PROYECTO FINAL MOMENTO II: DISEÑO Y ANÁLISIS

INTEGRANTES:

DANIELA CARVAJALINO

FABIANA DIAZ ANAYA

DOCENTE:

ANIBAL GUERRA

AUGUSTO SALAZAR

UNIVERSIDAD DE ANTIOQUIA

DEPARTAMENTO DE INGENIERIA ELECTRONICA

INFORMATICA 2

13 DE JUNIO DE 2025

# 1) DIAGRAMA DE CLASES DE LA CAPA LÓGICA, CON NIVEL DE DETALLE MEDIO O SUPERIOR, ESPECIFICANDO RELACIONES Y CONTENEDORES A NECESITAR.

Diagrama de clases de la capa lógica del juego de Goku vs Bandages (la Momia) y Akkuman. Se muestra la estructura de clases excluyendo la interfaz gráfica, con las relaciones de herencia (flechas con triángulo) entre Personaje/Jugador y Enemigo/Enemigos específicos, así como relaciones de composición (rombos llenos) donde una clase contiene a otras (por ejemplo, Nivel contiene enemigos, proyectiles y obstáculos).

## DESCRIPCIÓN GENERAL

El juego está diseñado con una arquitectura orientada a objetos. En el diagrama se distinguen las clases principales que intervienen en la lógica:

* **Juego** (gestor general del juego y los niveles).
* **Nivel** (representa cada escenario o nivel de combate, con su física y contenido).
* **Personaje** (clase base abstracta para todos los personajes, con Goku como Jugador derivado de esta)
* **Enemigo** (clase base para enemigos, con Bandages y Akkuman como subclases específicas).
* **Obstáculo** (clase base para elementos del entorno, como **Plataforma** y **Pared** en los niveles) y **Proyectil** (para ataques especiales como el rayo de Akkuman).

Cada nivel del juego corresponde a un combate contra un enemigo específico en un entorno particular, de acuerdo a la sinopsis dada (Nivel 1: Goku vs *Bandages* en la plataforma sobre el *Baño del Diablo*; Nivel 2: Goku vs *Akkuman* en el centro de energía demoníaca).

### CLASE PERSONAJE (BASE) Y JUGADOR (GOKU)

**Personaje**: Es la clase base que encapsula las propiedades y comportamientos comunes de cualquier personaje (jugador o enemigo). Por ejemplo, contiene atributos como la vida (vida: int), la posición (posX, posY para coordenadas) y la velocidad (velX, velY en cada eje), además de una masa (masa: float) para cálculos físicos. Entre sus métodos principales se incluyen:

* mover(): actualiza la posición del personaje en cada ciclo de juego en función de su velocidad actual (aplicando la física del movimiento). Por ejemplo, la posición se puede actualizar usando integración simple de la velocidad, y también se podría aplicar la gravedad afectando a velY en cada frame.
* saltar(vel0: float, angulo: float): aplica un impulso de salto al personaje, estableciendo su velocidad inicial en un arco parabólico. Esta acción seguiría la fórmula de movimiento parabólico, por ejemplo:

donde vel0 y angulo determinan la velocidad inicial y dirección del salto. Al invocar saltar(...), se asignarían velX y velY iniciales de Goku, mientras la gravedad g (definida en el **Nivel**) se encargará de curvar la trayectoria. Esto permite que Goku esquive ataques o se desplace de un extremo a otro de la plataforma en el Nivel 1.

* recibirDanio(cantidad: int, fuerza: float): reduce la vida del personaje en la cantidad indicada y aplica un retroceso físico si corresponde. Usando la masa (masa) del personaje, se calcula la aceleración resultante (donde fuerza representa la magnitud del golpe) y se ajusta la velocidad en dirección opuesta al impacto, simulando el efecto de ser empujado. Por ejemplo, si Goku recibe un golpe fuerte de Akkuman, este método le restará vida y le impartirá una velocidad (negativa o positiva según la dirección) proporcional a la fuerza del ataque (más fuerza implica mayor cambio de velocidad para una masa dada).
* atacar(objetivo: Personaje): método genérico para efectuar un ataque contra otro personaje (sea jugador o enemigo). En la clase base puede estar definido abstracto o genérico, y cada clase derivada (Jugador/Enemigo) implementará su versión según sus habilidades.

**Jugador**: Es una subclase de Personaje que representa al personaje controlado por el jugador (Goku). Hereda los atributos básicos de Personaje y podría agregar funcionalidades relacionadas al control por el usuario. En este diseño, **Goku** no requiere atributos adicionales específicos más allá de los de Personaje, pero se incluyen métodos particulares:

* atacar(objetivo: Enemigo\*): implementación del ataque de Goku hacia un enemigo. Por ejemplo, Goku podría realizar un golpe cuerpo a cuerpo reduciendo la vida del enemigo si está a distancia corta.
* moverDireccion(dir: int): permite mover a Goku lateralmente según la entrada del jugador (por ejemplo, las teclas de dirección para izquierda/derecha). Este método tomaría una dirección (−1 izquierda, +1 derecha) y ajustaría la posición o velocidad de Goku en consecuencia. Si se desea, se puede aplicar una aceleración constante para que el movimiento sea suave.

### Clase Enemigo (Base) y Enemigos Específicos.

**Enemigo**: Esta es la clase base para los enemigos controlados. Hereda de Personaje todos los atributos comunes (vida, posición, velocidad, etc.) y añade propiedades y métodos.

**Atributos adicionales podrían ser:**

* danioAtaque: int: el daño base que inflige el enemigo con sus ataques (para restar vida al jugador cuando acierta).
* objetivo: Personaje\*: una referencia al personaje objetivo, típicamente el Jugador (Goku). Esta referencia le permite al enemigo saber dónde está o interactuar con el jugador (por ejemplo, para perseguirlo o atacarlo). En el diagrama se indica con una asociación *dashed* llamada "objetivo" desde Enemigo hacia Jugador.

**Métodos principales en Enemigo incluyen:**

* patrullar(): movimiento automático del enemigo según su patrón de comportamiento en el nivel. Por ejemplo, Bandages podría avanzar y retroceder en la plataforma buscando acercarse a Goku, o Akkuman podría moverse rápidamente de lado a lado (incluso con aceleración) en el área de combate.
* atacar(objetivo: Personaje\*): un método polimórfico para que el enemigo ataque a su objetivo. Cada subclase enemiga lo implementará de forma distinta (por ejemplo, cuerpo a cuerpo vs. ataque a distancia). Puede incluir chequeos de distancia o condiciones (solo atacar si Goku está al alcance, etc.).

#### ENEMIGOS ESPECÍFICOS.

**Bandages (Momia)**: Subclase de Enemigo que representa al enemigo del Nivel 1, la momia **Bandages**. Bandages ataca principalmente **cuerpo a cuerpo** y tiene la habilidad de **inmovilizar con sus vendajes**. En la clase Bandages podemos destacar:

* Un atributo como alcanceVendaje: float para definir la distancia o área en la que puede atrapar a Goku con sus vendas.
* atacar(objetivo: Personaje\*): la implementación específica podría verificar si Goku está a una distancia corta. Si es así, Bandages podría usar un golpe o intentar atraparlo.
* inmovilizar(objetivo: Jugador): método especial para la momia, que aplicaría el efecto de envolver a Goku con vendas. Esto podría establecer algún estado en el jugador (p.ej., un **flag** de estaInmovilizado por cierto tiempo) o reducir su velocidad a 0 temporalmente, simulando que Goku no puede moverse. En el juego, el jugador tendría que liberarse tras unos segundos. Si Bandages logra inmovilizar a Goku y empujarlo cerca del borde, aumenta la probabilidad de que Goku caiga al Baño del Diablo.

*Mecánicas de Bandages:* Su **patrón de movimiento** sería acercarse lentamente al jugador en la plataforma. Cuando está muy próximo, usa inmovilizar para sujetarlo. El jugador debe saltar o esquivar a tiempo para evitar esta técnica. Bandages no posee ataques a distancia ni proyectiles, lo que se refleja en que no tiene una relación con la clase Proyectil. Sus colisiones con Goku son cuerpo a cuerpo; y si Goku logra golpearlo suficientemente (reduciendo su vida a 0), el nivel se supera. En términos de física, si Bandages recibe un golpe fuerte de Goku, la misma lógica de recibirDanio (heredada de Personaje) le aplicará un retroceso, potencialmente empujándolo hacia atrás en la plataforma.

**Akkuman**: Subclase de Enemigo que representa al enemigo del Nivel 2, **Akkuman** (el diablo del palacio). Akkuman es más complejo porque posee un ataque mágico de proyectil y movimientos más rápidos. Elementos destacados de su clase:

* Atributo tiempoRecarga: float (o **cooldown**): para controlar cada cuánto tiempo Akkuman puede lanzar su ataque especial *“Resplandor Mortal del Diablo”*. Esto evita que dispare constantemente; el valor se decrementaría en el tiempo y Akkuman solo dispara cuando este llega a cero, reiniciándolo luego del disparo.
* atacar(objetivo: Personaje\*): implementación que generalmente consistirá en **disparar un proyectil** en dirección a Goku. Si la línea de visión está libre (no está demasiado cerca quizá), Akkuman usará lanzarRayo(). Si Goku se acerca demasiado, Akkuman podría alternar con un ataque cuerpo a cuerpo débil o intentar moverse a otra posición
* lanzarRayo(): este método crea un nuevo objeto de la clase Proyectil representando el **rayo diabólico**. El proyectil nace en la posición de Akkuman y viaja en línea recta hacia Goku. La velocidad inicial del proyectil se determina de forma que siga una trayectoria rectilínea rápida. Este rayo es el que tiene la propiedad de **reflejarse** en las paredes del escenario.
* moverRapido(): Akkuman puede realizar movimientos laterales acelerados. Este método puede implementar la fórmula de movimiento con aceleración: por ejemplo usar

de manera que desde reposo Akkuman pueda **acelerar** rápidamente hacia la izquierda o derecha (por ejemplo, para esquivar un ataque o reposicionarse). En la práctica, moverRapido podría incrementar la velX de Akkuman aplicando una aceleración constante durante unos frames, lo que se traduce en un desplazamiento más brusco comparado al movimiento normal de caminar de Goku. Este comportamiento hace a Akkuman un enemigo más difícil de predecir, acorde a la descripción.

**Clase Proyectil (Ataque Especial)**

La clase **Proyectil** modela los proyectiles generados por ataques especiales (en este juego, principalmente los rayos de Akkuman). Contiene atributos para su posición (posX, posY), velocidad (velX, velY) y daño que ocasionan (dano: int). Sus métodos principales permiten manejar su movimiento y colisiones:

* mover(): actualiza la posición del proyectil con su velocidad actual. En cada frame del nivel, todos los proyectiles activos se mueven en línea recta. (No se les aplica gravedad si son rayos de energía, ya que viajan rectos).
* reflejar(): invierte la dirección del proyectil cuando choca con una superficie reflectante. Por ejemplo, si golpea una **Pared** lateral, este método podría simplemente hacer velX = -velX (cambiando el sentido horizontal, conservando la magnitud), logrando que el rayo rebote hacia el lado opuesto. Esto implementa la colisión **elástica** para el proyectil, donde el ángulo de incidencia equivale al ángulo de rebote. (Si hubiera superficies inclinadas, sería más complejo, pero en este nivel asumimos rebote horizontal).
* colisionar(objetivo: Personaje\*): verifica si el proyectil impacta a un personaje (jugador o enemigo). Podría chequear la cercanía o intersección de coordenadas. Si hay impacto, se reducirá la vida del objetivo en danio y el proyectil podría destruirse (removerse de la lista de activos). En nuestro caso, solo Goku puede ser objetivo de un proyectil (ya que solo Akkuman los genera).

Todos los proyectiles existentes en el juego son gestionados por el Nivel: este mantiene una **lista de proyectiles activos** y, en cada actualización de frame, llama mover() para cada proyectil y luego verifica colisiones: contra paredes (para reflejar) o contra el jugador (para aplicar daño). De esta manera, la lógica de los **rebotes** y daños por proyectil están encapsuladas en esta clase y en la detección de colisiones del nivel.

**Clase Obstáculo y Entorno del Nivel**

Para representar las características del **escenario** o entorno físico de cada nivel, se define la clase base **Obstáculo**, de la cual derivan tipos concretos de obstáculos del terreno: por ejemplo, **Plataforma** (suelo o plataforma donde se pelea) y **Pared** (límites verticales del área). La clase Obstáculo tiene propiedades genéricas: posición y dimensiones (posX, posY, ancho, alto podrían definir su área en coordenadas del nivel), y flags booleanos para indicar su naturaleza física:

* reflectivo: bool (si true, el obstáculo refleja proyectiles en colisiones elásticas).
* letal: bool (si true, el obstáculo causa muerte al jugador si este lo toca o lo cruza).

En la implementación, los **Obstáculos** de un nivel se guardan en una colección (por ejemplo, una lista) dentro de la clase Nivel. La relación de composición en el diagrama indica que un Nivel “contiene” múltiples obstáculos (0, 1 o varios según el escenario).

**Plataforma**: Representa superficies horizontales donde el personaje puede pararse/saltar. En el Nivel 1, la plataforma principal sobre el Baño del Diablo sería una instancia de Plataforma. Atributos y comportamientos:

* Puede tener un atributo específico como longitud: float para el largo de la plataforma (en X), además de usar posX, posY heredados (por ejemplo, posY sería la altura a la que está la plataforma sobre el fondo mortal).
* Método esFueraDePlataforma(x: float): bool: determina si una coordenada X dada está fuera de los límites de la superficie de la plataforma. Esto sirve para detectar si Goku se ha desplazado más allá del borde. En el Nivel 1, si Jugador.posX sale del rango [plataforma.x\_inicial, plataforma.x\_final], significa que Goku ha pisado el vacío. Entonces, al siguiente instante su posY comenzará a caer por gravedad y, si no hay otra plataforma debajo (que no la hay en este nivel), se considera **caída mortal**. El Nivel comprobará en cada frame: si Jugador.posY cae por debajo de posY de la plataforma entonces letal se activa y declara la derrota inmediata. La plataforma en sí podría tener letal: false (estar sobre ella no mata, claro), pero el área fuera de ella es letal.

**Pared**: Representa muros verticales que delimitan el área. En el Nivel 2, por ejemplo, hay dos paredes laterales que encierran la arena de Akkuman. Características:

* Las paredes se definen por posición (posX,posY de su esquina, y ancho/alto, aunque ancho podría ser muy pequeño si es muro delgado, el alto cubre desde el suelo hasta cierta altura).
* Tienen reflectivo: true en el Nivel 2, para que los proyectiles de Akkuman reboten. Esto significa que cuando la detección de colisiones del Nivel encuentra que un Proyectil toca una Pared, llamará a Proyectil.reflejar() en lugar de destruirlo. Por otro lado, para personajes (Goku/Akkuman), las paredes actúan como límites sólidos: un personaje no puede atravesarlas. Podemos manejar esto revirtiendo o anulando velX de un personaje al intentar cruzar la pared (efecto de tope).

NOTA: Para un **Nivel 3 opcional (Cámara antigravedad)** se podrían reutilizar estas clases de Obstáculo. Por ejemplo, Plataforma podría tener una subclase Plataforma Flotante (no incluida en el diagrama) que agrega un movimiento oscilatorio. Dichas plataformas flotantes se agregarían a la lista de obstáculos del nivel 3, y la gravedad reducida se manejaría simplemente estableciendo Nivel.gravedad = 0.3 \* g en ese nivel. Esto demuestra que la estructura diseñada es flexible para extender a nuevos niveles con distintas físicas, sin cambios fundamentales en las clases base.

**Clase Nivel (Gestor de la Lógica por Nivel)**

La clase **Nivel** encapsula todo lo que ocurre en un escenario particular de combate. Cada instancia de Nivel tiene:

* Atributos para configurar la física general del nivel, por ejemplo: numero: int (identificador o índice del nivel), gravedad: float (valor de la aceleración gravitacional aplicable, p.ej. 9.8 m/s² en niveles normales, o reducido en cámara antigravedad), coefRestitucion: float (el coeficiente *e* de restitución usado para colisiones inelásticas en ese nivel; por ejemplo, 0.8 en nivel 1 para paredes rocosas, tal vez 1.0 en nivel 2 para muros mágicos perfectamente elásticos con proyectiles). Estos valores permiten adaptar las fórmulas físicas a cada nivel fácilmente.
* Relaciones (composición/agregación) con los objetos que contiene el nivel: un **Jugador** (Goku) y **Enemigo** principal, además de listas de **Proyectiles** y **Obstáculos** presentes. En el diagrama, las multiplicidades indican que cada Nivel tiene **1 Jugador**, **1 Enemigo**, *0..*∞ **Proyectiles** (podría no haber proyectiles si el enemigo no lanza, o muchos a la vez si lanza varios) y *0..*∞ **Obstáculos** (según el escenario; el Nivel 1 tiene 1 plataforma y tal vez límite inferior, el Nivel 2 tiene 3 obstáculos – suelo y dos paredes, etc.).

**Los métodos clave de la clase Nivel incluyen:**

* iniciar(): prepara el nivel, inicializando la posición del Jugador y del Enemigo, configurando el escenario (por ejemplo, ubicando las plataformas/paredes en coordenadas, estableciendo la gravedad y coeficientes adecuados). Este método podría llamarse al cargar un nivel nuevo. Si hay elementos únicos del nivel podría controlarse aquí también.
* actualizar(): este es el **bucle de juego** por frame dentro del nivel. Se encarga de actualizar la lógica en cada pequeño intervalo de tiempo. Típicamente hará lo siguiente en orden:
  1. Mover al jugador y enemigo: llamar Jugador.mover() (según la última entrada del jugador y física) y también Enemigo.patrullar() para calcular su nueva posición o acción. Ambos movimientos estarán sujetos a la física (gravedad afectando velY de ambos, fricción si se considerara, etc.).
  2. Actualizar proyectiles: para cada Proyectil en la lista, llamar mover() para avanzar su posición.
  3. Llamar a detectarColisiones(): este método maneja todas las interacciones y choques en el nivel:
* **Jugador vs Obstáculos**: evitar que Goku atraviese el piso o paredes. Si Goku está por debajo de la plataforma (caída), o fuera de límites, este método detectará esa condición (usando esFueraDePlataforma o comparando posiciones) y marcará el nivel como perdido para el jugador. Si Goku choca contra una pared en Nivel 2, se ajustará su posición para que no salga del campo y opcionalmente reducirá su velocidad velX multiplicándola por coefRestitucion (simulando impacto).
* **Enemigo vs Obstáculos**: similar al jugador, evita que el enemigo salga del área. (Por ejemplo, Akkuman rebotaría o se detendría al llegar a una pared).
* **Jugador vs Enemigo**: si están lo suficientemente cerca para golpearse, podría resolver colisiones cuerpo a cuerpo o simplemente dejar que los ataques manejen el daño. Si se implementa detección de choque físico, podría hacer que reboten ligeramente uno contra otro (pero en este juego probablemente el daño se aplica mediante ataques explícitos más que choques físicos).
* **Proyectil vs Pared/Plataforma**: cuando un proyectil alcanza un obstáculo: si el obstáculo es reflectivo (pared mágica), llamar proyectil.reflejar() para que cambie de dirección en vez de destruirlo. Si fuese un obstáculo no reflectivo (por ejemplo, si un proyectil choca con el suelo o con una persona, aunque personas se manejan aparte), podría eliminarse.
* **Proyectil vs Jugador/Enemigo**: verificar si algún proyectil golpea al Jugador (lo más probable) u ocasionalmente al Enemigo (por ejemplo, si un rayo rebotado podría dañar al propio Akkuman, dependiendo de la lógica; esto no está especificado, pero podría considerarse). Si hay impacto, usar Personaje.recibirDanio(dano, fuerza) en el objetivo correspondiente para restar vida y aplicar empuje. Luego quitar el proyectil de la lista (se destruye tras impactar).
  1. Verificar condiciones de victoria/derrota: el método esTerminado(): bool del nivel puede chequear si el **Enemigo ha sido vencido** (Enemigo.vida <= 0, victoria) o si el **Jugador ha muerto o caído** (Jugador.vida <= 0 o se activó condición de caída mortal, derrota). Según el resultado, el nivel termina y se notificará a la clase Juego para proceder (pasar al siguiente nivel o terminar el juego).
* esTerminado(): bool: indica si el nivel ha concluido ya sea por victoria o derrota, posiblemente diferenciando el estado. Podría ser complementado con métodos como nivelGanado() o nivelPerdido() según convenga. En una implementación, cuando Goku derrota al enemigo, podría desencadenarse una transición (mostrar algún mensaje o cinemática) antes de cargar el siguiente nivel.

En síntesis, la clase Nivel orquesta la interacción de personaje, enemigo, proyectiles y entorno para recrear las situaciones de cada episodio: en el nivel de la momia, controlar que Goku no caiga al ácido y que la momia intente atraparlo; en el nivel de Akkuman, manejar los rebotes del rayo y el espacio cerrado con paredes. Los parámetros físicos (gravedad, coefRestitucion) se configuran aquí para ajustar el comportamiento: por ejemplo, un coefRestitucion < 1 para que los choques no sean perfectamente elásticos (pérdida de velocidad en cada rebote contra superficies normales), mientras que los proyectiles de Akkuman ignoran este coeficiente al rebotar en las paredes mágicas (son **totalmente elásticos** en esa mecánica específica). Si se activara un nivel con gravedad alterada, simplemente se modificaría Nivel.gravedad a un valor menor (como 0.3\*g) y quizás coefRestitucion a otro valor si se quisiera que los rebotes sean diferentes en ese entorno.

**Clase Juego (Control General del Juego)**

La clase **Juego** es un controlador de alto nivel que maneja la secuencia de niveles y el estado global del juego. Dado que el diagrama se enfoca en lógica y no en interfaz, Juego actuaría tras bambalinas para inicializar y cambiar niveles, además de mantener al jugador persistente si es necesario. Sus responsabilidades incluyen:

* **Gestionar el Jugador**: Juego contiene probablemente una instancia única de Jugador (Goku) que persiste a través de todos los niveles. En el diagrama, Juego tiene una relación de composición con Jugador (rombo en Juego apuntando a Jugador), indicando que crea y posee al objeto Goku. Esto permite conservar atributos como la vida acumulada o puntaje entre niveles (si se deseara). Alternativamente, podría recrearse el Jugador por nivel, pero generalmente es único.
* **Listado de Niveles**: Juego mantiene los niveles que componen la aventura. Podría ser una lista (niveles: list<Nivel\*>) preconstruida con Level1, Level2, etc., o Juego va creando el Nivel necesario al momento. En el diagrama se muestra un contenedor 1..\* de niveles en Juego, implicando que el juego tiene uno o varios niveles a manejar. Por ejemplo, inicialmente se carga Nivel 1 (Bandages), al ganarlo Juego avanzará a Nivel 2 (Akkuman), y así sucesivamente.
* **Transición de niveles**: métodos como iniciarJuego() y cargarNivel(n: int) controlan el flujo. iniciarJuego() puede establecer el estado inicial (nivelActual = 1, vida de Goku llena, etc.). cargarNivel(n) instanciaría o activaría el nivel número *n*: por ejemplo, crea un objeto Nivel correspondiente (inicializando sus obstáculos y enemigo apropiado según *n*). También colocaría al Jugador en la posición de inicio del nivel.
* **Bucle general de juego**: aunque cada Nivel tiene su lógica de actualización, Juego podría tener un método actualizarJuego() que se llama periódicamente (por ejemplo, ligado al timer del framework) y que internamente delega a nivel.actualizar() del nivel actual. Tras actualizar, Juego comprueba nivel.esTerminado(): si true, decide si fue victoria o derrota. En caso de victoria, puede incrementar nivelActual y cargar el siguiente nivel (o finalizar el juego si no hay más). En caso de derrota, podría reiniciar el nivel o finalizar la partida (dependiendo de las reglas, quizás dar opción de reintentar).
* **Manejar colisiones/inputs a nivel global**: En este diseño, las colisiones se manejaron dentro de Nivel. Juego.manejarColision() aparece en el diagrama más como referencia de que puede haber lógica de colisiones global (pero realmente la mayoría está en Nivel). Podríamos usar manejarColision() para, por ejemplo, reaccionar a ciertos eventos globales, pero es posible que no sea necesario si todo se resuelve en cada nivel. En cuanto a **inputs**, estos serían recibidos externamente (desde la ventana de juego) y pasados a Jugador.moverDireccion() o Jugador.saltar() etc. Juego podría orquestar eso llamando los métodos del Jugador según la entrada, pero eso depende de cómo se integre con la interfaz. Dado que no incluimos la interfaz gráfica, podemos asumir que en algún lugar los eventos de teclado invocan acciones del Jugador directamente.

En resumen, Juego se asegura de que Goku pase por los niveles 1 y 2 (y eventualmente 3 si existe) en orden, manteniendo la continuidad. Actúa como el **contenedor principal** de la lógica del juego, mientras que nivel encapsula la lógica de cada enfrentamiento individual. Cada clase está pensada para corresponder a elementos del diseño del juego, facilitando implementar las mecánicas descritas (saltos parabólicos, colisiones elásticas/inelásticas, ataques especiales, etc.) de forma estructurada y modular.

# 2. CAMBIOS REALIZADOS Y DETALLES AMPLIADOS RESPECTO AL MOMENTO I

Durante la segunda fase del proyecto, se consolidaron y extendieron significativamente varios aspectos del videojuego "Goku en el Palacio de Uranai Baba". A continuación, se documentan los cambios clave implementados desde la entrega del Momento I:

1. **Mayor nivel de abstracción y estructuración orientada a objetos**  
   Se realizó una formalización completa de la capa lógica del juego mediante el uso de un diagrama UML. Este incluye clases base (Personaje, Obstaculo, Nivel, Juego) y sus derivadas (Jugador, Enemigo, Bandages, Akkuman, Plataforma, Pared), permitiendo una implementación reutilizable y escalable. El modelo facilita agregar nuevos enemigos o escenarios sin alterar la estructura existente.
2. **Detalle explícito de interacciones y comportamientos**  
   A diferencia del Momento I, donde se describieron interacciones generales, ahora cada enemigo cuenta con métodos propios que reflejan comportamientos específicos: inmovilizar() en Bandages y lanzarRayo() + moverRapido() en Akkuman. Así mismo, se definió el patrón de ataque, condición de activación y alcance efectivo.
3. **Incorporación de proyectiles y colisiones reflejadas**  
   Se creó una clase Proyectil con atributos de posición, velocidad y daño, y métodos para moverse, reflejarse y colisionar. Esto permite modelar el "Resplandor Mortal del Diablo" de Akkuman y su comportamiento físico al rebotar en paredes reflectantes del Nivel 2.
4. **Parametrización de la física por nivel**  
   La clase Nivel ahora incluye atributos como gravedad y coefRestitucion para configurar el entorno físico. Esto permite que cada nivel tenga gravedad distinta (por ejemplo, reducida en el Nivel 3), así como comportamiento elástico o inelástico en colisiones según se requiera.
5. **Preparación para niveles extendidos (Nivel 3)**  
   El sistema está diseñado para soportar niveles adicionales como el modo de desafío antigravedad, incorporando plataformas oscilantes, trayectorias curvas y ataques dirigidos. No fue necesario modificar la jerarquía central, lo que demuestra la extensibilidad del diseño.

## 3. DESCRIPCIÓN DETALLADA DE VISTAS, INTERACCIONES Y FÍSICAS POR NIVEL

Cada nivel se diseñó con un enfoque específico en la mecánica de juego, variando el entorno, los patrones enemigos y el sistema de físicas. A continuación, se presentan las descripciones completas:

**NIVEL 1: GOKU VS BANDAGES LA MOMIA**

**Vista del escenario:** Plataforma horizontal sin paredes laterales, suspendida sobre el "Baño del Diablo", un fondo letal que provoca la derrota inmediata al caer. Estilo 2D con vista lateral.

**Interacciones:**

* Bandages patrulla lateralmente.
* Si Goku entra en su alcanceVendaje, activa inmovilizar() por 2 segundos.
* Durante ese estado, Bandages puede empujarlo con un golpe. Si Goku cae, pierde la partida.
* Goku puede saltar, moverse y atacar con golpes cuerpo a cuerpo.

**Físicas implementadas:**

* Movimiento parabólico al saltar (saltar(vel0, ángulo) con gravedad).
* Colisión inelástica (empuje sin rebote).
* Caída mortal detectada con esFueraDePlataforma() y posY por debajo de la plataforma.

**Objetivo:** Evitar la inmovilización, no caer del escenario y derrotar a Bandages con ataques cuerpo a cuerpo.

**NIVEL 2: GOKU VS AKKUMAN**

**Vista del escenario:** Arena rectangular cerrada por dos paredes reflectantes laterales. Fondo energético. Vista lateral 2D.

**Interacciones:**

* Akkuman lanza un rayo (Proyectil) en línea recta cada cierto tiempoRecarga.
* El rayo rebota horizontalmente si toca una Pared con reflectivo = true.
* Goku debe esquivarlo; si es golpeado, pierde salud y retrocede.
* Si Goku logra posicionarse correctamente, puede hacer que el rayo reflejado impacte a Akkuman.

**Físicas implementadas:**

* Colisión elástica (rebote con mismo ángulo de incidencia).
* Movimiento lateral con aceleración (moverRapido() usando aceleración constante).
* Retroceso por impacto calculado por masa y fuerza del golpe.

**Objetivo:** Esquivar ataques, encontrar ventanas seguras para contraatacar, y utilizar el entorno reflectante a su favor.

**NIVEL 3 (OPCIONAL): GOKU VS GOKU OSCURO (CAMARA ANTIGRAVEDAD)**

**Vista del escenario:** Entorno flotante con fondo espacial. Plataformas que suben y bajan con movimiento sinusoidal. Algunas desaparecen temporalmente. Vista lateral 2D.

**Interacciones:**

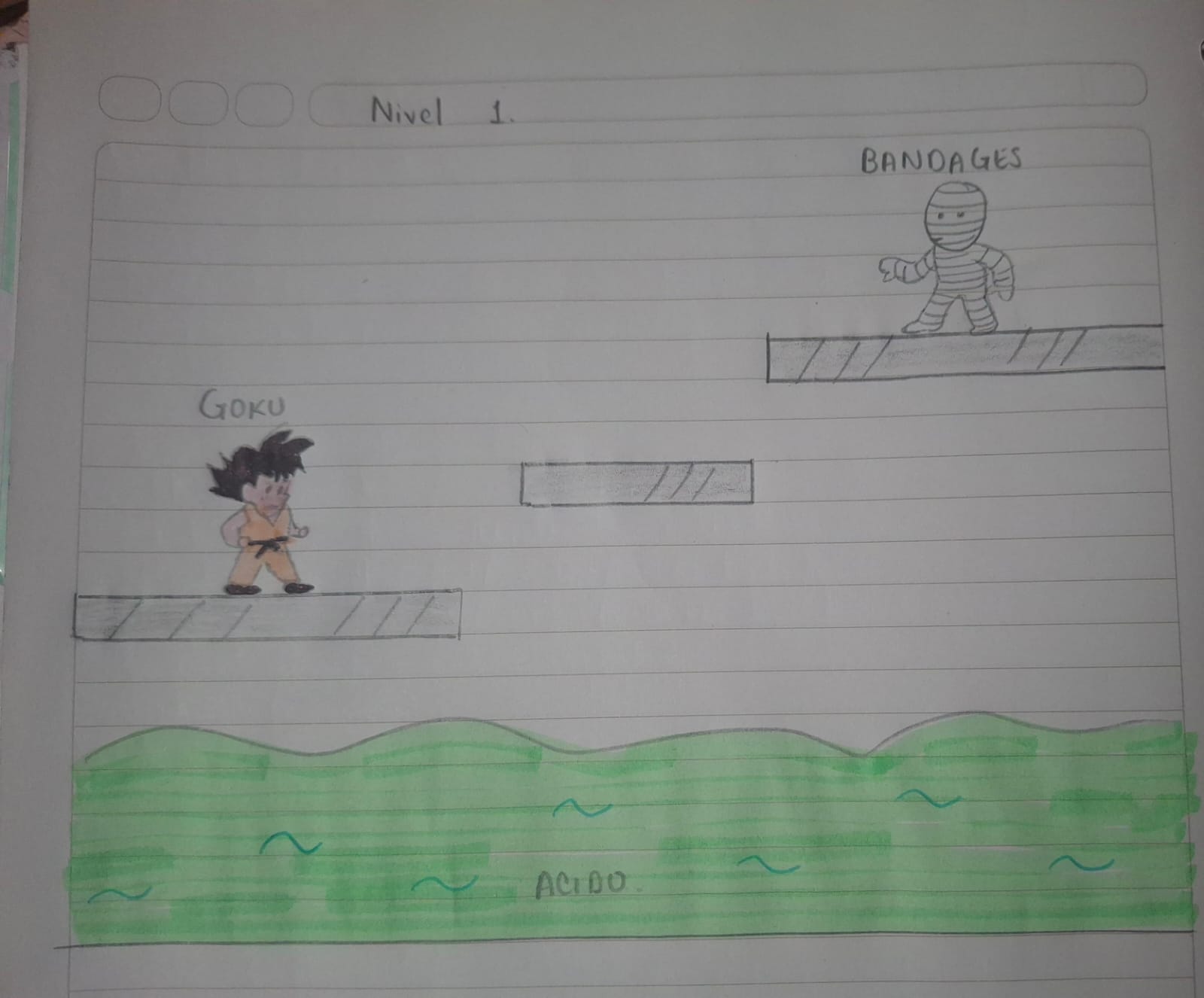
* Goku Oscuro imita y contraataca los movimientos del jugador.
* Ataca con proyectiles de trayectoria curva que corrigen su rumbo.
* Teletransportación si el jugador permanece estático.
* Si el jugador flota sin tocar plataformas por mucho tiempo, entra en estado de "descontrol" (pérdida de precisión).

**Físicas implementadas:**

* Gravedad reducida (gravedad = 2.5 m/s^2).
* Movimiento de plataformas: y(t) = A \* sin( ωt ).
* Trayectorias curvas en proyectiles dirigidos.
* Penalización por flotar: control reducido si el personaje no aterriza periódicamente.

**Objetivo:** Dominar el entorno inestable, anticipar patrones del enemigo espejo y atacar desde posiciones seguras.

**DIBUJOS DE LOS NIVELES**

**NIVEL 1.**

**NIVEL 2.**



## 4. SPRITERS



Ilustración 2 Sprite de Goku a la izquierda.

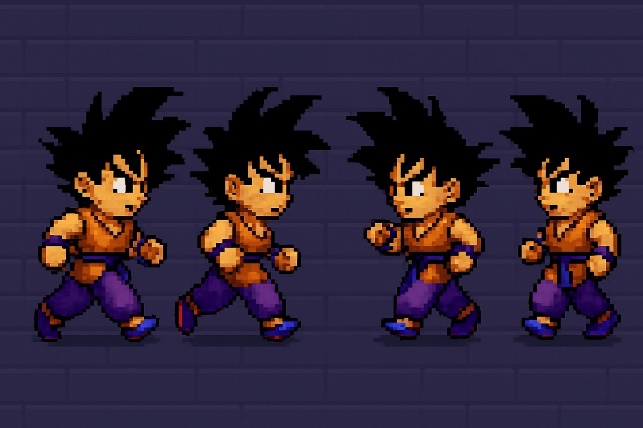
Ilustración 1 Sprite de Goku al frente.

Ilustración 4 Sprite de Goku a la derecha.

Ilustración 3 Sprite de Goku moviéndose.