Proyecto GameMaker: Modificación de GameLluvia

Matias Pardo
Juan Pablo Pizarro
Joaquín Saldivia

24 de noviembre de 2024
Pontificia Universidad Católica de Valparaíso
Proyecto de Programación Avanzada

Docente: Claudio Cubillos.

Índice

1. Introducción

El paradigma de la Programación Orientada a Objetos (POO) ha revolucionado la forma en que desarrollamos software, proporcionando un marco estructurado para construir aplicaciones modulares, escalables y fáciles de mantener. La POO se basa en cuatro principios fundamentales: encapsulamiento, herencia, polimorfismo y abstracción. Estos conceptos permiten a los desarrolladores modelar sistemas complejos dividiendo responsabilidades entre objetos que interactúan entre sí. Cada objeto encapsula su propio estado y comportamiento, facilitando una clara separación de preocupaciones dentro del código y promoviendo el diseño de software reutilizable y extensible.

En el desarrollo de videojuegos, la aplicación de POO es particularmente valiosa, ya que ayuda a gestionar de manera eficiente las diferentes entidades del juego, como los personajes, enemigos, y el entorno.

Este informe se centra en la implementación de un videojuego basado en libGDX llamado 'Come Galletas', el cual es es una modificación del juego base 'Game Lluvia', el cual sigue los principios de POO y se estructura en varias clases que representan diferentes elementos del juego. A lo largo del desarrollo de este proyecto, se han aplicado técnicas avanzadas de POO como la encapsulación y el diseño modular para asegurar la escalabilidad y mantenibilidad del código. Además, se analiza el uso de clases abstractas, interfaces y principios de diseño como el Principio de Responsabilidad Única (SRP) y la Inversión de Dependencias (DIP), con el objetivo de crear un sistema robusto y extensible.

En las secciones siguientes, se detallarán los elementos clave de la implementación, incluyendo el diseño de las clases, los diagramas UML correspondientes y un análisis técnico de las funcionalidades actuales del juego. Además, se discutirán las mejoras realizadas, que incluyen la introducción de diferentes fondos para las pantallas de menú, juego y fin del juego, y cómo estas modificaciones cumplen con los principios de diseño orientados a objetos.

2. Explicación del Juego desde la Perspectiva del Jugador (GM 1.1)

El videojuego 'Come Galletas' sigue el estilo de su versión anterior (Juego base) 'Game Lluvia' pero con modificaciones, donde el jugador controla a un perro con el objetivo de recolectar la mayor cantidad de galletas posible mientras esquiva bombas que aparecen en su camino. La jugabilidad es sencilla pero envolvente, manteniendo al jugador atento y activo a lo largo de la partida.

2.1. Objetivos y Narrativa

El objetivo principal del jugador es obtener la mayor cantidad de puntos posibles. Estos puntos se consiguen recolectando galletas y, en algunas situaciones, botiquines. Sin embargo, el reto está en evitar las bombas que, al hacer contacto, reducen las vidas del perro. El juego termina cuando el jugador pierde todas sus vidas.

2.1.1. Elementos Clave del Juego

A continuación se describen los elementos principales que forman parte de la mecánica del juego:

- **Galletas**: Recolectar una galleta otorga al jugador una cantidad estándar de puntos. Son el elemento principal para acumular puntaje.
- **Bombas**: Las bombas son obstáculos que el jugador debe esquivar. Si el perro entra en contacto con una bomba, pierde una vida.
- **Botiquines**: Los botiquines otorgan vidas adicionales si el perro tiene menos de tres vidas. Si el jugador ya tiene el máximo de vidas, el botiquín no otorga vidas adicionales, pero a cambio proporcionará una cantidad considerable de puntos extra, mayor de lo que se puede obtener juntando las galletas.

2.2. Mecánicas del Juego

El juego presenta un equilibrio entre acción y estrategia, ya que el jugador no solo debe esquivar obstáculos y recolectar galletas, sino también gestionar el sistema de vidas y puntos. Algunas de las principales mecánicas son:

2.2.1. Control del Personaje

El personaje principal, un perro, es controlado por el jugador usando opciones de control:

- **Teclado**: Usando las teclas de dirección (flechas). Se escoge esta combinación de teclas al ser la más intuitiva para el usuario, ya que las flechas indican movimiento.

2.2.2. Sistema de Vidas

El perro comienza la partida con tres vidas. Cada vez que el jugador toca una bomba, pierde una vida. Si se recolecta un botiquín mientras tiene menos de tres vidas, recupera una vida. Si el jugador ya tiene el máximo de vidas, recolectar un botiquín otorga puntos adicionales en lugar de más vidas, lo que añade una capa estratégica a la recolección de objetos.

2.2.3. Sistema de Puntos

El puntaje se acumula principalmente recolectando galletas. Adicionalmente, los botiquines también otorgan puntos. Si el perro tiene las tres vidas completas al recolectar un botiquín, el jugador recibe más puntos que por una galleta, recompensando al jugador por administrar sus recursos eficientemente.

2.2.4. Aumento de la Dificultad

A medida que avanza el tiempo en el juego, la dificultad aumenta con la velocidad en la aparición de objetos, los cuales son distribuidos de manera aleatoria y a veces en ubicaciones más complicadas. Asimismo, los objetos como galletas y botiquines pueden comenzar a aparecer en zonas más difíciles de alcanzar, lo que obliga al jugador a planificar sus movimientos con mayor precisión.

2.3. Interacción y Controles

El sistema de controles es simple y es especificado en la pantalla previa al juego, lo que facilita el acceso a una amplia audiencia, pero la introducción de diferentes objetos (como galletas, bombas y botiquines) añade profundidad al juego.

Los controles permiten que el jugador desplace al perro de lado a lado en la pantalla, eludiendo bombas y buscando objetos beneficiosos. El jugador debe estar en constante movimiento para maximizar su puntaje y evitar peligros.

2.4. Pantallas del Juego

El juego se divide en varias pantallas que guían al jugador a través de la experiencia:

- **Menú Principal**: Pantalla inicial donde el jugador puede comenzar una nueva partida y donde se detallan las teclas de movimiento.

Figura 1: Pantalla de Menú Principal

- **Pantalla de Juego**: Es la pantalla principal donde ocurre la acción. Aquí el perro (destacado en azul) recoge galletas, esquiva bombas, obtiene vidas (destacado en rojo) y gestiona su puntaje (destacado en verde). Esto se realiza utilizando las flechas de movimiento del jugador.

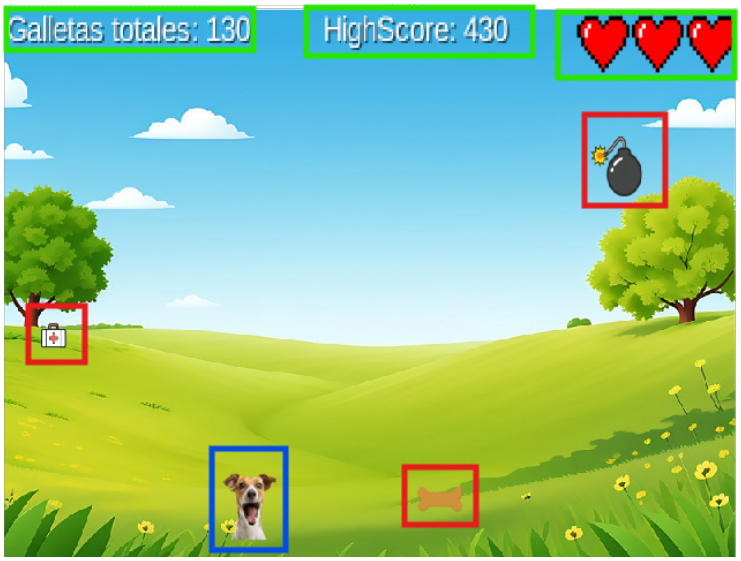

Figura 2: Pantalla de Juego

- **Pantalla de Pausa**: Esta pantalla solo aparece cuando la partida está en curso y el jugador presiona la tecla ESC. Muestra una imagen de pausa e indica que con ESC se vuelve al juego.

Figura 3: Pantalla de Pausa

- **Pantalla de Game Over**: Aparece cuando el jugador pierde todas sus vidas. En esta pantalla se muestra el puntaje final y el jugador puede decidir si quiere volver a jugar o salir del juego.

Figura 4: Pantalla de Game Over

2.5. Aspectos Visuales y Ambientación

El estilo visual del juego es amigable y colorido, con un fondo animado que refuerza el carácter divertido del juego. Los elementos clave como las galletas y los botiquines son de colores brillantes para destacar entre los peligros, como las bombas, que están diseñadas para ser claramente identificables y diferenciadas de los objetos beneficiosos.

2.6. Progresión y Dificultad

A medida que el jugador avanza, la dificultad aumenta de manera progresiva, cabe destacar que al principio el juego es muy facil, volviendose de esta manera, más amigable a cualquier usuario. Se introducen a mayor velocidad bombas, galletas y botiquines haciendo al juego cada vez más desafiante. Esta progresión gradual mantiene al jugador en constante alerta y lo motiva a seguir mejorando su puntaje, creando un equilibrio entre habilidad y estrategia.

3. Análisis del Juego desde la Perspectiva del Ingeniero de Software (GM 1.2)

El juego 'Come Galletas' ha sido desarrollado utilizando el framework libGDX, que facilita la creación de videojuegos multiplataforma. Desde el punto de vista del ingeniero de software, este proyecto tiene una arquitectura modular, con pantallas separadas que manejan diferentes fases del juego (menú principal, pantalla de juego, pantalla de pausa y pantalla de Game Over). A continuación, se detalla el análisis de la implementación actual del juego base y las modificaciones propuestas para mejorar su funcionalidad y estructura.

3.1. Funcionalidades originales

El juego base ya cuenta con las siguientes funcionalidades implementadas:

- **Menú principal**: Ofrece al jugador la posibilidad de iniciar una nueva partida. Este menú es la primera pantalla que el jugador ve al iniciar el juego.
- **Pantalla de juego**: Es la pantalla donde ocurre la mayor parte de la acción. Aquí se implementan las mecánicas principales de recolección de objetos y mantener el control de las vidas.
- **Pantalla de Game Over**: Esta pantalla se muestra cuando el jugador pierde todas sus vidas. Muestra el puntaje final y permite al jugador decidir si desea reiniciar el juego.

3.2. Framework Utilizado

El juego ha sido desarrollado con libGDX, una librería de código abierto que permite el desarrollo de videojuegos en múltiples plataformas (Windows, Android, iOS, HTML5) sin la necesidad de realizar grandes cambios en el código base. libGDX proporciona una infraestructura sólida para gestionar gráficos, entrada de usuario, físicas y audio, lo cual facilita la creación de un entorno de juego interactivo y optimizado.

Entre las ventajas de utilizar libGDX para este proyecto están:

- **Multiplataforma**: Permite exportar el juego a distintas plataformas, maximizando el alcance potencial del mismo.
- **Abstracción de complejidades técnicas**: libGDX se encarga de manejar aspectos técnicos como el renderizado, la entrada de dispositivos y la gestión de recursos.
- **Modularidad**: La arquitectura de libGDX permite separar la lógica del juego en diferentes pantallas y manejar el ciclo de vida de manera clara y eficiente.

3.3. Modificaciones Propuestas

Para mejorar la jugabilidad y la estructura del código, se proponen las siguientes modificaciones:

3.3.1. Adición de fondos de pantalla personalizados

Se añadirá un fondo específico para cada una de las pantallas principales: menú principal, pantalla de juego, pantalla de pausa y pantalla de Game Over. Estos fondos servirán para darle una estética más atractiva al juego y ayudarán a que el jugador distinga fácilmente en qué parte del flujo de juego se encuentra.

Justificación técnica: Los fondos añadirán un nivel de personalización visual que hará que el juego sea más inmersivo. Además, se utilizarán texturas adaptadas para cada resolución de pantalla, haciendo que el juego sea visualmente consistente.

3.3.2. Sistema de recuperación de vidas

Se implementará un sistema de botiquines que el jugador puede recolectar durante la partida. Estos botiquines tendrán dos tamaños: pequeño y grande. Si el jugador tiene menos de tres vidas, el botiquín pequeño le otorgará una vida, mientras que el grande recuperará todas las vidas. Si el jugador ya tiene las tres vidas, en vez de recibir vidas adicionales, se le otorgarán puntos extra (50 puntos para el botiquín pequeño y 100 puntos para el grande).

Justificación técnica: Esta mecánica introduce una capa adicional de estrategia en el juego. El jugador debe gestionar eficientemente sus vidas mientras decide si vale la pena recoger más botiquines por los puntos extra, agregando más profundidad a la jugabilidad.

3.3.3. Mejora en la movilidad del personaje

Se introducirá una funcionalidad que permite que el personaje se desplace continuamente de un borde de la pantalla al otro. Esto significa que si el perro sale del borde derecho de la pantalla, aparecerá automáticamente por el borde izquierdo, y viceversa.

Justificación técnica: Esto permitirá un flujo de juego más dinámico y continuo, evitando que el jugador quede atrapado en los bordes de la pantalla. Además, añade un elemento estratégico donde el jugador puede planificar movimientos rápidos de un lado a otro.

3.3.4. Pantalla de pausa

Se incluirá una pantalla de pausa que permita al jugador detener temporalmente el juego. Esta pantalla debe ser accesible mediante el botón ESC y permitir al jugador retomar el juego.

Justificación técnica: Esta funcionalidad mejora la experiencia del usuario al ofrecerle control sobre la dinámica del juego, permitiéndole pausarlo en cualquier momento sin perder el progreso.

3.3.5. Mejoras visuales generales

Se implementarán mejoras visuales que incluyen animaciones distintas en pantalla para los objetos que en su primera versión fueron gotas de lluvia.

Justificación técnica: Estas mejoras harán que el juego sea visualmente más atractivo y profesional, esto refuerza la inmersión del jugador y mejora la experiencia de juego en general.

3.3.6. Refactorización del código

Se realizará una refactorización para adoptar el patrón de diseño Modelo-Vista-Controlador (MVC), separando la lógica de negocio del juego (Modelo) de la representación visual (Vista) y la interacción del usuario (Controlador).

Justificación técnica: El patrón MVC permitirá una mejor organización del código, facilitando futuras modificaciones o expansiones del juego sin afectar otras partes del sistema. Esto también mejorará la mantenibilidad del proyecto a largo plazo.

3.4. Cambios Técnicos en la Estructura del Código

3.4.1. Nuevas Clases

Se añadirán nuevas clases para manejar las mecánicas relacionadas con los botiquines y la pantalla de pausa. Por ejemplo, se creará una clase `Botiquin` que gestionará las propiedades y comportamientos de los botiquines.

3.4.2. Herencia y Abstracción

Se introducirá una clase base para todas las pantallas del juego, que manejará funcionalidades comunes como la transición entre pantallas o la configuración inicial. Esto evitará la duplicación de código y facilitará el mantenimiento del proyecto.

También se usarán clases abstractas para definir la estructura de las diferentes pantallas, lo que permitirá una mayor extensibilidad a la hora de añadir nuevas fases o niveles al juego en el futuro.

3.4.3. Interfaz `Collectible`

Para asegurar que todos los objetos que se pueden recolectar (como los botiquines) implementen la misma funcionalidad, se ha diseñado la interfaz `Collectible`. Esta interfaz define un método `collect` que permite que cualquier objeto implementador pueda ser recogido por el jugador, asegurando un comportamiento uniforme.

Justificación técnica: La implementación de esta interfaz permite añadir nuevos objetos coleccionables en el futuro sin alterar el comportamiento general del juego, facilitando su extensibilidad.

3.4.4. Principios de Programación Orientada a Objetos (POO)

Se han aplicado los principios de la POO como la encapsulación y la separación de responsabilidades:

- **Encapsulación**: Los atributos como las vidas, los puntos y las áreas de colisión del personaje están encapsulados en sus respectivas clases, y solo son accesibles a través de métodos específicos.
- **Separación de Responsabilidades (SRP)**: Cada clase tiene una única responsabilidad, lo que mejora la mantenibilidad y legibilidad del código. Por ejemplo, la clase `GameScreen` se encarga solo de la lógica del juego, mientras que `MainMenuScreen` gestiona el menú principal.

Justificación técnica: Estos principios garantizan que el código sea fácil de mantener y de escalar en el futuro, ya que cualquier cambio o mejora se puede realizar de manera aislada sin afectar otras partes del sistema.

4. Diseño de Diagrama UML con Clases del Dominio del Juego y su Código en Java (GM 1.3)

El diseño de un diagrama UML es crucial para comprender y visualizar la estructura de un sistema basado en la Programación Orientada a Objetos (POO). Este diagrama permite a los desarrolladores y a otros interesados tener una representación clara de las clases, sus relaciones y cómo interactúan entre sí dentro del sistema. En el contexto del juego *Come Galletas*, se han modelado diferentes clases y relaciones que reflejan las mecánicas del juego y cómo se organiza la lógica del sistema. A continuación se presenta un análisis detallado del diseño UML aplicado a este proyecto, destacando las clases principales, la herencia, la abstracción, las interfaces y la implementación general.

4.1. Clases Principales

El diseño UML del proyecto *Come Galletas* incluye varias clases clave que controlan tanto la lógica del juego como las interacciones visuales y de usuario. Cada clase tiene una responsabilidad específica, lo que respeta el Principio de Responsabilidad Única (SRP), un principio fundamental en el diseño de software. A continuación, se describen las clases más relevantes:

- **GameLluviaMenu**: Es la clase principal que hereda de `Game`, y se encarga de iniciar y manejar el flujo del juego. Esta clase controla las transiciones entre las diferentes pantallas (`MainMenuScreen`, `GameScreen`, `PausaScreen`, y `GameOverScreen`). A través de `SpriteBatch` y `BitmapFont`, maneja la lógica común del juego, como el dibujo de gráficos y la gestión de texto.

- **Tarro**: Representa al personaje principal controlado por el jugador. Esta clase se encarga de la actualización de movimiento del personaje, el control de vidas y la interacción con los elementos recolectables y obstáculos. Su encapsulamiento de atributos como vidas y puntos, junto con métodos como `dañar()` y `sumarVida()`, garantizan la integridad de los estados del juego.

- **Botiquin** (y subclases `BotiquinGrande` y `BotiquinPequeno`): Estas clases son responsables de manejar los botiquines que el jugador puede recolectar. `Botiquin` es una clase abstracta que encapsula las propiedades y comportamientos comunes de los botiquines, mientras que sus subclases definen comportamientos específicos para otorgar puntos o recuperar vidas.

- **Lluvia**: Modela la mecánica de las gotas de lluvia que caen en la pantalla y los obstáculos que el jugador debe esquivar. Esta clase controla tanto las gotas 'buenas' como las 'malas' y define la lógica para aumentar la dificultad del juego conforme el jugador avanza.

4.2. Herencia y Polimorfismo

El diagrama UML refleja el uso de herencia y polimorfismo en el diseño del juego. El uso de herencia se observa en la relación entre la clase abstracta `Botiquin` y sus subclases `BotiquinGrande` y `BotiquinPequeno`. Ambas subclases heredan de `Botiquin`, pero implementan sus propios comportamientos en el método `aplicarEfecto()`. El polimorfismo permite que estas subclases se traten de manera uniforme en el código, utilizando la referencia general `Botiquin`, lo que mejora la extensibilidad del sistema y facilita la incorporación de nuevos tipos de botiquines en el futuro.

Además, se implementa una interfaz, `Collectible`, que define el método `collect()`. Esta interfaz asegura que cualquier objeto recolectable (como botiquines) pueda integrarse sin modificar el código base. El uso de la interfaz facilita la futura expansión del sistema sin necesidad de alterar el comportamiento de las clases ya existentes, lo cual es una muestra clara del Principio Abierto/Cerrado (OCP).

4.3. Relaciones entre Clases

El diagrama UML del juego también resalta las relaciones de agregación y composición entre las clases:

- **Agregación**: La clase `GameScreen` tiene una relación de agregación con las clases `Tarro`, `Lluvia` y `Botiquin`, ya que instancia y utiliza estos objetos para manejar la lógica de juego. Aunque `GameScreen` depende de estos objetos, ellos pueden existir independientemente fuera del contexto de `GameScreen`.
- **Composición**: La clase `Lluvia` tiene una relación de composición con las gotas de lluvia que maneja (`rainDropsPos`). Esto significa que las gotas de lluvia no pueden existir fuera de la instancia de `Lluvia`, y cuando se destruye una instancia de `Lluvia`, todas las gotas desaparecen.

4.4. Jerarquía de Pantallas (Screens)

El juego está dividido en varias pantallas que heredan de la clase `Screen` de libGDX, lo cual facilita la separación de responsabilidades y la gestión de los diferentes estados del juego. La clase `GameLluviaMenu` maneja la creación y el cambio entre estas pantallas:

- **MainMenuScreen**: Pantalla de inicio que permite al usuario comenzar el juego.
- **GameScreen**: La pantalla principal donde ocurre la jugabilidad.
- **PausaScreen**: Pantalla de pausa que se muestra cuando el jugador decide detener temporalmente el juego.

- **GameOverScreen**: Pantalla que aparece cuando el jugador ha perdido todas sus vidas y el juego ha terminado.

Cada pantalla tiene su propio ciclo de vida, con métodos como `show()`, `render()` y `dispose()`. Esta estructura permite que cada pantalla maneje su propia lógica y recursos gráficos, mejorando la mantenibilidad del código y facilitando la adición de nuevas pantallas en el futuro.

4.5. Principios de Diseño Aplicados

A lo largo del diagrama UML y la implementación del código, se aplican varios principios de diseño de software que aseguran que el juego sea mantenible, extensible y robusto:

- **Encapsulamiento**: Los atributos de las clases, como las vidas y puntos del jugador, están protegidos mediante métodos `getters` y `setters`, lo que asegura que estos valores solo puedan modificarse de manera controlada.
- **Abstracción**: Las clases abstractas y las interfaces se utilizan para definir comportamientos generales que pueden ser implementados de manera concreta por subclases o clases que implementan interfaces.
- **Separación de Responsabilidades (SRP)**: Cada clase tiene una única responsabilidad. Por ejemplo, `Tarro` se encarga del personaje del jugador, mientras que `Lluvia` maneja la mecánica de las gotas que caen, y las pantallas (`Screen`) se encargan de la navegación entre las diferentes etapas del juego.

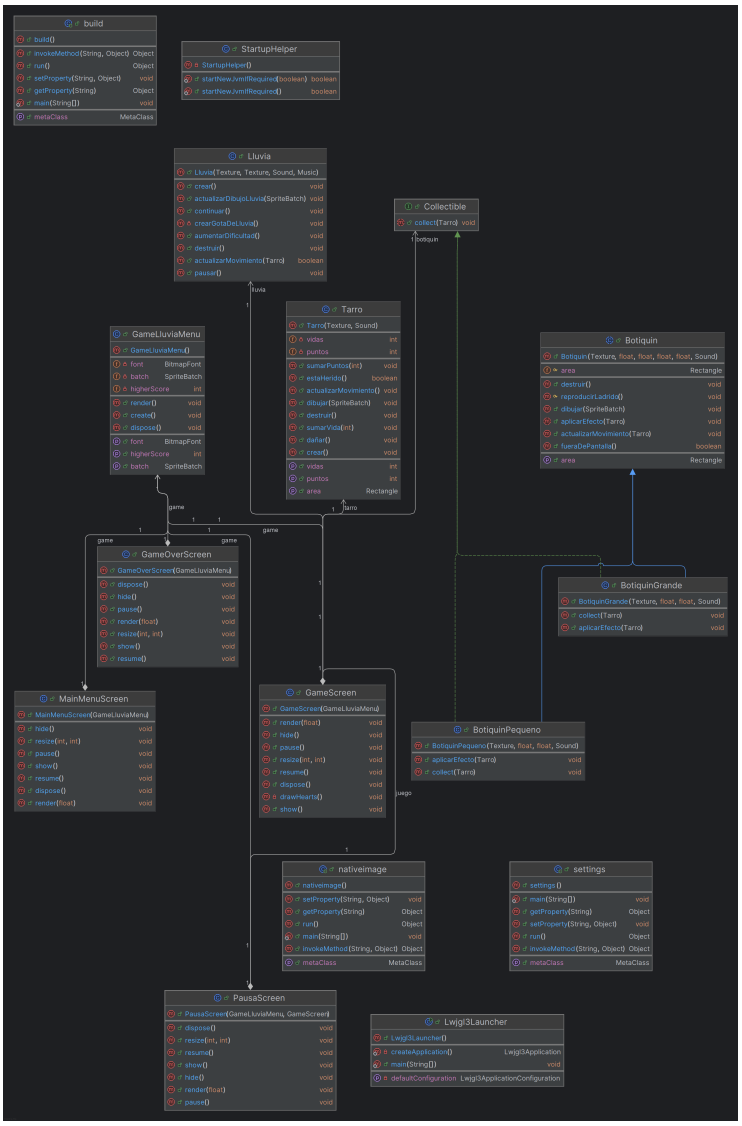

Figura 5: Diagrama UML del Juego *Come Galletas*

En el diagrama se muestran las relaciones entre las clases y cómo interactúan dentro del framework libGDX. Cada clase tiene una responsabilidad específica y maneja diferentes aspectos del juego, como las pantallas principales, la lógica de juego, y las interacciones entre los objetos recolectables y obstáculos. Este diseño facilita la comprensión y modificación del sistema, permitiendo una clara separación de las responsabilidades y una organización eficiente de la lógica y los elementos visuales del juego.

5. Diseño y Codificación de Clase Abstracta (GM 1.4)

En el desarrollo del juego *Come Galletas*, se implementa la clase abstracta `Botiquin` con el objetivo de manejar los botiquines que el jugador puede recoger en la partida. La clase abstracta permite definir comportamientos y atributos compartidos entre los diferentes tipos de botiquines, tales como el movimiento de caída y el área de colisión, evitando duplicación de código y facilitando la extensibilidad.

5.1. Necesidad de la Clase Abstracta Botiquin

La clase abstracta `Botiquin` fue diseñada para unificar las funcionalidades comunes a todos los botiquines en el juego, de modo que cada variante pueda implementar solo las diferencias específicas. Algunos comportamientos comunes incluyen:

- **Movimiento descendente**: Todos los botiquines caen a la misma velocidad y posición relativa. El método `actualizarMovimiento()` de la clase abstracta facilita esta acción.
- **Colisión y eliminación**: `Botiquin` define el área de colisión y controla la eliminación del objeto cuando este sale de la pantalla.

5.2. Método Abstracto y Sobrescritura

La clase `Botiquin` incluye el método abstracto `aplicarEfecto(Tarro tarro)`, que es sobrescrito en sus subclases. Esto permite que cada tipo de botiquín aplique un efecto específico al jugador al ser recogido.

```
1 @Override
2 public void aplicarEfecto(Tarro tarro) {
3     // Efecto espec fico de cada subclase, como sumar vida
          o puntos
4 }
```

Listing 1: Método aplicarEfecto() en la clase abstracta Botiquin

5.3. Subclases de Botiquin

Las subclases `BotiquinPequeno` y `BotiquinGrande` implementan `aplicarEfecto()` para definir comportamientos únicos:

- **BotiquinPequeno**: Aumenta una vida si el jugador tiene menos de tres, o da 50 puntos si ya tiene el máximo.
- **BotiquinGrande**: Restaura todas las vidas hasta un máximo de tres o da 100 puntos si el jugador ya tiene el máximo.

6. Diseño y Codificación de una Interfaz (GM 1.5)

En el proyecto *Come Galletas*, se implementa la interfaz `Collectible` para unificar el comportamiento de los objetos recolectables. La interfaz define el método `collect(Tarro tarro)`, permitiendo que cada clase que la implemente defina el efecto específico que produce al ser recogida.

6.1. Funcionalidad y Contexto de Uso

La interfaz `Collectible` es implementada por las clases `BotiquinPequeno` y `BotiquinGrande`, que representan distintos tipos de botiquines en el juego. Cada clase sobrescribe el método `collect()` para aplicar un efecto específico al jugador: recuperación de vidas o acumulación de puntos.

En el contexto del juego, `Collectible` se utiliza en la clase `GameScreen`. Cuando el jugador recoge un botiquín, `GameScreen` llama al método `collect()` de dicho botiquín, aplicando su efecto sobre el jugador.

```
1 public interface Collectible {
2     void collect(Tarro tarro);
3 }
```

Listing 2: Código de la Interfaz Collectible

6.2. Sobrescritura del Método collect() en BotiquinPequeno y BotiquinGrande

```
1 @Override
2 public void collect(Tarro tarro) {
3     if (tarro.getVidas() < 3) tarro.sumarVida(1);
4     else tarro.sumarPuntos(50);
5 }
```

Listing 3: Ejemplo de Sobrescritura de collect()

6.3. Cumplimiento de los Requisitos

- **Diseño justificado**: `Collectible` estandariza los objetos recolectables.
- **Implementación en dos clases**: `BotiquinPequeno` y `BotiquinGrande` implementan `collect()`.
- **Uso en el juego**: `GameScreen` utiliza la interfaz en su interacción con el jugador.

6.4. Justificación Técnica

La interfaz `Collectible` respeta el Principio Abierto/Cerrado (OCP), permitiendo la integración de nuevos objetos recolectables sin alterar el código existente. Esto mejora la extensibilidad y modularidad del juego.

7. Aplicación del Encapsulamiento y Principios de Programación Orientada a Objetos (POO) (GM 1.6)

El diseño del juego *Come Galletas* aplica los principios fundamentales de la Programación Orientada a Objetos (POO), implementando prácticas de encapsulamiento y adhiriéndose a los principios de Responsabilidad Única (SRP), Abierto/Cerrado (OCP), Sustitución de Liskov (LSP), Segregación de Interfaces (ISP), e Inversión de Dependencias (DIP).

7.1. Encapsulamiento

En el proyecto, el encapsulamiento se asegura mediante el uso de modificadores de acceso `private` para los atributos, que son accesibles únicamente a través de métodos específicos. En la clase `Tarro`, los atributos `vidas` y `puntos` están privados y se modifican mediante métodos como `sumarVida()`, manteniendo la integridad del estado del objeto.

```
1 private int vidas;
2 public int getVidas() { return vidas; }
```

Listing 4: Ejemplo de Encapsulamiento en la Clase Tarro

7.2. Principio de Responsabilidad Única (SRP)

Cada clase en el proyecto está diseñada para manejar una única responsabilidad. Por ejemplo, la clase `Tarro` gestiona el personaje controlado por el jugador, mientras que `Botiquin` y sus subclases manejan los objetos coleccionables. Esta separación clara de responsabilidades facilita la mantenibilidad y claridad del código.

7.3. Principio Abierto/Cerrado (OCP)

El principio de Abierto/Cerrado se respeta mediante el uso de herencia y polimorfismo. La clase abstracta `Botiquin` define el comportamiento base, y sus subclases (`BotiquinPequeno` y `BotiquinGrande`) sobrescriben el método `aplicarEfecto()` para definir efectos específicos, permitiendo añadir nuevos tipos de botiquines sin modificar las clases existentes.

7.4. Principio de Sustitución de Liskov (LSP)

Las subclases de `Botiquin` pueden reemplazar a su clase base sin alterar la funcionalidad esperada. Esto asegura que `BotiquinPequeno` o `BotiquinGrande` puedan usarse en cualquier contexto donde se espera un `Botiquin`, cumpliendo así con el principio de Liskov.

7.5. Principio de Segregación de Interfaces (ISP)

La interfaz `Collectible` sigue el principio ISP al definir exclusivamente el método `collect(Tarro tarro)`, requerido por los objetos coleccionables. Esto permite que `BotiquinPequeno` y `BotiquinGrande` implementen únicamente la funcionalidad necesaria.

```
1 public interface Collectible {
2     void collect(Tarro tarro);
3 }
```

Listing 5: Código de la Interfaz Collectible

7.6. Principio de Inversión de Dependencias (DIP)

La clase `Tarro` interactúa con objetos coleccionables a través de la interfaz `Collectible`, asegurando que dependa de abstracciones en lugar de implementaciones concretas, lo que facilita la extensibilidad del sistema.

```
1 public void recoger(Collectible objeto) {
2     objeto.collect(this);
3 }
```

Listing 6: Ejemplo de DIP en la Interacción con Collectible

Este enfoque garantiza que el diseño del código cumpla con los principios de POO, promoviendo la escalabilidad, modularidad y facilidad de mantenimiento del sistema.

8. Aplicación del Patrón de Diseño Singleton (GM 2.1)

8.1. Problema

En el desarrollo del juego *Come Galletas*, surgió la necesidad de gestionar de manera centralizada configuraciones globales como la puntuación máxima, el nivel de dificultad y el estado del audio. La descentralización inicial de estas configuraciones generaba múltiples problemas:

- Redundancia de datos.
- Errores de sincronización entre pantallas.
- Dificultades al aplicar cambios globales de manera eficiente, lo que afectaba la funcionalidad y mantenibilidad del proyecto, complicando la integración de nuevas características.

8.2. Contexto

El proyecto incluye múltiples pantallas (MainMenuScreen, GameScreen, GameOverScreen) que requieren acceso a configuraciones compartidas. Sin un enfoque centralizado, estas configuraciones debían ser manejadas individualmente por cada pantalla, resultando en una lógica duplicada y propensa a errores. Era necesario un mecanismo que garantizara una única fuente de verdad para estas configuraciones, permitiendo que cualquier cambio se propagara de manera inmediata y uniforme.

8.3. Solución

Se implementó el patrón de diseño **Singleton** mediante la clase ConfiguracionJuegoSingleton, garantizando una única instancia centralizada en todo el proyecto. Esta clase:

- Centraliza la gestión de configuraciones clave, como el audio, la puntuación máxima y la dificultad.
- Proporciona métodos accesibles desde cualquier parte del proyecto para configurar y recuperar estas propiedades globales.
- Elimina problemas de sincronización al garantizar que todas las pantallas accedan a la misma instancia.

8.4. Participantes, Roles e Interrelaciones

8.4.1. Clase `ConfiguracionJuegoSingleton`

Rol: Gestionar de manera centralizada las configuraciones globales del juego.
Métodos principales:

- `obtenerInstancia()`: Devuelve la única instancia del Singleton.
- `configurarPuntuacionMaxima(int puntuacion)`: Actualiza la puntuación máxima solo si el nuevo valor es mayor.
- `configurarNivelDificultad(int nivel)`: Define el nivel de dificultad.
- `configurarAudioActivado(boolean activado)`: Activa o desactiva el audio.
- `obtenerPuntuacionMaxima()`, `obtenerNivelDificultad()`, `audioEstaActivado()`: Recuperan los valores configurados.

8.4.2. Pantallas del Juego (`MainMenuScreen`, `GameScreen`, `GameOverScreen`)

Rol: Consumir los datos centralizados del Singleton para sincronizar configuraciones.

- `MainMenuScreen`: Permite al jugador modificar configuraciones, como la dificultad o el audio, actualizando el Singleton.
- `GameScreen`: Consulta el Singleton para ajustar la dificultad, reproducir sonidos y actualizar la puntuación.
- `GameOverScreen`: Recupera la puntuación máxima desde el Singleton para mostrarla al jugador.

8.4.3. Clases Auxiliares (`Lluvia`, `Tarro`, `Botiquin`)

Rol: Consultar el estado del Singleton para garantizar que las configuraciones globales (como el audio) se respeten durante la ejecución del juego.

8.5. Aplicación en el Proyecto

La clase `ConfiguracionJuegoSingleton` se integró exitosamente, proporcionando un punto único de control para las configuraciones clave del juego. Los beneficios principales incluyen:

- **Consistencia**: Todas las pantallas y componentes acceden a la misma instancia, eliminando errores de sincronización.
- **Mantenibilidad**: Al centralizar la lógica, se reduce la duplicación de código y se facilita la introducción de nuevas características.
- **Extensibilidad**: La solución permite agregar nuevas configuraciones globales sin alterar el flujo existente.

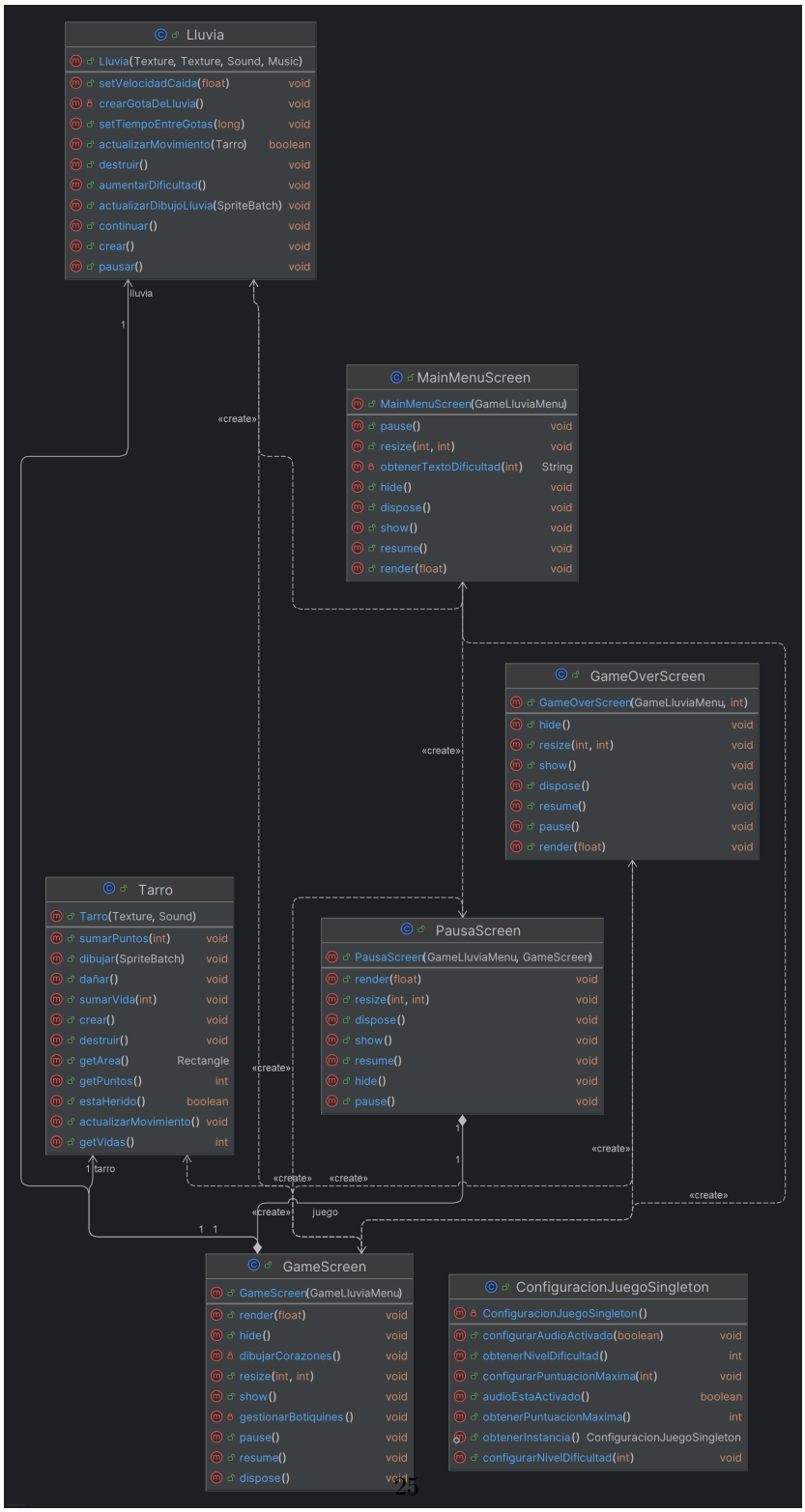

Figura 6: Diagrama UML del Patrón Singleton aplicado en *Come Galletas*

En este diagrama UML se muestra cómo se aplica el patrón de diseño Singleton mediante la clase `ConfiguracionJuegoSingleton`, que centraliza la gestión de configuraciones globales como la puntuación máxima, el nivel de dificultad y el estado del audio. También se presentan las relaciones con otras clases clave del juego, incluyendo las pantallas (`MainMenuScreen`, `GameScreen`, `GameOverScreen`, `PausaScreen`) y las clases auxiliares (`Lluvia`, `Tarro`), que interactúan con el Singleton para sincronizar y respetar las configuraciones globales. Este diseño garantiza la consistencia, extensibilidad y mantenibilidad del sistema.

9. Aplicación del Patrón de Diseño Template Method (GM 2.2)

9.1. Problema

En el desarrollo del juego *Come Galletas*, los objetos que caen (como los botiquines) requerían un comportamiento compartido, pero con variaciones específicas dependiendo del tipo de objeto. La implementación inicial duplicaba la lógica común en cada clase concreta, lo que generaba los siguientes problemas:

- Redundancia de código en las clases relacionadas con los objetos que caen.
- Dificultades para agregar nuevos tipos de objetos que caen, debido a la falta de una estructura general.
- Incremento en la complejidad y mantenimiento del proyecto al no reutilizar el flujo común.

9.2. Contexto

En el juego, los objetos que caen comparten una lógica general, como moverse, dibujarse en pantalla, detectar colisiones con el tarro y aplicar efectos. Sin embargo, cada tipo de objeto tenía variaciones específicas, como diferentes movimientos o efectos al colisionar. Esto hizo evidente la necesidad de un mecanismo que permitiera centralizar el flujo común y delegar los detalles específicos a subclases.

9.3. Solución

Se implementó el patrón de diseño **Template Method** mediante la creación de la clase abstracta `ObjetoCayendo`, que define el flujo general de los objetos que caen y delega los comportamientos específicos a las subclases. Esta implementación:

- Centraliza la lógica común en una clase base abstracta.
- Define un flujo general, como actualizar movimiento, dibujar, verificar colisiones y aplicar efectos, dejando los detalles específicos en las subclases.
- Facilita la extensión del sistema al permitir agregar nuevos tipos de objetos con facilidad.

9.4. Participantes, Roles e Interrelaciones

9.4.1. Clase `ObjetoCayendo`

Rol: Proporcionar un esqueleto para la lógica general de los objetos que caen, dejando las personalizaciones específicas a las subclases.

Métodos principales:

- `actualizarMovimiento(float deltaTime, boolean tarroHerido)`: Llama al método `mover`, gestionando el movimiento dependiendo del estado del tarro.
- `dibujar(SpriteBatch batch)`: Renderiza el objeto en pantalla.
- `colisionaConTarro(Tarro tarro)`: Verifica si el objeto colisiona con el área del tarro.
- `aplicarEfecto(Tarro tarro)`: Método abstracto para definir el efecto aplicado al colisionar con el tarro.

9.4.2. Clase `BotiquinPequeno`

Rol: Subclase de `ObjetoCayendo`, implementa un movimiento vertical y aplica efectos específicos, como sumar una vida o puntos.
Métodos principales:

- `mover(float deltaTime)`: Realiza un movimiento vertical simple.
- `aplicarEfecto(Tarro tarro)`: Suma una vida al tarro o puntos si ya tiene el máximo de vidas.

9.4.3. Clase `BotiquinGrande`

Rol: Subclase de `ObjetoCayendo`, con un movimiento personalizado (zigzag) y efectos avanzados, como restaurar todas las vidas o dar más puntos.
Métodos principales:

- `mover(float deltaTime)`: Realiza un movimiento en zigzag.
- `aplicarEfecto(Tarro tarro)`: Restaura todas las vidas del tarro o añade puntos.

9.5. Aplicación en el Proyecto

La implementación del patrón `Template Method` en el proyecto trajo los siguientes beneficios:

- **Reutilización de código**: La lógica común fue centralizada en `ObjetoCayendo`, eliminando duplicación en las subclases.
- **Extensibilidad**: Fue sencillo agregar nuevos tipos de objetos que caen, como el `BotiquinGrande`, personalizando únicamente los métodos abstractos.
- **Claridad**: Se mejoró la organización del código al separar el flujo general del comportamiento específico de cada objeto.

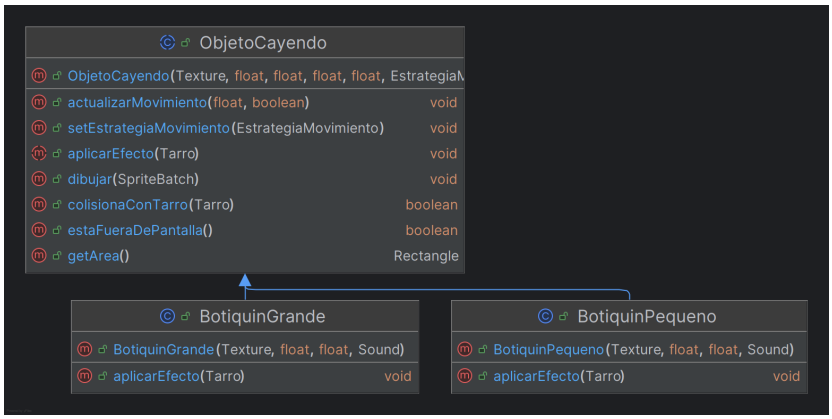

Figura 7: Diagrama UML del Patrón Template Method aplicado en *Come Galletas*

En este diagrama UML se observa la implementación del patrón de diseño Template Method mediante la clase abstracta `ObjetoCayendo`, que centraliza la lógica común de los objetos que caen en el juego. Esta clase define un flujo general para actualizar el movimiento, detectar colisiones y renderizar los objetos en pantalla, delegando los comportamientos específicos al método abstracto `aplicarEfecto()`, implementado por sus subclases `BotiquinPequeno` y `BotiquinGrande`. Este diseño permite una clara separación entre la lógica común y las variaciones específicas, facilitando la extensión del sistema y reduciendo la duplicación de código. Cada subclase define su comportamiento único, como los diferentes efectos aplicados al interactuar con el jugador.

10. Aplicación del Patrón de Diseño Strategy (GM 2.3)

10.1. Problema

En el desarrollo del juego *Come Galletas*, se observó que los objetos que caen (como los botiquines) tenían diferentes patrones de movimiento, como una caída vertical simple o movimientos más complejos, como zigzag. Sin embargo, esta lógica de movimiento estaba acoplada directamente a las clases concretas, lo que generaba los siguientes problemas:

- Falta de flexibilidad para cambiar los patrones de movimiento durante la ejecución del juego.
- Duplicación de código al implementar variaciones similares en diferentes clases concretas.
- Dificultad para escalar el sistema al agregar nuevos patrones de movimiento.

10.2. Contexto

El juego incluye múltiples tipos de objetos que caen, como `BotiquinPequeno` y `BotiquinGrande`, cada uno con su propio comportamiento de movimiento. La implementación inicial acoplaba la lógica de movimiento dentro de estas clases, lo que limitaba la capacidad de reutilizar y modificar estos patrones dinámicamente. Era necesario un enfoque que separara los movimientos en componentes independientes para facilitar la extensión y reutilización.

10.3. Solución

Se implementó el patrón de diseño **Strategy**, separando los patrones de movimiento en clases independientes que pueden ser asignadas dinámicamente a los objetos que caen. Esta implementación:

- Define una interfaz `EstrategiaMovimiento` que abstrae los diferentes patrones de movimiento.
- Implementa estrategias concretas como `MovimientoVertical` y `MovimientoZigzag`, cada una con su propia lógica de movimiento.
- Permite asignar o cambiar la estrategia de movimiento de un objeto en tiempo de ejecución.

10.4. Participantes, Roles e Interrelaciones

10.4.1. Interfaz `EstrategiaMovimiento`

Rol: Definir un contrato para todas las estrategias de movimiento.
Método principal:

- `mover(Rectangle area, float deltaTime, boolean tarroHerido)`: Actualiza la posición del objeto dependiendo de su estrategia.

10.4.2. Clase `MovimientoVertical`

Rol: Implementar un movimiento vertical simple hacia abajo.
Lógica principal:

- Desplaza el objeto hacia abajo a una velocidad constante.
- Detiene el movimiento si el tarro está herido.

10.4.3. Clase `MovimientoZigzag`

Rol: Implementar un movimiento en zigzag, con lógica de rebote en los bordes.
Lógica principal:

- Combina un desplazamiento hacia abajo con un desplazamiento horizontal oscilante.
- Rebota en los bordes de la pantalla al alcanzar los límites.
- Detiene el movimiento si el tarro está herido.

10.4.4. Clase Base `ObjetoCayendo`

Rol: Usar la interfaz `EstrategiaMovimiento` para delegar la lógica de movimiento de los objetos que caen.
Lógica principal:

- Contiene un atributo de tipo `EstrategiaMovimiento`.
- Llama al método `mover` de la estrategia durante la actualización del movimiento.
- Permite cambiar dinámicamente la estrategia de movimiento mediante el método `setEstrategiaMovimiento`.

10.4.5. Clases Concretas (`BotiquinPequeno`, `BotiquinGrande`)

Rol: Asignar una estrategia de movimiento al crearse.

- `BotiquinPequeno`: Utiliza la estrategia `MovimientoVertical`.
- `BotiquinGrande`: Utiliza la estrategia `MovimientoZigzag`.

10.5. Aplicación en el Proyecto

La implementación del patrón `Strategy` en el proyecto trajo los siguientes beneficios:

- **Flexibilidad**: Los patrones de movimiento pueden ser asignados o modificados dinámicamente, permitiendo una mayor variabilidad en el comportamiento de los objetos.
- **Reutilización**: Las estrategias de movimiento son independientes y reutilizables en diferentes tipos de objetos que caen.
- **Extensibilidad**: Es fácil agregar nuevas estrategias de movimiento, como un desplazamiento en espiral, sin modificar el código existente.

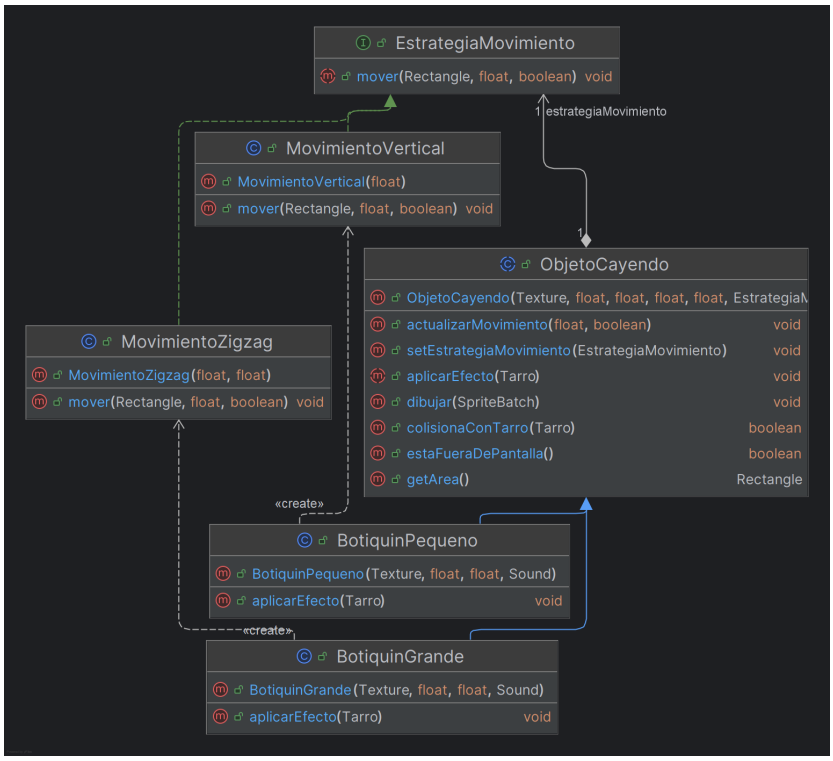

Figura 8: Diagrama UML del Patrón Strategy aplicado en *Come Galletas*

En este diagrama UML se representa la implementación del patrón de diseño Strategy en el proyecto *Come Galletas*. La interfaz `EstrategiaMovimiento` define un contrato para las distintas estrategias de movimiento, implementadas por las clases `MovimientoVertical` y `MovimientoZigzag`. Estas clases encapsulan la lógica de movimiento específica y pueden asignarse dinámicamente a

instancias de `ObjetoCayendo`, lo que incluye las subclases `BotiquinPequeno` y `BotiquinGrande`. Esto permite que cada objeto que cae tenga un comportamiento de movimiento personalizado sin acoplar la lógica a las clases concretas, mejorando la flexibilidad, la reutilización y la extensibilidad del sistema.

11. Aplicación del patrón de diseño Abstract Factory (GM 2.4)

11.1. Problema

En el desarrollo del juego **Lluvia de Botiquines**, se observó que la creación de los diferentes tipos de botiquines (como *BotiquinGrande* y *BotiquinPequeno*) estaba acoplada directamente a la lógica de la clase principal (*GameScreen*). Esto generaba los siguientes problemas:

- **Falta de flexibilidad:** La lógica de creación estaba mezclada con el flujo principal del juego, lo que dificultaba agregar nuevos tipos de botiquines.
- **Duplicación de código:** La lógica de instanciación se repetía en varias partes, creando redundancias innecesarias.
- **Escalabilidad limitada:** Cualquier cambio o adición de nuevos tipos de botiquines requería modificar múltiples partes del código, rompiendo el principio de abierto/cerrado.

11.2. Contexto

El juego incluye dos tipos principales de botiquines:

- **BotiquinGrande:** Otorga más beneficios al jugador y utiliza un movimiento en zigzag.
- **BotiquinPequeno:** Otorga beneficios menores y utiliza un movimiento vertical.

Inicialmente, la creación de estos botiquines estaba gestionada directamente en la clase *GameScreen*, lo que dificultaba su mantenimiento y expansión. Se requería una solución que desacoplara la lógica de creación de los objetos del flujo principal del juego.

11.3. Solución

Se implementó el patrón de diseño **Abstract Factory**, permitiendo gestionar la creación de los diferentes tipos de botiquines de manera desacoplada. Este enfoque incluyó los siguientes pasos:

- Definir una interfaz `BotiquinFactory` que actúa como un contrato para las fábricas concretas.
- Crear fábricas concretas (*BotiquinGrandeFactory* y *BotiquinPequenoFactory*) encargadas de la lógica específica de instanciación de cada tipo de botiquín.
- Modificar la clase *GameScreen* para utilizar las fábricas en lugar de instanciar directamente los objetos, mejorando la flexibilidad y escalabilidad.

Con esta solución, se eliminó la lógica de creación de botiquines de *GameScreen* y se encapsuló en las fábricas, promoviendo un diseño modular y extensible.

11.4. Participantes, Roles e Interrelaciones

11.4.1. Interfaz `BotiquinFactory`

- **Rol:** Definir un contrato para la creación de botiquines.
- **Método principal:** `crearBotiquin(Texture textura, float x, float y, Sound sonido)`: Crea y devuelve un botiquín específico.

11.4.2. Clase `BotiquinGrandeFactory`

- **Rol:** Crear instancias del tipo `BotiquinGrande`.
- **Lógica principal:** Encapsula la lógica de creación de `BotiquinGrande`, asignándole el movimiento zigzag.

11.4.3. Clase `BotiquinPequenoFactory`

- **Rol:** Crear instancias del tipo `BotiquinPequeno`.
- **Lógica principal:** Encapsula la lógica de creación de `BotiquinPequeno`, asignándole el movimiento vertical.

11.4.4. Clase `Botiquin`

- **Rol:** Actuar como clase base para los diferentes tipos de botiquines (`BotiquinGrande` y `BotiquinPequeno`).
- **Lógica principal:**
  - Contiene atributos comunes, como textura, posición y velocidad de caída.
  - Permite que las clases concretas implementen el método abstracto `aplicarEfecto(Tarro tarro)`.

11.4.5. Clase `GameScreen`

- **Rol:** Actuar como cliente que utiliza las fábricas para gestionar la creación de botiquines.
- **Lógica principal:**
  - Selecciona dinámicamente una fábrica (`BotiquinGrandeFactory` o `BotiquinPequenoFactory`) basada en una probabilidad.
  - Utiliza la fábrica seleccionada para instanciar un botiquín y delega el resto de la lógica al objeto creado.

11.5. Beneficios de la Solución

La implementación del patrón **Abstract Factory** proporcionó los siguientes beneficios:

- **Flexibilidad:** Ahora es posible agregar nuevos tipos de botiquines sin modificar la lógica de *GameScreen*.
- **Reutilización:** Las fábricas concretas encapsulan la lógica de creación, facilitando su reutilización en otras partes del proyecto.
- **Mantenibilidad:** Se eliminó la duplicación de código y se redujo el acoplamiento, mejorando la claridad y organización del proyecto.

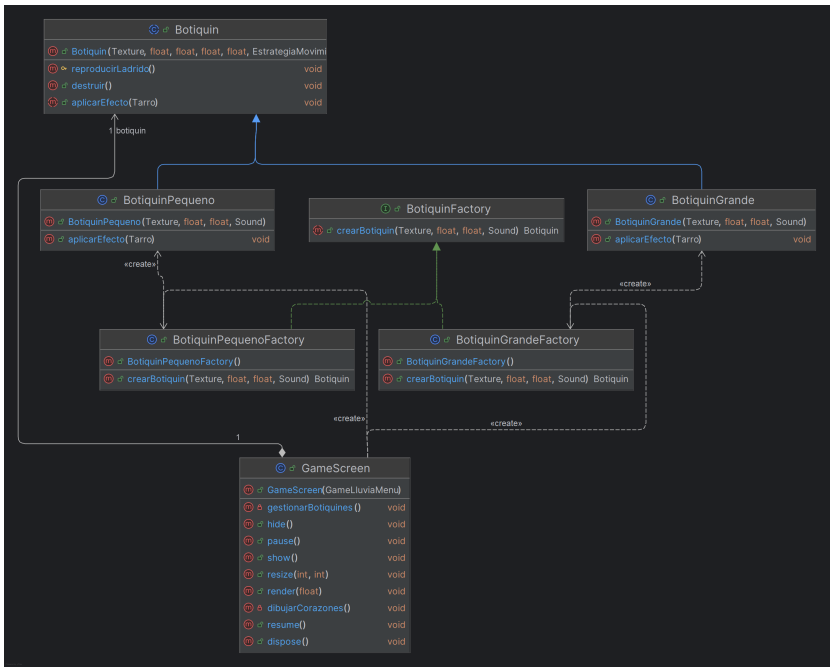

Figura 9: Diagrama UML del Patrón Abstract Factory aplicado en *Lluvia de Botiquines*

En este diagrama UML se ilustra la implementación del patrón de diseño **Abstract Factory** en el proyecto *Lluvia de Botiquines*. La interfaz `BotiquinFactory` actúa como un contrato para las fábricas concretas (`BotiquinPequenoFactory` y `BotiquinGrandeFactory`), que encapsulan la lógica específica de creación de `BotiquinPequeno` y `BotiquinGrande`. Estas fábricas son utilizadas por la clase `GameScreen`, que delega la creación de los botiquines a las fábricas seleccionadas dinámicamente, mejorando la flexibilidad y escalabilidad del sistema. El diseño promueve la reutilización, facilita la adición de nuevos tipos de botiquines y asegura un bajo acoplamiento entre las clases.

Referencias

[1] Link del Repositorio en GitHub: https://github.com/mpardo24/gamemakerGalletas

[2] Link del Repositorio en GitHub para acceso directo a la branch Avance GM 1: https://github.com/mpardo24/gamemakerGalletas/tree/Avance-GM-1

[3] Link de acceso directo al Diagrama UML en github: https://github.com/mpardo24/gamemakerGalletas/blob/Avance-GM-1/Diagrama%20UML.png

[4] LibGDX Framework. Consultado desde: https://libgdx.com/

[5] Programación Orientada a Objetos por Claudio Cubillos. Consultado desde: https://www.youtube.com/@claudioalonsocubillosfigue8881

[6] Programación Orientada a Objetos en Java. Consultado desde: https://www.w3schools.com/java/

[7] Abstract Factory https://refactoring.guru/es/design-patterns/abstract-factory

[8] Singleton https://refactoring.guru/es/design-patterns/singleton

[9] Template Method https://refactoring.guru/es/design-patterns/template-method

[10] Patrón de diseño Builder en Java https://refactorizando.com/patron-builder-java/

[11] Herramienta de diseño de nuestro informe desarrollado en LaTeX https://www.overleaf.com