GM-Final

Profesor Claudio Cubillos

Ignacio Reyes Torres

Joaquín Muñoz Alarcón

Benjamin Pizarro Saavedra

**INTRODUCCIÓN**

El proyecto “SpaceNavigation” consiste en el desarrollo de un videojuego 2D creado con la biblioteca **LibGDX** en el lenguaje **Java**, cuyo objetivo principal es aplicar los fundamentos de la programación orientada a objetos dentro de un entorno interactivo y visual. En este juego, el jugador controla una nave submarina que debe esquivar obstáculos, destruir bombas y recoger distintos tipos de power-ups que mejoran su rendimiento o le otorgan ventajas temporales.

El desarrollo del proyecto incluyó la implementación de diversas pantallas, como el menú principal, la pantalla de instrucciones, el juego en ejecución y la pantalla de fin de partida, siguiendo la estructura que ofrece el framework LibGDX. Además, se incorporaron clases propias para representar los diferentes elementos del juego, aplicando conceptos como herencia, encapsulamiento, abstracción e interfaces.

**GM 1.1 Explicación del juego desde la perspectiva del jugador**

Narrativa y concepto del juego

“Subma Game” Es un juego estilo shooter en 2D estilo arcade con una vista cenital. El jugador toma los mandos de un pequeño submarino de guerra que se adentra en aguas enemigas, cubiertas de minas submarinas que son arrastradas con fuerza por la corriente marina, haciendo de gran dificultad el paso de los tripulantes. Con precisión y torpedos debe eliminar las minas sin ser tocados por ellas, pues unas cuantas bastan para acabar con el submarino.

Sobrevive las máximas rondas que puedas y demuestra tu habilidad teniendo la máxima puntuación junto a los powerUps que aparecen aleatoriamente.

Objetivo del jugador:

Destruir Enemigos: El jugador debe disparar torpedos para destruir todas las minas (Ball2) que aparecen en la pantalla.

Avanzar de Ronda: Una vez que todas las minas de la ronda actual son eliminadas, el juego avanza a la siguiente ronda, la cual presenta una mayor cantidad y velocidad de minas.

Evitar Daño: El jugador debe esquivar las minas. Cada colisión con una mina le resta vida al submarino. Si las vidas llegan a cero, la partida termina.

Conseguir Power-Ups: Al destruir minas o según el transcurso de la ronda, existe la probabilidad de que aparezcan ítems especiales (power-ups) que otorgan ventajas temporales.

Superar el High Score: El objetivo final es obtener la mayor puntuación (Score) posible, superando el récord anterior (HighScore).

Elementos Principales del Juego:

El Submarino (Jugador): Es el avatar controlado por el jugador (Nave4). Puede moverse libremente por la pantalla y disparar torpedos. Tiene un número limitado de vidas.

Las Minas (Enemigos): Son los obstáculos principales (Ball2). Se mueven por la pantalla rebotando en los bordes. Destruirlas otorga puntos.

Los Torpedos (Disparos): Son los proyectiles (Bullet) que dispara el submarino. Se mueven en línea recta (hacia arriba) y destruyen las minas al impacto.

Power-Ups: Son ítems que caen por la pantalla y otorgan beneficios al ser recogidos:

Vida Extra: Otorga una vida adicional.

Velocidad: Aumenta drásticamente la velocidad de movimiento del submarino por 5 segundos.

Escudo: Otorga un escudo que protege al jugador del próximo impacto de una mina.

Disparo Doble: Aumenta el disparo de uno a dos torpedos simultáneos.

Controles de Movimiento y Acción:

Movimiento:

Tecla W : Mover el submarino hacia arriba.

Tecla S : Mover el submarino hacia abajo.

Tecla A : Mover el submarino hacia la izquierda.

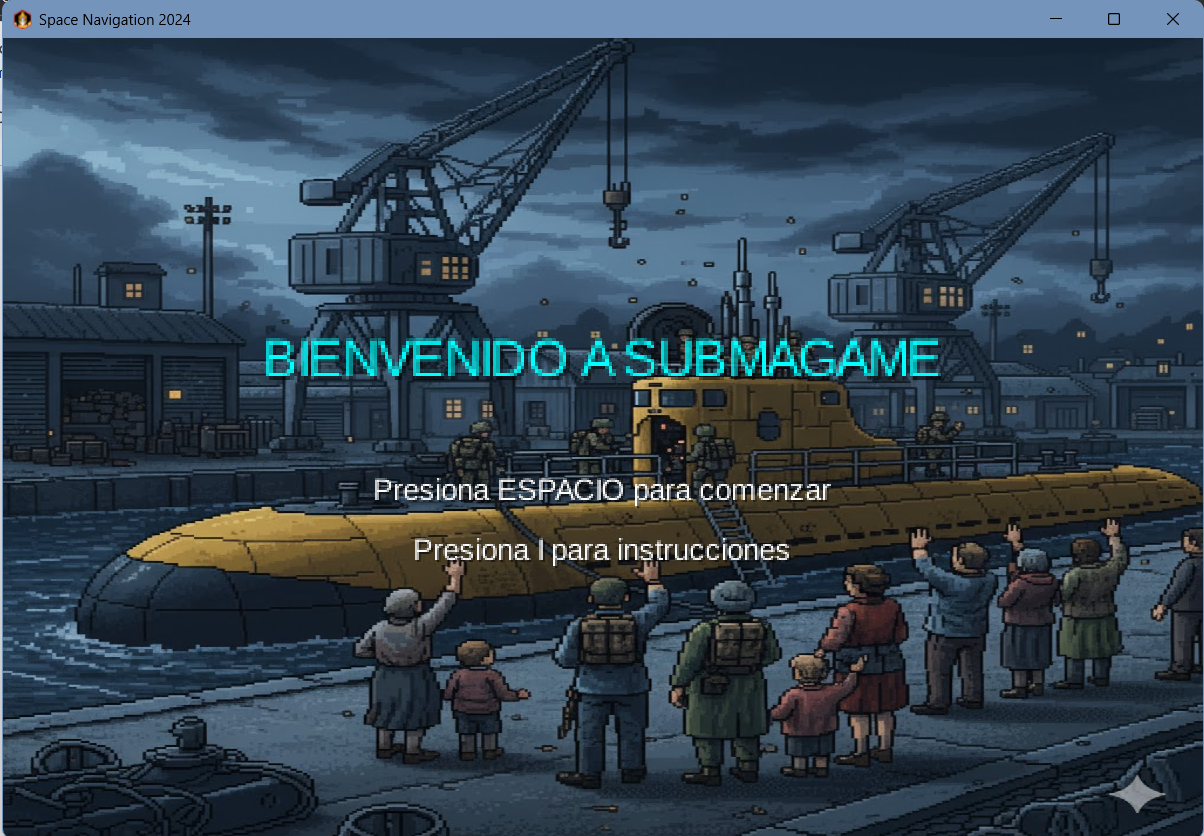
Tecla D : Mover el submarino hacia la derecha.

Tecla Barra Espaciadora: Disparar un torpedo.

Tutorial y Flujo de Juego:

El flujo del juego es un ciclo simple de tres pantallas:

Pantalla de Menú: Es la primera pantalla que ve el jugador. Muestra el título del juego y una instrucción simple para comenzar, como "Presiona ESPACIO para comenzar", Además de la opción “Presiona I para instrucciones”.



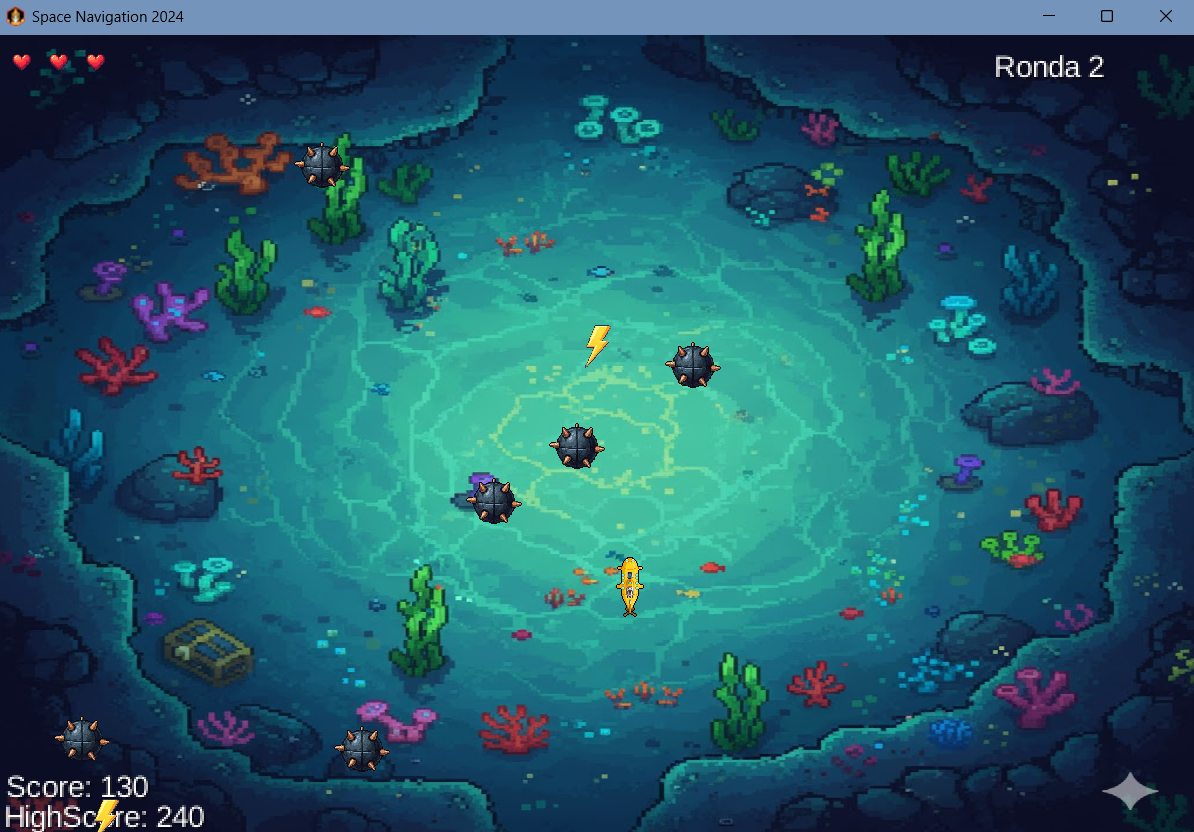
Pantalla de Juego (Interfaz Principal): Esta es la pantalla donde ocurre toda la acción. El jugador puede ver:

Su Submarino y las Minas.

Los Power-Ups cuando aparecen.

El HUD en la parte superior/inferior, que muestra información crucial:

1. Vidas restantes.
2. Ronda actual.
3. Puntuación actual (Score).
4. Puntuación más alta (High Score).



Pantalla de Fin de Partida (Game Over): Si el jugador pierde todas sus vidas, es llevado a esta pantalla. Muestra un mensaje de "Game Over" y la puntuación final. También ofrece una instrucción para reiniciar la partida.



**GM 1.2 Análisis del juego desde la Perspectiva de ingeniería de software**

Juego Base de Referencia

El proyecto actual es una modificación y refactorización de un código base preexistente llamado "SpaceNav",El proyecto requiere el uso de JDK 17, aunque inicialmente se consideró la posibilidad de utilizar JDK 11 como está especificado en las bases del proyecto. Esta elección se debe a que la dependencia de Gradle, **io.github.fourlastor:construo:1.4.1**, demanda de forma explícita una máquina virtual Java de versión 17 o superior para poder compilar correctamente, también se le envió un correo tanto al profesor como a los ayudantes debido a un error de compilación al intentar usar JDK 11 (Dependency requires at least JVM run time version 17. this build uses a java 11 JVM) pero al no tener una respuesta nuestra solución fue instalar y configurar todo para el uso de JDK 17, resolviendo y permitiendo la compilación del proyecto para poder seguir avanzando.

Modificaciones Clave del Proyecto

A partir de la base de "SpaceNav", se realizaron las siguientes modificaciones funcionales y estructurales:

Cambio de Temática: Se reemplazaron todos los assets visuales y de audio. El juego pasó de la temática espacial de "SpaceNavigation" a una temática submarina (submarino y minas).

Refactorización del Movimiento:

Se implementó el movimiento basado en Delta-Time (delta) para que la velocidad sea independiente de los FPS.

Se cambió el control de la nave, pasando de un sistema rotacional (giro y empuje) a un movimiento libre en 4 direcciones (W, A, S, D).

Implementación de Clase Abstracta:

Se diseñó un sistema de power-ups (Vida, Velocidad, Escudo) usando la clase abstracta PowerUp.

PantallaJuego maneja una lista genérica ArrayList<PowerUp>, cumpliendo con el principio de Abierto/Cerrado.

Implementación de Interfaz:

Se creó la interfaz Colisionable para estandarizar las colisiones.

Todas las entidades físicas (Nave4, Ball2, Bullet, PowerUp) ahora implementan esta interfaz, dependiendo de la abstracción (.getHitbox()).

Mejoras de Jugabilidad:Se añadió un sistema de menús completo (PantallaMenu, PantallaInstrucciones, PantallaGameOver).

Se implementó un HUD visual (corazones para las vidas) y una función de Pausa (tecla ESC).

Modificaciones realizadas:

Sonidos personalizados: Le pusimos un sonido único a cada PowerUp (Vida, Escudo, Velocidad y doble Misil).

Efecto al recibir daño: Ahora la nave parpadea cuando te golpean. Sirve para avisarte visualmente que eres invencible por unos segundos.

Explosiones animadas: Creamos una clase de explosiones para cuando destruyas una mina, aparezca una animación en lugar de simplemente desaparecer.

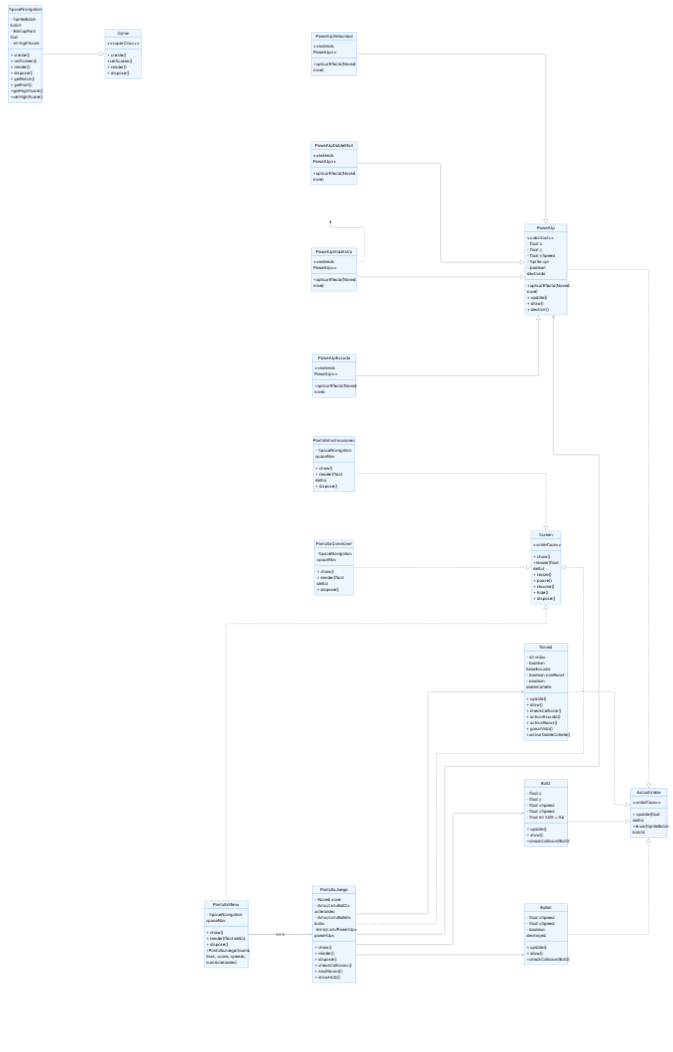
Imagen propia para el Doble Cohete: El PowerUp de Doble Cohete tiene su propio sprite para que lo reconozcas al instante.

Fondo en movimiento: Implementamos un efecto de “scrolling” en el fondo marino. Ahora se mueve constantemente, dando la sensación real de que la nave está avanzando por el océano.

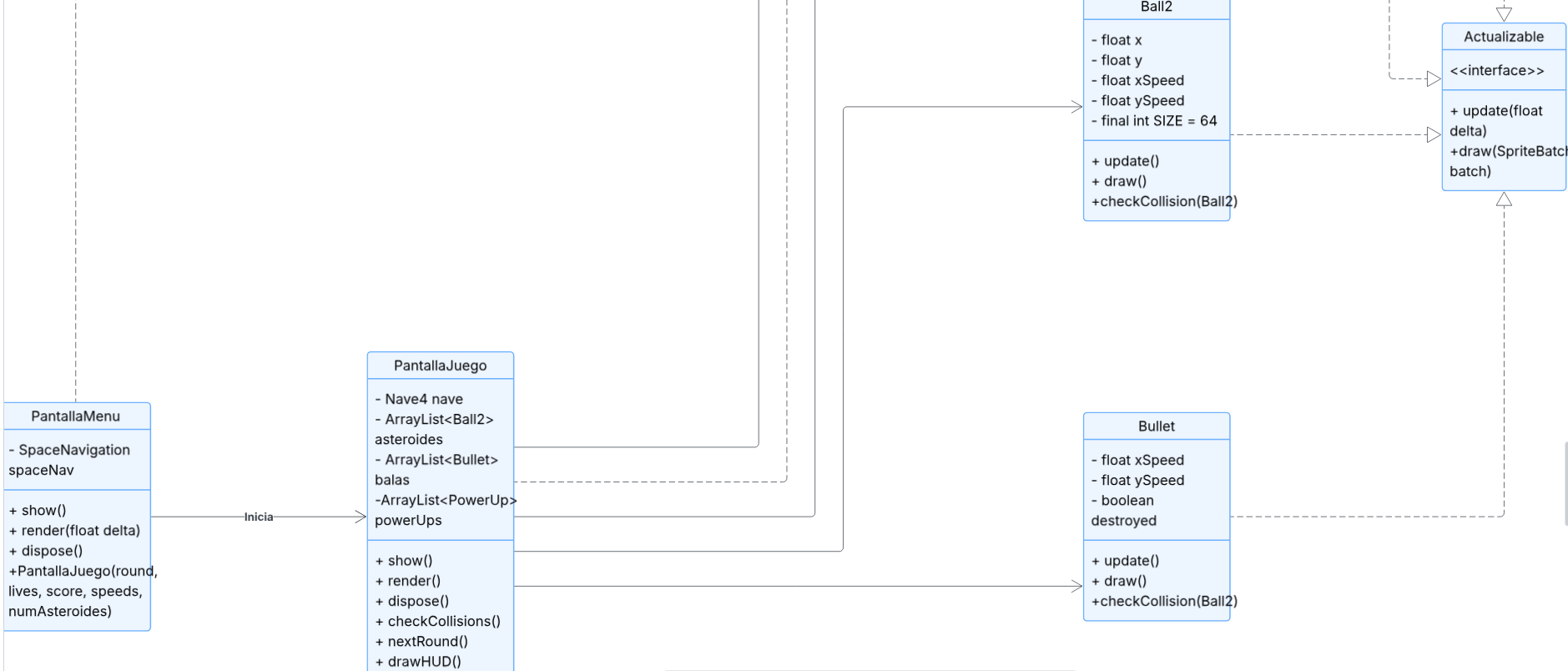
Modificación de puntos: Ahora cada vez que avances de ronda, el jugador podrá conseguir más puntos al explotar las minas.

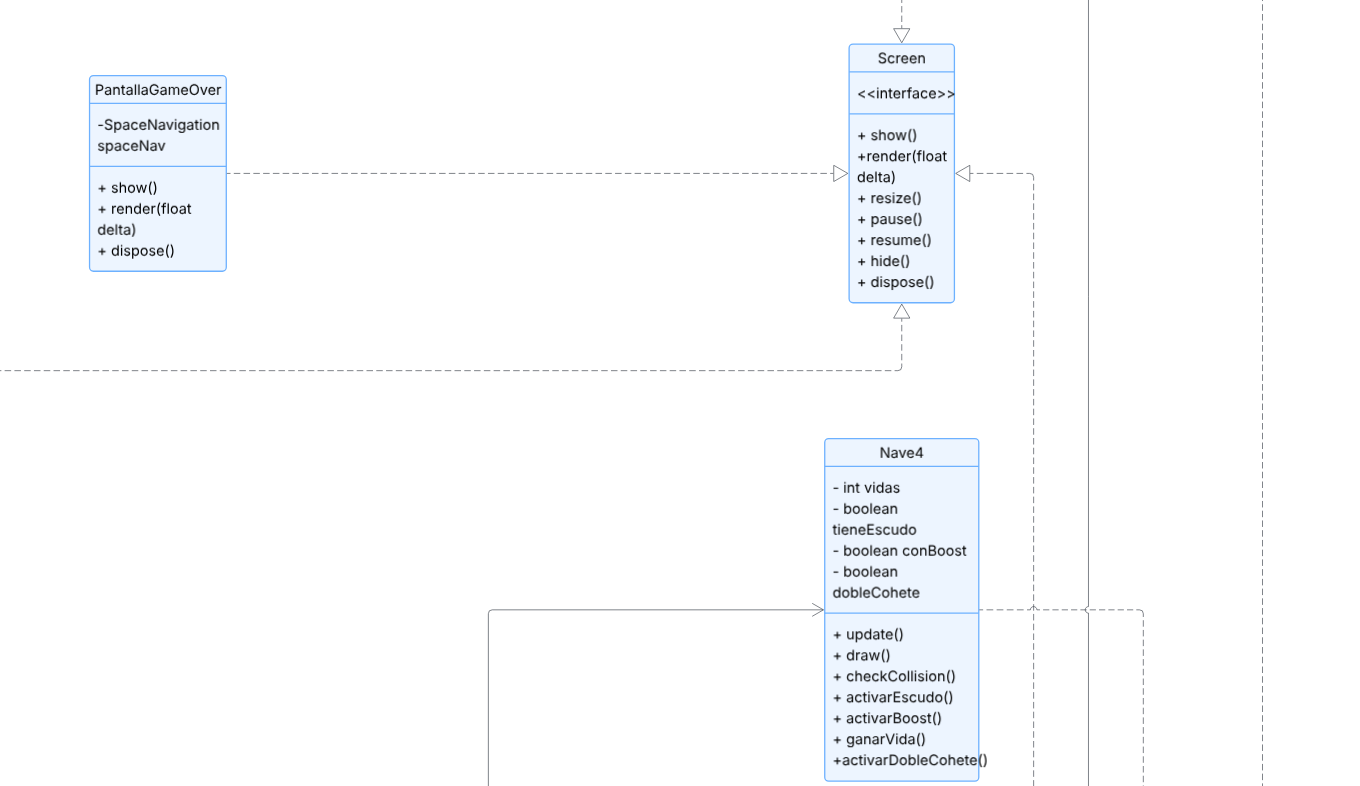
**GM 1.3 Diseño de Diagrama UML**

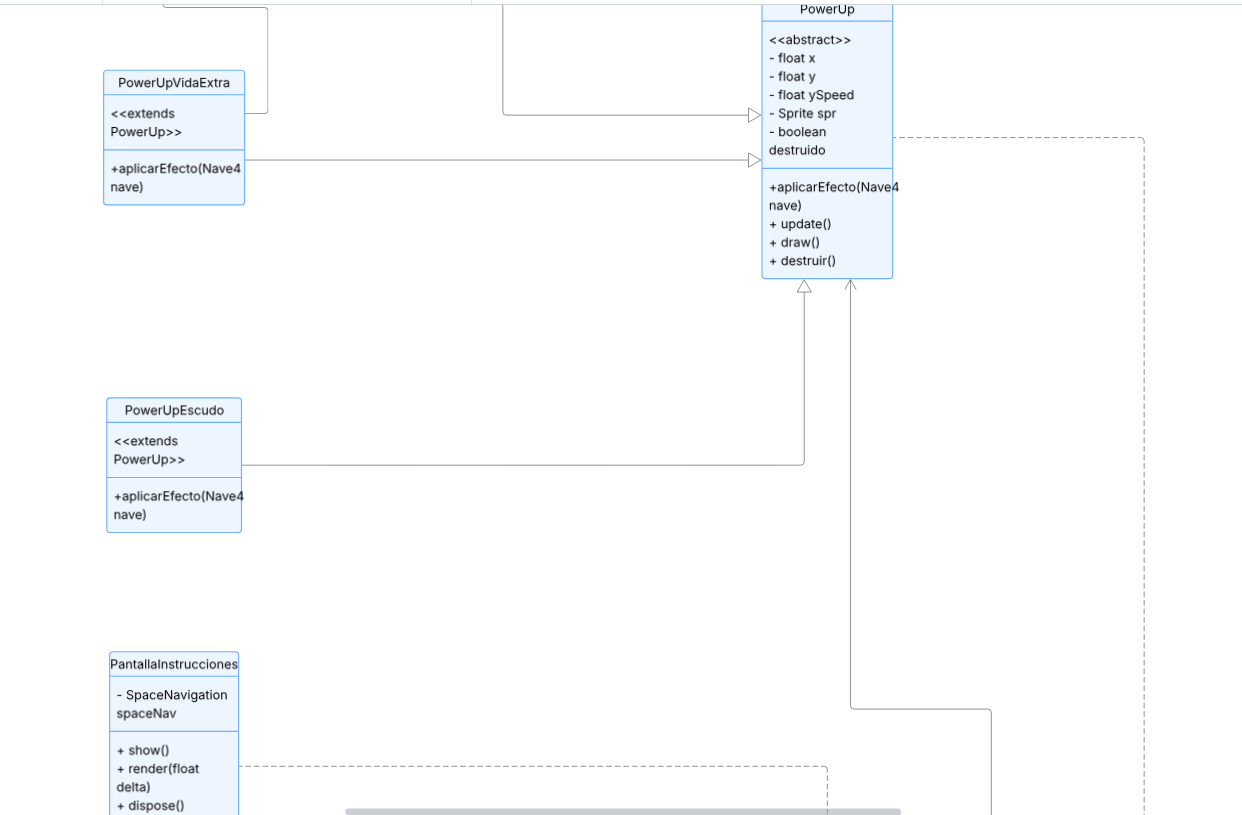
GENERAL:

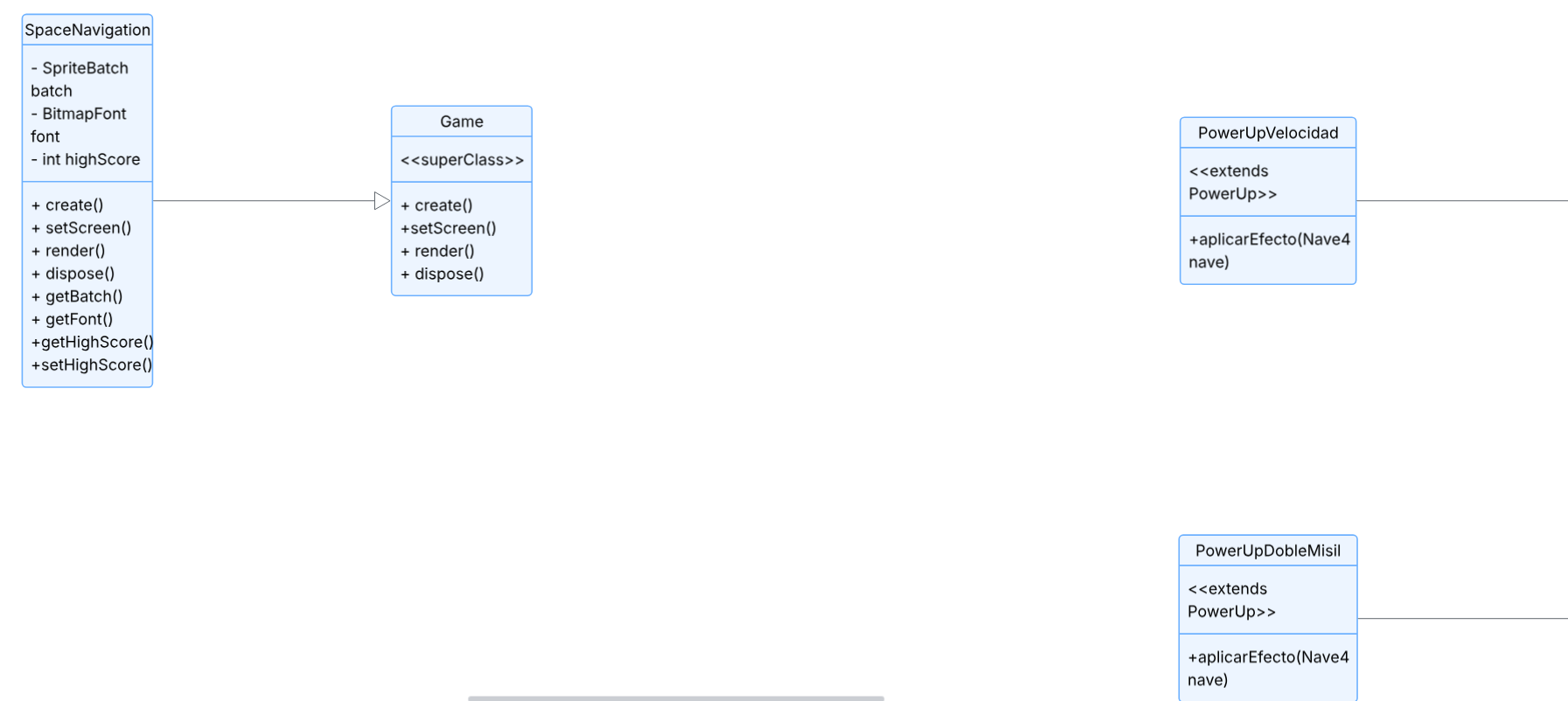


**Diagrama por partes:**









**Conclusión Uml:**

El UML se hizo así para reflejar un diseño profesional, modular y mantenible, donde cada clase cumple una función clara, las relaciones son coherentes, el código está encapsulado y el proyecto puede expandirse sin romper su estructura principal.

**GM 1.4 Diseño y codificación de 1 (una) clase abstracta que sea padre de al menos 2 (dos) clases.**

Clase Abstracta: Se implementó public abstract class PowerUp como clase padre para todos los ítems recolectables.

Comportamiento Base: Esta clase abstracta maneja el código común (movimiento, dibujo) y define un método abstracto clave: public abstract void aplicarEfecto(Nave4 nave).

Clases Hijas: Se crearon 3 clases hijas que heredan de PowerUp e implementan el método abstracto:

PowerUpVidaExtra

PowerUpVelocidad

PowerUpEscudo

Contexto (PantallaJuego): La clase PantallaJuego utiliza la abstracción. Gestiona todos los ítems en una ArrayList<PowerUp> y llama a p.aplicarEfecto() sin necesidad de saber qué tipo de power-up específico es, cumpliendo así con el Principio de Abierto/Cerrado.

**GM 1.5 Diseño y codificación de 1 (una) interfaz que sea implementada por al menos 2(dos) clases.**

En el proyecto se implementó la interfaz Actualizable, que define el método update(float delta) para unificar el comportamiento de los objetos que cambian su estado durante el juego. Esta interfaz fue aplicada en las clases Bullet y PowerUp, permitiendo que ambas actualicen su posición y controlen su eliminación de forma independiente. Con ello se aplica el principio de polimorfismo, logrando un código más modular, reutilizable y coherente con los principios de la programación orientada a objetos.

**GM 1.6 Aplicación de Encapsulamiento y Principios OO.**

Durante el desarrollo del juego se aplicaron los principales principios de la programación orientada a objetos, como el encapsulamiento, la herencia y el polimorfismo.

Se usó encapsulamiento al mantener los atributos como privados y controlar su acceso mediante métodos, evitando que otras clases modifiquen el estado interno de los objetos de forma directa. Por ejemplo, en Nave4 las vidas y estados se gestionan a través de métodos como ganarVida() o activarEscudo(), mientras que en Bullet y PowerUp se devuelven copias seguras de las áreas de colisión para proteger los datos.

El polimorfismo se aplicó mediante la interfaz Actualizable, implementada por Nave4, Bullet, Ball2 y PowerUp. Esto permitió que todas las entidades del juego se actualicen de forma uniforme dentro de PantallaJuego, sin depender de su tipo específico.

También se utilizó herencia en la clase abstracta PowerUp, de la cual derivan las distintas subclases que representan los ítems del juego. Así se evita repetir código y se facilita agregar nuevos tipos de power-ups en el futuro.

En conjunto, estas prácticas hacen que el código sea más ordenado, fácil de mantener y coherente con los principios de la programación orientada a objetos.

**GM 1.7 Utilización de GitHub.**

Se realizaron 4 commits en total, reflejando las distintas etapas del trabajo: la configuración inicial del proyecto, la implementación de las clases principales, la integración de la interfaz Actualizable y las correcciones finales.

enlace: <https://github.com/deftonenjoyer/SubmaGame>

**Conclusión 1.1:**

El desarrollo del proyecto “SpaceNav” permitió aplicar de manera práctica los principales conceptos de la programación orientada a objetos, integrando aspectos teóricos con la creación de un videojuego funcional. A lo largo del proceso se implementaron clases, interfaces y herencias, además de aplicar principios como el encapsulamiento y la abstracción para mantener un código limpio, modular y reutilizable.

El uso del framework LibGDX facilitó la creación de las distintas pantallas, el manejo de texturas y la interacción entre los elementos del juego, mientras que GitHub permitió un control adecuado de versiones y un seguimiento del progreso. Gracias a esto, se logró construir una estructura sólida que puede ampliarse con nuevas funcionalidades en el futuro.

**GM 2.1 Aplicación del Patrón de diseño Singleton.**

Singleton Clase Implementadora: HighScoreManager

Problema: El juego necesita mantener un estado global persistente (el puntaje más alto o High Score) que debe ser accesible desde múltiples pantallas (PantallaJuego para guardar el récord y PantallaMenu/PantallaGameOver para mostrarlo). Pasar este valor como parámetro entre constructores de pantallas crea un acoplamiento innecesario y hace el código propenso a errores si se pierden los datos al cambiar de contexto.

Contexto: El sistema de puntuación es único para toda la sesión de juego. No debe existir más de una que gestione el récord, ya que tener múltiples copias podría llevar a inconsistencias en los datos mostrados al jugador.

Solución: Se implementó la clase HighScoreManager utilizando el patrón Singleton.

Se privatizó el constructor (private HighScoreManager()) para evitar la instanciación directa con new.

Se creó una clase interna estática SingletonHolder que contiene la única instancia (INSTANCE), garantizando una inicialización segura y diferida (lazy initialization).

Se expone el método estático y público getInstance(), que es el único punto de acceso global para leer (getHighScore) o modificar (setHighScore) el récord desde cualquier parte del juego.

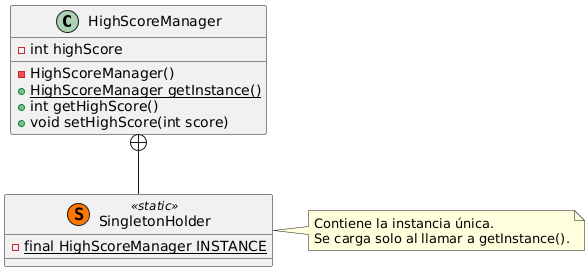
Clase HighScoreManager.

Atributo: - static INSTANCE: HighScoreManager (o dentro del Holder).

Método: + static getInstance(): HighScoreManager.

Relación: PantallaJuego y PantallaMenu tienen una relación de dependencia (flecha punteada) hacia HighScoreManager.

UML:Para el HighScoreManager usamos una implementación eficiente con una clase interna estática llamada SingletonHolder. Esto asegura que la tabla de puntajes sea única en todo el juego, también optimiza la memoria porque la instancia solo se crea realmente cuando la pedimos con el método getInstance, garantizando que sea seguro y rápido.



**GM 2.2 Aplicación del Patrón de diseño Template Method.**

Template Method (Método Plantilla)

Clase Implementadora: PowerUp (Clase Abstracta)

Problema: El juego cuenta con diferentes tipos de power-ups (Vida, Velocidad, Escudo, Doble Misil) en donde todos comparten una secuencia lógica idéntica al ser recogidos que es reproducir un sonido, aplicar un efecto y luego autodestruirse, sin embargo, el "efecto" específico varía drásticamente entre cada uno, duplicar la lógica de sonido y destrucción en cada subclase violaría el principio DRY (Don't Repeat Yourself).

Contexto: La clase PantallaJuego detecta la colisión y necesita activar el ítem, no debería ser responsabilidad de la pantalla orquestar los pasos internos (sonido, efecto, destrucción) de cada objeto.

Solución: Se utilizó el patrón Template Method en la clase abstracta PowerUp.

Se definió un método final llamado activarPowerUp(Nave4 nave). Este es el "esqueleto" del algoritmo que ejecuta los pasos en orden fijo: reproducirSonido(), aplicarEfectoUnico(nave), y destruir().

Se definieron los pasos comunes (reproducirSonido, destruir) en la clase padre.

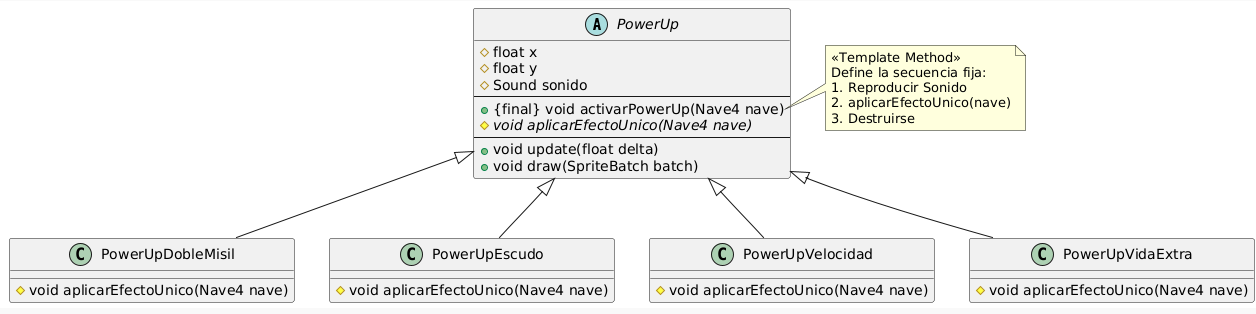
Se definió el paso variable como un método abstracto: protected abstract void aplicarEfectoUnico(Nave4 nave).

Las subclases (PowerUpVidaExtra, PowerUpVelocidad, etc.) solo implementan la lógica específica en aplicarEfectoUnico, sin preocuparse por la gestión del sonido o la destrucción.

Clase Abstracta PowerUp con el método + activarPowerUp() (Template) y el método abstracto aplicarEfectoUnico().

Clases Concretas (PowerUpVidaExtra, etc.) heredan de PowerUp y sobrescriben aplicarEfectoUnico().

UML: En la Nav4 aplicamos este patrón para desacoplar el ataque. La nave no tiene el código del disparo pegado adentro ya que delega la acción a la interfaz EstrategiaDisparo, esto es muy útil porque nos permite cambiar de disparo simple a doble en tiempo real cuando agarramos el ítem, simplemente asignándole una nueva estrategia a la nave, sin tener que tocar ni una línea de su código base.



**GM 2.3 Aplicación del Patrón de diseño Strategy.**

Patrón: Strategy (Estrategia) Clase Implementadora: Nave4 (Contexto) y EstrategiaDisparo (Interfaz)

Problema: La nave del jugador necesita cambiar su comportamiento de disparo dinámicamente durante el juego (de un solo disparo a disparo doble) al recoger un power-up. Utilizar condicionales complejos (if (tieneDobleMisil)else) dentro del método update o draw hace que la clase Nave4 sea difícil de mantener y viola el Principio de Abierto/Cerrado, ya que añadir un nuevo tipo de disparo requeriría modificar la clase de la nave.

Contexto: El mecanismo de disparo es una familia de algoritmos intercambiables. La nave necesita delegar cómo disparar a un objeto experto en esa tarea.

Solución: Se implementó el patrón Strategy.

Interfaz Strategy: Se creó la interfaz EstrategiaDisparo con el método disparar(Nave4 nave, PantallaJuego juego).

Estrategias Concretas: Se crearon las clases DisparoSimple y DisparoDoble que implementan la interfaz con sus lógicas respectivas (crear 1 bala vs crear 2 balas).

Contexto: La clase Nave4 mantiene una referencia privada private EstrategiaDisparo estrategiaDisparo.

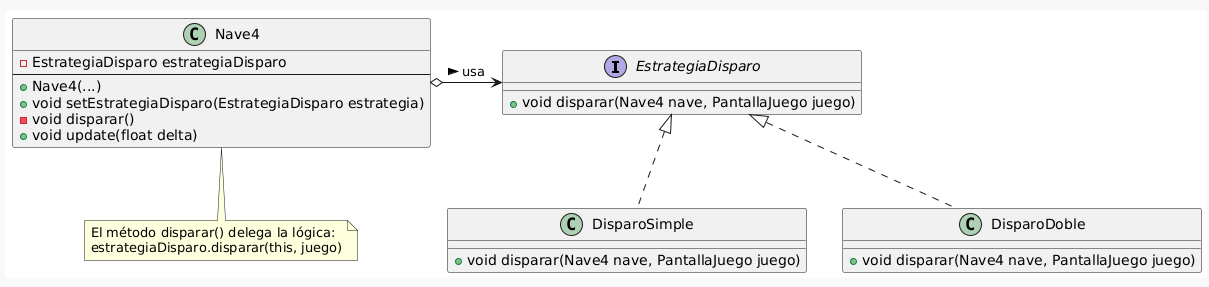
Intercambio: Cuando se recoge el power-up, se llama a nave.activarDobleCohete(), lo cual cambia la referencia en tiempo de ejecución (this.estrategiaDisparo = new DisparoDoble()). El método disparar() de la nave simplemente delega la acción: estrategiaDisparo.disparar(this, juego).

Interfaz EstrategiaDisparo.

Clases DisparoSimple y DisparoDoble implementan la interfaz (flecha punteada con triángulo cerrado).

Clase Nave4 tiene una relación de agregación (rombo blanco) o asociación con EstrategiaDisparo.

UML: Para los PowerUps definimos un esqueleto en la clase padre PowerUp,usamos un método final que obliga a seguir siempre el mismo orden de reproducir sonido, aplicar el efecto y destruirse. De esta forma, las clases hijas como VidaExtra o Escudo solo se preocupan por rellenar la parte específica del efecto en el método abstracto, reutilizando toda la lógica común sin repetir código.

****

**GM 2.4 Continuidad en la utilización de GitHub.**

**Conclusión 2.1:**

En conclusión, la integración de estos tres patrones nos permitió dejar un código mucho más ordenado y profesional. El Singleton aseguró la integridad de los puntajes globales, el Strategy nos dio la flexibilidad para cambiar las armas de la nave en tiempo real sin romper nada y el Template Method organizó la lógica de los PowerUps para no repetir código innecesario, construimos una base sólida que hace que el juego sea fácil de mantener y expandir en el futuro.