El encapsulamiento que utilizamos en nuestro juego fue el de datos, el cual se refiere a la restricción de los atributos de una clase, es decir, que tenga visibilidad

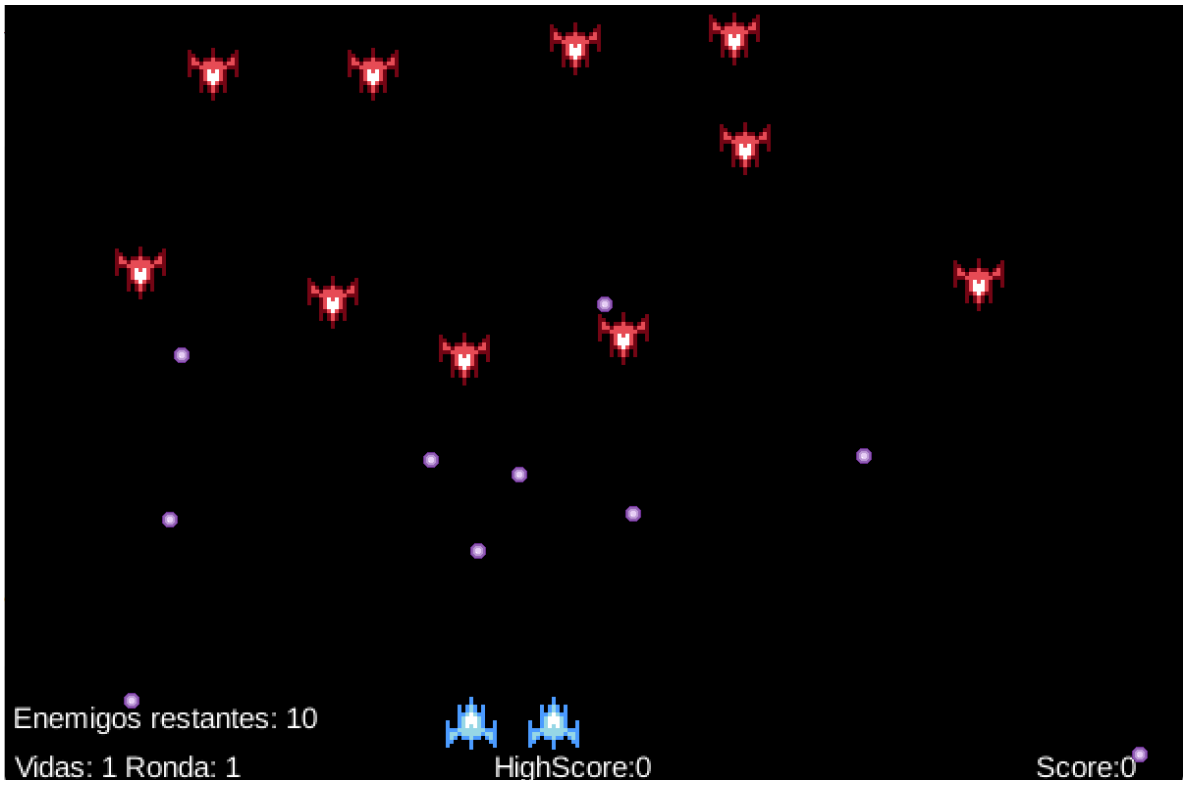
private o protected. En nuestro caso, en todas las clases se puede revisar que los atributos son privados y que la mayoría de los atributos tienen sus setter y getter

En el código se pueden observar el principio de Liskov.

El principio se utiliza con la clase abstracta “SpaceShip” y las clases concretas “PlayerShip”, “EnemyShip” y “BossShip”, ya que estas últimas extienden de la clase abstracta “SpaceShip” sin cambiar ninguno de los métodos concretos existentes, solamente añadiendo nueva funcionalidad.

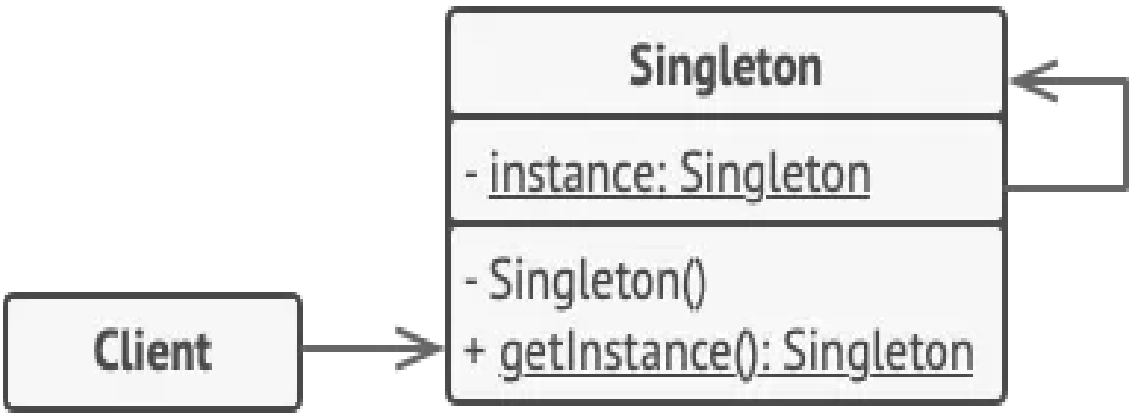
GM2.1 Aplicación del patrón de diseño Singleton. Describir problema, contexto, solución, UML con participantes, roles e interrelaciones.

A la hora de testear o modificar el proyecto puede ser que agreguemos nuevas instancias sin percatarse de la existencia de instancias anteriores lo que puede causar problemas en la ejecución del juego.

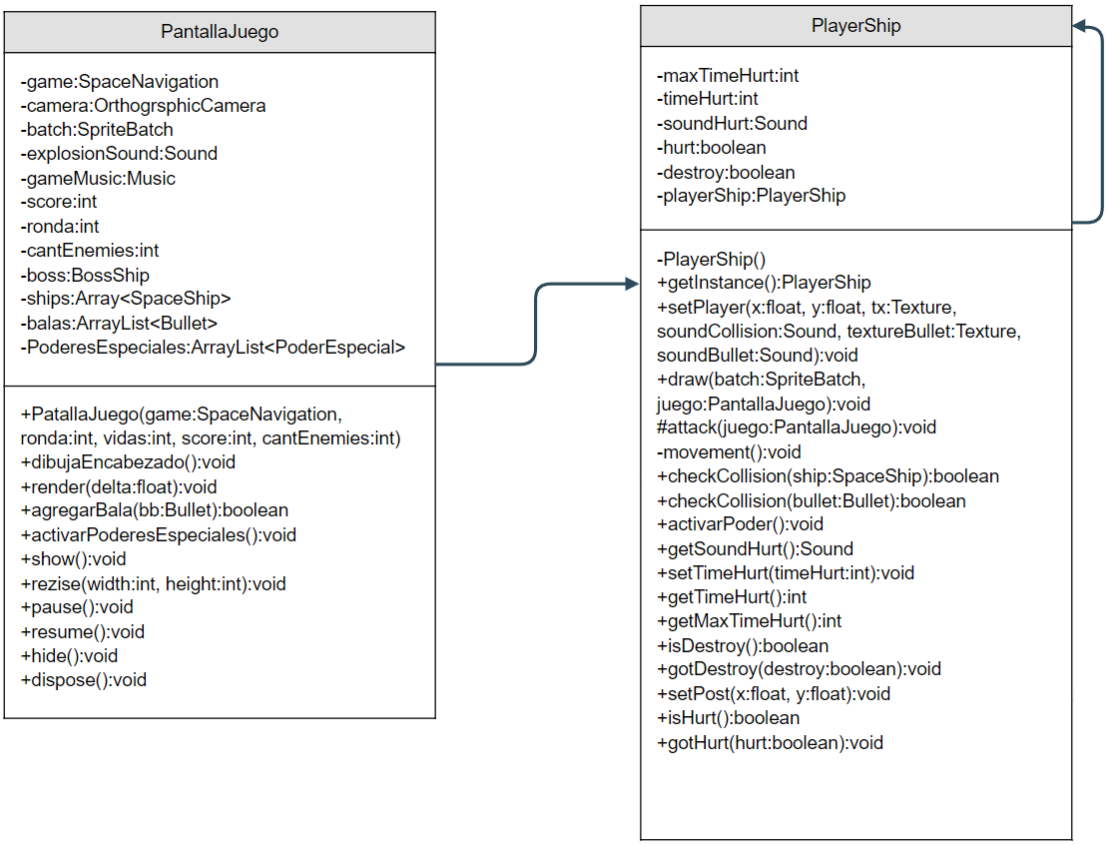


Para evitar esto en este programa utilizaremos el patrón Singleton el cual evita la creación de nuevas instancias, en este caso se utilizara a la hora de crear la nave del jugador, ya que la creación de múltiples naves del jugador podría causar una mala experiencia de juego al interferir con la ejecución normal de la partida confundiendo al jugador y añadiendo dificultad y mecánicas no deseada.

La clase Singlenton en este caso sería la clase NaveJugador, esta clase se encarga de darle control e interacción al usuario con el resto de objetos del juego, esta clase interactúa con las clases Bala, Enemigo y PantallaJuego. **



UML Patrón Singleton



UML Patrón implementado

GM2.2 Aplicación del patrón de diseño Template Method. Describir problema, contexto, solución, UML con participantes, roles e interrelaciones.

En el juego existen dos tipos de niveles:

- Nivel que contiene solo naves enemigas

- Nivel que contiene solo un jefe final

Al principio, toda la lógica de los niveles había sido implementada en la clase PantallaJuego, pero esto presenta un problema al momento de querer agregar nuevos tipos de nivel, ya que cualquier modificación implica cambiar el código dentro de PantallaJuego y posiblemente modificar la lógica de los niveles ya hechos. Esto haría que el código sea más complicado de entender y más propenso a errores, por lo que para evitar esto se utiliza el patrón Template Method.

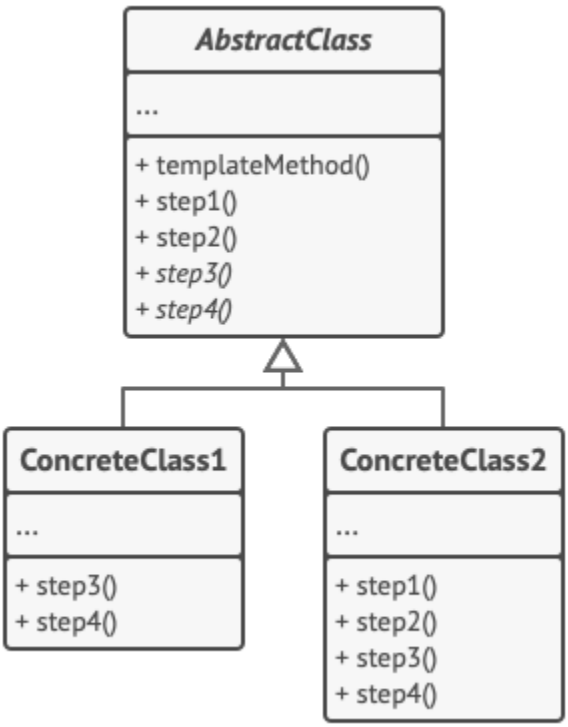
Este patrón es ideal para los algoritmos que tienen una base en común, pero con pasos que varían según el contexto. Para este caso el patrón ayuda a definir una estructura común para los niveles, mientras que los tipos de niveles se podrán manejar en clases concretas donde cada una tendrá sus pasos específicos.

La base de los niveles es una clase abstracta llamada Nivel, y de esta extienden dos clases NivelEnemigo y NivelJefe (debido a que existen esos dos tipos de niveles). La clase Nivel, actúa como la clase template. Esta clase es el esqueleto para cada nivel, incluye pasos para inicializar el nivel y mientras el nivel no esté completado el juego sigue hasta que finalice, y el nivel aumenta.

Esta clase también incluye un método que no es implementado, debido a que cada subclase tiene su propia implementación.

Las subclases NivelEnemigo y NivelJefe son las implementaciones concretas que extienden de la clase Nivel. Cada una de estas subclases implementa los pasos específicos que varían entre los tipos de nivel, como el término del nivel.

La clase que interactúa con este patrón es la clase PantallaJuego al invocar su método para iniciar y gestionar un nivel.



UML Patrón Template Method

GM2.3 Aplicación del patrón de diseño Strategy (Estrategia). Describir problema, contexto, solución, UML con participantes, roles e interrelaciones.

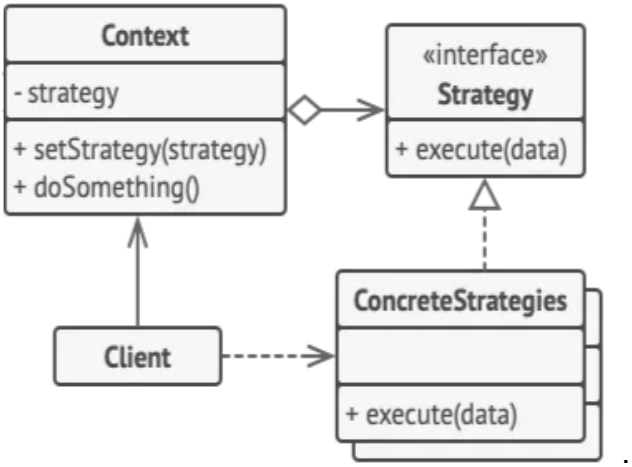
En el juego se quiso que el jefe final tuviera una diversa cantidad de patrones de ataque y, además pudiera elegir entre estos de manera aleatoria, esto con la intención de crear un nivel mucho más desafiante para el jugador.

Para realizar esto se utilizó el patrón Strategy, ya que este nos permite añadir diversos patrones de ataque y tener un mayor control de cuando se utiliza cada uno, obteniendo así un jefe final que es desafiante e impredecible, al que también se le pueden agregar nuevos patrones de ataque de manera sencilla.

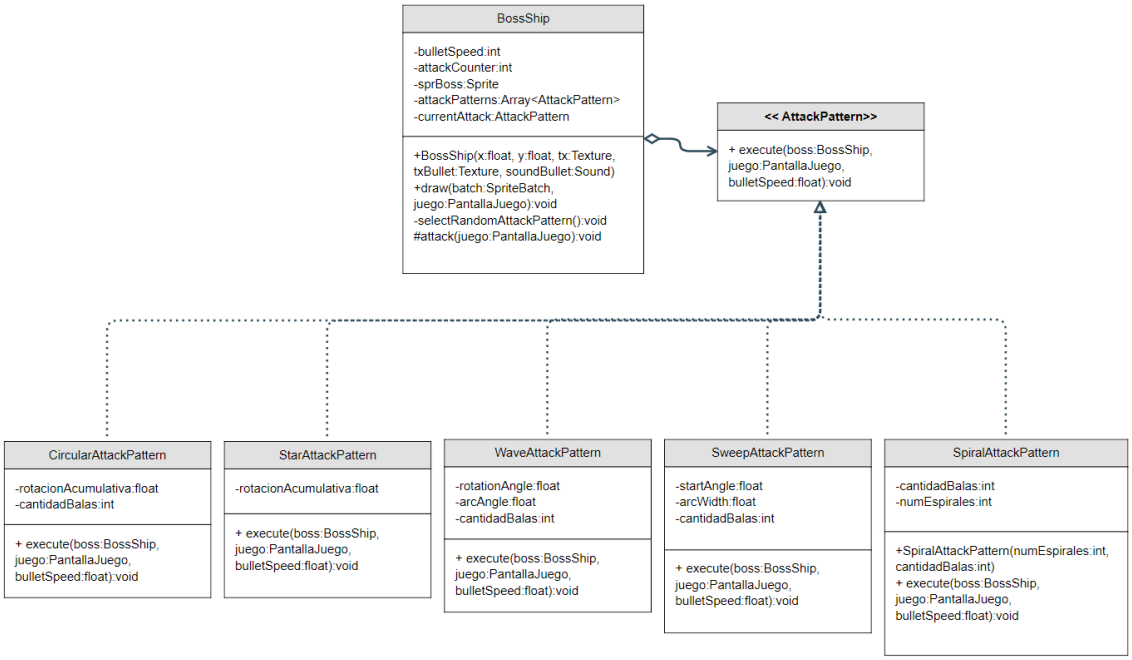
La interfaz Strategy que se utiliza se llama PatronAtaque, ésta es implementada por 5 clases concretas, llamadas: PatronCircular, PatronOla, PatronEstrella, PatronEspiral y PatronBarerr.

El contexto de esta interfaz sería la clase NaveJefe.

Cada clase concreta que extiende de la interfaz PatronAtaque se encarga de la creación bala en un patrón distinto, interactuando con la clase Bala y la clase Nivel, además de la clase contexto NaveJefe.



UML Patrón Strategy



UML Patrón implementado

GM2.4 Aplicación de un patrón de diseño creacional (Builder o Abstract Factory)

1. *Problema*

En un sistema donde se requiere crear diferentes objetos relacionados que comparten una estructura común, pero varían en sus características específicas, el proceso de creación puede volverse repetitivo, poco flexible y propenso a errores. Por ejemplo, en un juego, necesitas instanciar diferentes tipos de naves espaciales (PlayerShip, EnemyShip, BossShip) con configuraciones específicas según su tipo.

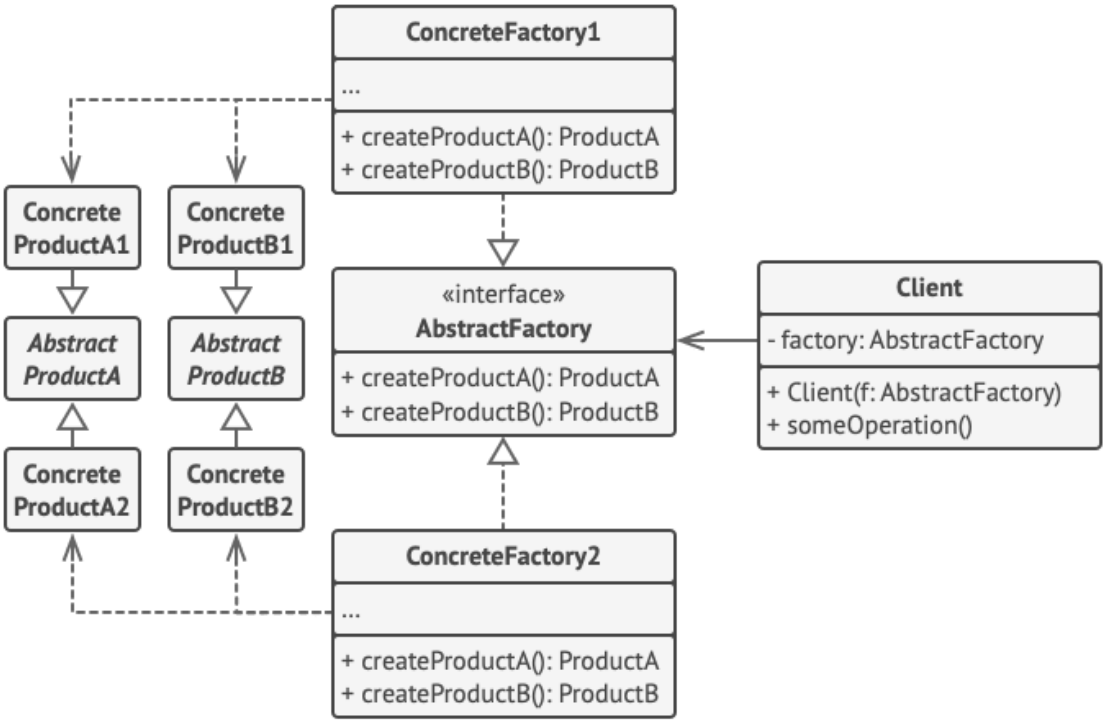
2. *Contexto*

Los objetos comparten características comunes (como posición, textura, sonido), pero cada uno tiene comportamientos particulares.

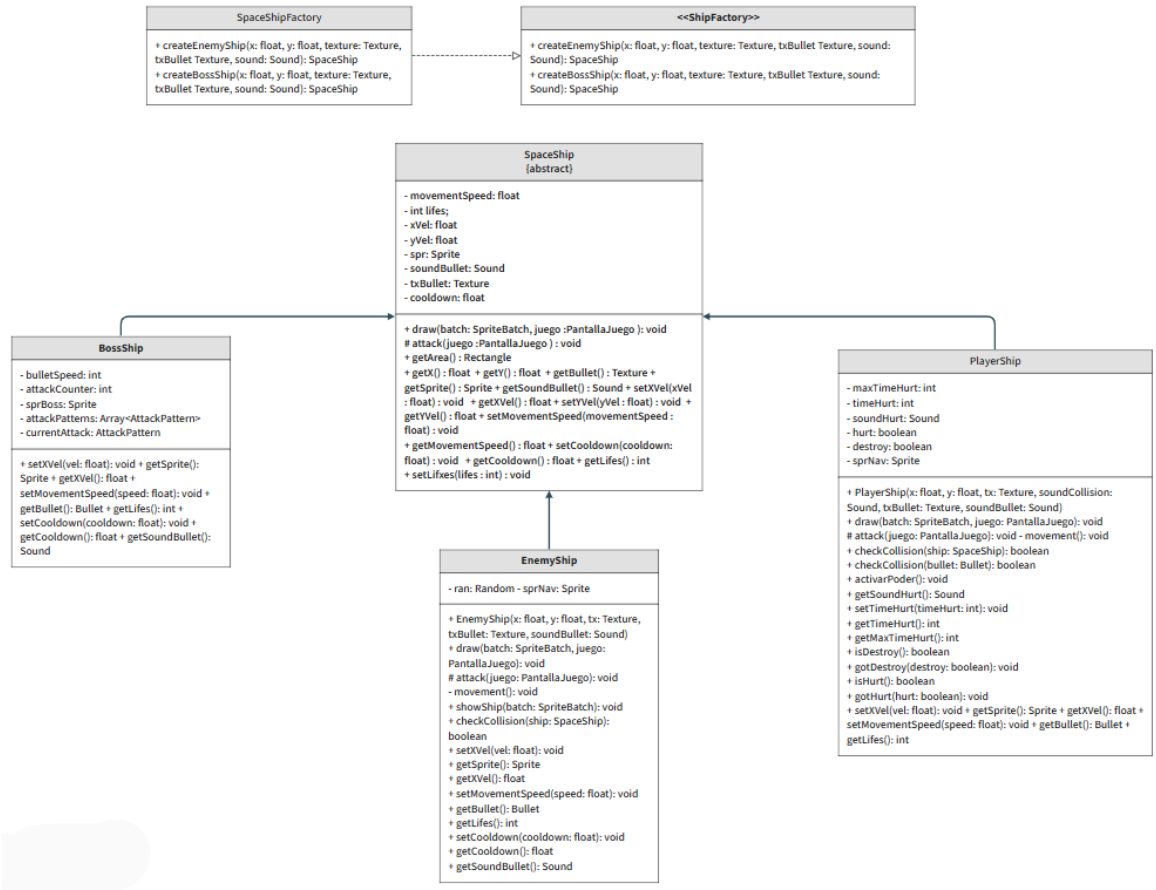
Es necesario centralizar y desacoplar la lógica de creación de objetos del resto del código para facilitar su mantenimiento, extensibilidad y prueba.

3. *Solución*

El patrón Abstract Factory ofrece una solución al encapsular la creación de objetos relacionados en una familia. Esto permite que el sistema use una interfaz común para crear instancias sin preocuparse por sus clases concretas. El patrón asegura que los objetos se creen con las configuraciones correctas y que las variaciones entre tipos se manejen de forma centralizada.



UML Patrón Abstract Factory



UML implementado

Participantes

- ShipFactory (Interfaz): Define los métodos para crear los diferentes tipos de naves.
- SpaceShipFactory (Fábrica Concreta): Implementa los métodos de la interfaz NaveFactory y devuelve instancias específicas de las naves.
- SpaceShip (Clase Abstracta): Define la estructura y el comportamiento común de todas las naves.
- PlayerShip, EnemyShip, BossShip (Clases Concretas): Representan las variaciones específicas de SpaceShip con comportamientos y características particulares.

Roles e Interrelaciones

La fábrica concreta (SpaceShipFactory) utiliza la interfaz (NaveFactory) para instanciar objetos específicos de SpaceShip.

Las subclases (PlayerShip, EnemyShip, BossShip) heredan atributos y métodos comunes de la clase abstracta SpaceShip, pero implementan sus propios comportamientos.

El cliente interactúa con la fábrica y no necesita conocer las clases concretas, manteniendo el desacoplamiento.