

# PARADIGMAS Y TÉCNICAS DE PROGRAMACIÓN

Ingeniería Matemática e Inteligencia Artificial

---

## Tower Defense: Hordes of Apocalypse

---

### Informe de Proyecto Final

**Autores:**  
Andrés Gil Vicente  
Jorge Carnicero Príncipe

**Fecha:**  
2 de diciembre de 2025

# Índice

<b>1. Descripción del proyecto</b>	<b>2</b>
1.1. Objetivos . . . . .	2
1.1.1. Cumplimiento de requisitos del proyecto . . . . .	2
1.2. Funcionamiento y Experiencia de Juego . . . . .	2
1.2.1. Navegación y Selección de Nivel . . . . .	3
1.2.2. La Partida: Interfaz y Mecánicas . . . . .	4
1.2.3. Sistema de Torres . . . . .	7
1.2.4. Sistema de Enemigos . . . . .	7
1.3. Esquema General . . . . .	8
1.3.1. Flujo de datos típico en una oleada . . . . .	9
<b>2. Diseño UML de la Arquitectura del Proyecto</b>	<b>10</b>
<b>3. Patrones de Diseño Implementados</b>	<b>11</b>
3.1. Patrón 1: Singleton (Creacional) . . . . .	11
3.2. Patrón 2: Factory (Creacional) . . . . .	11
3.3. Patrón 3: Strategy (Comportamiento) . . . . .	11
3.4. Patrón 4: Observer (Comportamiento) . . . . .	11
3.5. Patrón 5: State (Comportamiento) . . . . .	12
3.6. Patrón 6: Command (Comportamiento) . . . . .	12
3.7. Patrón 7: Facade (Estructural) . . . . .	12

## 1. Descripción del proyecto

A pesar de que aún queda bastante tiempo para entregar el proyecto final completamente desarrollado, hemos avanzado bastante en él por lo que prácticamente lo tenemos terminado. Es por eso que en este documento hemos añadido bastante detalle a nuestras explicaciones sobre el propio proyecto.

### 1.1. Objetivos

El objetivo principal de este proyecto ha sido desarrollar un videojuego de tipo **Tower Defense** en Unity 6000.2.6f2, cumpliendo con los requisitos establecidos en el enunciado del proyecto final de la asignatura. Además, nuestro juego integra un sistema de inteligencia artificial que se va entrenando progresivamente cuando un usuario juega, cuyo propósito es hacer que el propio usuario pierda mediante la generación dinámica y adaptativa de oleadas de enemigos cada vez más difíciles de vencer.

#### 1.1.1. Cumplimiento de requisitos del proyecto

Nuestro proyecto cumple todos los requisitos que se nos especifican en el enunciado:

- **Desarrollo en Unity 6000.2.6f2:** El proyecto ha sido desarrollado íntegramente en la versión especificada de Unity.
- **Inclusión de IA:** Se ha implementado un agente de IA que toma decisiones estratégicas para intentar derrotar al jugador.
- **Funcionamiento de la IA para hacer perder al jugador:** La IA analiza continuamente el rendimiento del jugador y genera oleadas progresivamente más difíciles, ajustando el tipo de enemigos que peor se le dan al usuario para así maximizar la probabilidad de derrota.
- **Loop de juego cerrado:** El juego implementa un ciclo completo (Preparación, Oleada, Victoria/Derrota, Reinicio) con 15 oleadas totales, incluyendo una oleada final con un boss.

Más allá de los requisitos mínimos que se nos pedían, hemos querido dejar el videojuego lo más pulido posible, con detalles y funcionalidades adicionales para que se haga ameno jugarlo y podamos divertirnos realmente mientras lo usamos.

### 1.2. Funcionamiento y Experiencia de Juego

A continuación, describimos el flujo natural que sigue una partida en nuestro proyecto, detallando la experiencia desde la perspectiva del jugador y los elementos visuales que componen la interfaz.

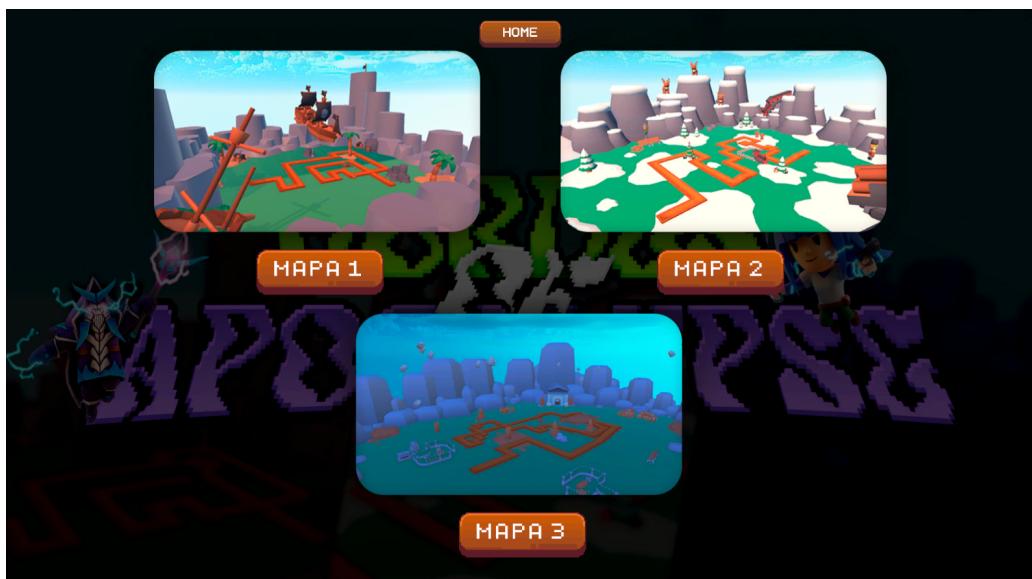
### 1.2.1. Navegación y Selección de Nivel

Lo primero que encontramos al iniciar la aplicación es la **Pantalla de Inicio**. Hemos diseñado una interfaz intuitiva donde se presenta el título del juego junto con los botones fundamentales de 'Jugar' y 'Salir'.



**Figura 1:** Pantalla de inicio del juego.

Al pulsar el botón de jugar, se reproduce una pequeña animación que nos transporta a la **Selección de Mapas**. En esta pantalla, el jugador puede elegir entre los 3 escenarios distintos que hemos diseñado. Cada mapa tiene su propio botón dedicado para acceder directamente, y también contamos con un botón de 'Menú' para regresar a la pantalla anterior.



**Figura 2:** Menú de selección de mapas.

### 1.2.2. La Partida: Interfaz y Mecánicas

Una vez seleccionamos un mapa, entramos en la escena de juego. Aunque internamente todos los sistemas se inicializan, la partida comienza en un estado de espera. El juego no arranca y los enemigos no aparecen hasta que nosotros estemos listos y pulsemos el botón de *Play*.

La interfaz de usuario visible al inicio de la partida nos presenta toda la información vital de un vistazo:



**Figura 3:** Vista inicial del jugador y HUD antes de comenzar la oleada.

- **Panel de Estado (Superior):** En la parte superior central encontramos la barra de vida del jugador (indicando 5/5 vidas) y el contador de oleadas actual, junto con las instrucciones de estado ("Presiona 'Play' para comenzar").
- **Navegación:** En la esquina superior derecha se ubica el botón "HOME", que permite pausar o regresar al menú principal.
- **Inicio de Ronda:** Flotando sobre el personaje, en el centro de la pantalla, aparece el botón "PLAY" para dar comienzo a la partida.
- **Barra de Torres:** En la barra inferior se despliegan las 6 unidades de torres disponibles, mostrando claramente su ícono y su coste económico para facilitar la compra rápida. Además, esta barra se adaptará dinámicamente al dinero que tenga el jugador, desactivando la colocación de ciertas torres cuando no tenga la economía suficiente.



**Figura 1:** Con dinero suficiente



**Figura 2:** Sin dinero (desactivado)

La mecánica principal del juego consiste en colocar torres estratégicamente. Al seleccionar una torre, el juego nos proyecta un círculo visual que indica su **rango de ataque o alcance**, facilitando al usuario la decisión de dónde ubicarla. Además, este rango cambia de color en función de si se puede colocar la torre o no en la posición seleccionada.



**Figura 5:** Indicadores visuales de colocación válida e inválida.

**Dinámica de Oleadas y Mejoras:** El juego alterna entre la defensa activa y pequeños descansos. Tras superar una oleada, disponemos de 10 segundos de preparación, lo cual es indicado mediante audio por el propio juego. Sin embargo, la gestión de la defensa es continua: el jugador puede comprar nuevas torres o mejorar las existentes en cualquier momento, ya sea durante la oleada o durante la pausa.

Al seleccionar una torre ya colocada, accedemos al **Menú de Gestión**. Este panel nos ofrece una comparativa visual directa: a la izquierda las estadísticas actuales (Daño, Rango y Velocidad) y a la derecha los valores que alcanzará la unidad si decidimos invertir en ella.



**Figura 6:** Comparativa de evolución: A la izquierda, una Knight Tower de Nivel 1. A la derecha, la misma torre mejorada al Nivel 5.

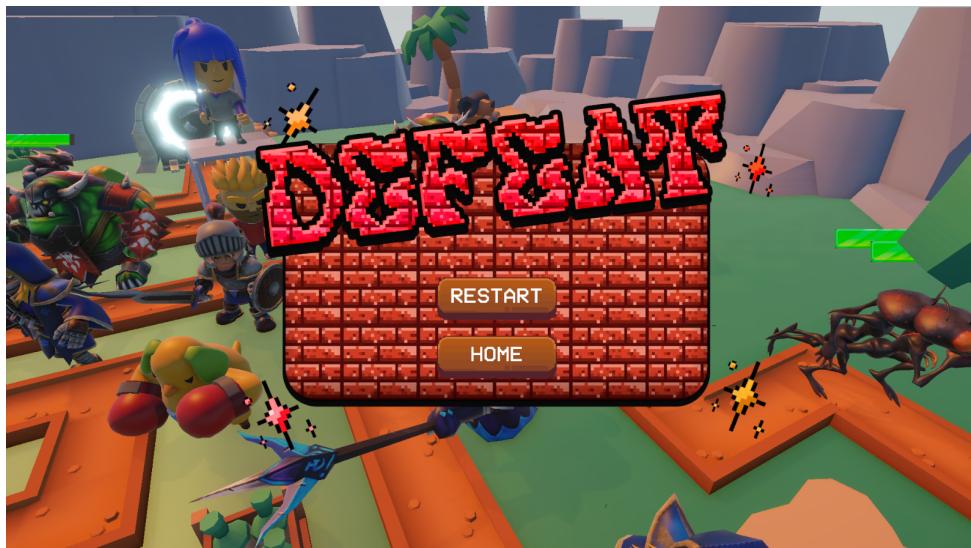
El menú también pone a nuestra disposición tres herramientas de control principales:

- **Botón de Mejora (Upgrade):** Indica el coste necesario para subir de nivel. Se deshabilita en caso de no tener dinero suficiente.
- **Botón de Venta (Sell):** Nos permite retirar la torre del mapa recuperando un porcentaje de la inversión total.
- **Selector de Prioridad:** Nos otorga control sobre la torre, permitiéndonos definir qué enemigo debe ser atacado primero, según la estrategia elegida.



**Figura 7:** Knight Tower nivel máximo y menú de prioridad desplegado.

**Condiciones de Victoria y Derrota:** El objetivo es evitar que los enemigos crucen el mapa. Cada enemigo que llega al final nos resta 1 punto de vida. Si nuestra vida llega a 0, perdemos. En la **Oleada 15** (Jefe final), si este llega a la base, nos quita 5 puntos de vida, haciendo que perdamos directamente.



**Figura 8:** Menú derrota

Para la ambientación del videojuego y cuidado de la experiencia de usuario, hemos integrado efectos de sonido dinámicos, además de muchos otros detalles. Cabe mencionar también, que aunque los modelos 3D provienen de assets externos, la mayoría de la interfaz gráfica (UI) ha sido creada por nosotros en Photoshop para mantener una estética unificada y personalizada.

### 1.2.3. Sistema de Torres

Para ofrecer variedad estratégica, hemos decidido implementar 6 tipos de torres distintas:

1. **Knight Tower:** Diseñada como nuestra unidad estándar y polivalente. Ofrece un equilibrio sólido entre daño, velocidad y rango, siendo ideal para la defensa en las primeras fases del juego.
2. **Orc Tower:** Se enfoca en el daño bruto (DPS alto). A cambio de su potencia, hemos penalizado su velocidad de ataque y limitado su rango a corta-media distancia, haciéndola efectiva en puntos de choque.
3. **Mage Tower:** Nuestra unidad de "francotirador". Destaca por su gran alcance y daño elevado, pensada para cubrir áreas extensas, aunque su alto coste de invocación obliga al jugador a gestionar bien su economía.
4. **Chicken Tower:** Una unidad rápida basada en el concepto de "boxeador". Al combatir cuerpo a cuerpo, su rango es muy reducido, pero lo compensa con una velocidad de ataque rápida y un daño medio constante.
5. **Alien Tower:** Aprovecha una temática tecnológica para ofrecer ventajas tácticas, destacando principalmente por un rango de visión superior que permite atacar enemigos mucho antes que otras torres.
6. **Couple Tower:** Una variante equilibrada similar al Knight. Aunque mantiene prestaciones competentes en todos los aspectos, ofrece una alternativa visual y temática para diversificar las opciones del jugador.



**Figura 9:** Tipos de torres disponibles en el juego.

### 1.2.4. Sistema de Enemigos

Los enemigos que hemos diseñado se dividen en varias categorías según sus estadísticas, intentando complementarse para desafiar constantemente al jugador:

- **Básicos:** Unidades estándar que definen la línea base de dificultad. Tienen un equilibrio medio entre salud y velocidad.

- **Tanques:** Diseñados para actuar como esponjas de daño. Aunque su velocidad de movimiento es reducida, poseen una gran resistencia (menos daño recibido por disparo) y salud.
- **Rápidos:** Enemigos con salud baja pero alta velocidad.
- **Bosses:** Hemos diseñado tres variantes de jefes de gran tamaño para la última ronda. El sistema, mediante la lógica de oleadas (IA), determina cuál de estas variantes es la más eficaz para derrotar al jugador.



**Figura 10:** Tipos de monstruos en el juego.

### 1.3. Esquema General

La arquitectura del proyecto sigue un diseño modular, organizado en capas claramente definidas:

#### Capa de Gestión de Estado (State Management Layer)

- **GameStateManager:** Manager central que coordina el flujo (Singleton).
- **GameEvents:** Sistema de eventos (Observer) para comunicación desacoplada.
- **SceneTransitionManager:** Gestiona transiciones y persistencia entre escenas.
- **SceneInitializer:** Asegura la inicialización correcta de dependencias.

#### Capa de Lógica de Juego (Game Logic Layer)

- **GameLoopManager:** Coordina el tick de enemigos y torres y la aplicación de daño.
- **WaveManager:** Gestiona el sistema de oleadas, spawning y comunicación con IA.
- **PlayerManager:** Controla vida y condiciones de derrota.
- **EconomyManager:** Sistema transaccional con historial y validación.

#### Capa de Entidades (Entity Layer)

- **Enemigos:** EntitySummoner (Factory + Pooling), EnemyHealthManager (Escalado), EnemyPerformanceTracker (Métricas).

- **Torres:** TowerManager, TowerBehaviour, TowerConfigManager, TowerTargeting (Jobs System).

### Capa de Inteligencia Artificial (AI Layer)

- **MLWaveGeneratorAgent:** Agente de ML-Agents que decide la composición de oleadas.
- **MLTrainingManager** y **MLWaveInjector:** Soporte para entrenamiento e inyección de datos.

### Capa de Interfaz (UI Layer)

- **UIManager**, **TowerUpgradeUI**, **TowerButtonAutoConfig**, **TowerRangeVisual**.

#### 1.3.1. Flujo de datos típico en una oleada

El siguiente flujo resume la interacción de sistemas:

1. **GameStateManager** cambia a *Preparing*. **WaveManager** inicia cuenta atrás.
2. **MLWaveGeneratorAgent** recopila observaciones (Vida, Dinero, Historial) y genera la oleada.
3. Jugador compra torres (**TowerPlacement** → **TowerManager** → **EconomyManager**).
4. El estado cambia a *WaveInProgress*. **WaveManager** usa **EntitySummoner** para spawnear enemigos.
5. Bucle de juego:
  - Enemigos se mueven (**EnemyManager**).
  - Torres detectan y atacan (**TowerTargeting** con Jobs System).
  - Se aplica daño centralizado (**GameLoopManager**).
6. Al morir un enemigo: Eventos disparados, recompensa económica, registro de métricas.
7. Al finalizar oleada: Recompensas, evaluación de desempeño de la IA y transición de estado.

## 2. Diseño UML de la Arquitectura del Proyecto

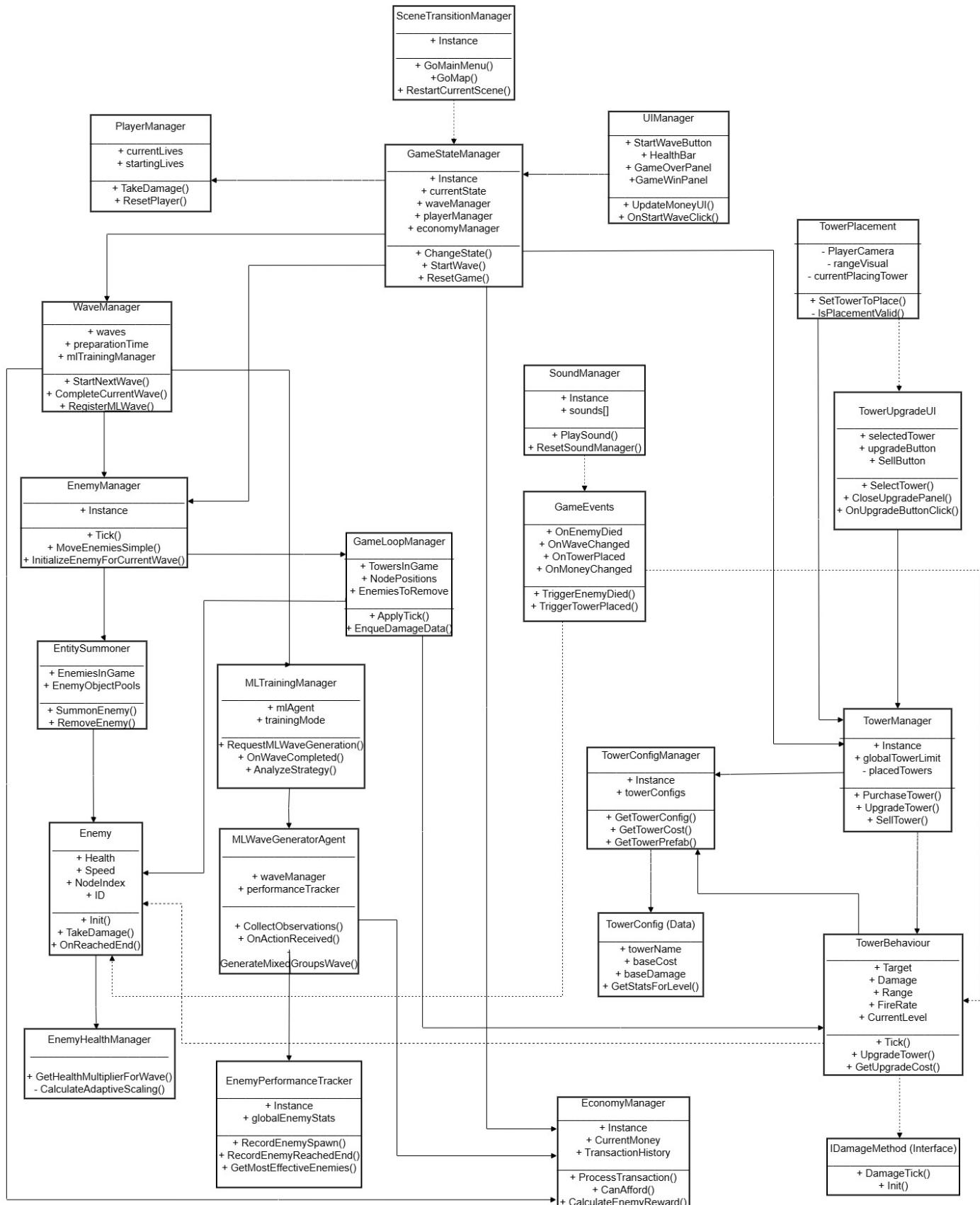


Figura 11: UML - Visión General (Vista ampliada)

### 3. Patrones de Diseño Implementados

En el desarrollo de este proyecto, hemos optado por la implementación de múltiples patrones de diseño para resolver problemas arquitectónicos específicos, buscando siempre el equilibrio entre escalabilidad y rendimiento.

#### 3.1. Patrón 1: Singleton (Creacional)

Hemos implementado este patrón en los gestores principales del sistema, tales como `GameStateManager`, `EconomyManager`, `SoundManager` y algunos managers más.

Esta estructura nos permite garantizar un estado global único, algo vital para manejar la economía y la configuración, además de ofrecernos un acceso conveniente desde cualquier punto del código sin necesidad de pasar referencias, asegurando siempre una inicialización controlada de los sistemas core.

#### 3.2. Patrón 2: Factory (Creacional)

Aplicamos este patrón en la clase `EntitySummoner` a través del método `SummonEnemy(int enemyID)`. Este método actúa como una factoría inteligente que decide si debe instanciar un nuevo enemigo o reutilizar uno existente del pool.

Gracias a esta implementación, logramos desacoplar el sistema de oleadas de la lógica de construcción y, lo más importante, optimizamos el rendimiento reduciendo el Garbage Collection mediante el uso de Object Pooling, reutilizando objetos existentes en lugar de crear otros nuevos constantemente cada vez que queremos spawnear un enemigo.

#### 3.3. Patrón 3: Strategy (Comportamiento)

Implementamos el patrón Strategy en la clase `TowerTargeting` para encapsular los diferentes algoritmos de disparo, tales como First, Last, Close, Strong y Weak.

Permitiendo al jugador adaptar el comportamiento de cada torre individualmente. Además, nos facilita extender el sistema, ya que añadir una nueva estrategia en el futuro nos requiere agregar una nueva clase concreta sin modificar el código base.

#### 3.4. Patrón 4: Observer (Comportamiento)

Para la gestión de eventos, desarrollamos una clase estática `GameEvents` basada en eventos nativos de C#, la cual dispara notificaciones como `OnEnemyDied` o `OnMoneyChanged`. Permiéndonos un desacoplamiento total entre sistemas; por ejemplo, la clase `Enemy` puede notificar su muerte sin tener conocimiento del `SoundManager` o el `EconomyManager`. Esto mejora el mantenimiento y nos facilita la integración de nuevas funcionalidades.

### 3.5. Patrón 5: State (Comportamiento)

Gestionamos el flujo principal del juego implementando el patrón State en el `GameManager`, definiendo estados explícitos como *Preparing*, *WaveInProgress*, *Victory* y *Defeat*. Al estructurarlo de esta manera, conseguimos una mejor organización del código eliminando condicionales complejos y definimos un contexto seguro donde solo las acciones legales para el estado actual pueden ser ejecutadas.

### 3.6. Patrón 6: Command (Comportamiento)

Lo implementamos en el sistema económico encapsulando cada operación en objetos `MoneyTransaction` dentro del `EconomyManager`. Al tratar cada ingreso o gasto como un objeto independiente con datos de fuente y timestamp, obtenemos una trazabilidad completa que resulta esencial para el balanceo y la depuración, permitiéndonos además centralizar la validación de todas las transacciones en un único punto de control.

### 3.7. Patrón 7: Facade (Estructural)

Finalmente, utilizamos el patrón Facade en la clase `TowerUpgradeUI`. Esta clase actúa como una interfaz unificada que coordina la complejidad del `TowerManager`, `EconomyManager` y `TowerConfigManager`. Esto nos permite una gran simplificación en la capa de interfaz de usuario, ya que la UI solo necesita invocar la acción "Mejorar", dejando que la fachada se encargue de coordinar internamente el cobro, la actualización de estadísticas y las animaciones visuales.