Tarea programada grupal 1

Programación I

Benjamin Hernández Mairena C5F913

Julián Barrantes Madrigal C5D034

Gabriel Perez Schlager C4I384

Prof. Rodrigo Li

Decisiones de diseño

El apartado de decisiones de diseño se dividirá en clases, mismas que se dividirán respectivamente en constructores, instancias y métodos

MAIN

1. Patrones y Estructura Principal
Inicialización Externa: Se utiliza la clase Inicializar para crear todos los datos base del juego (elementos, ataques, Pokémon, etc.). Esto adhiere al Principio de Responsabilidad Única, manteniendo a Main enfocado en la orquestación del juego.

Flujo de Control Lineal: La lógica principal reside en un bucle while en main, que define una secuencia lineal de combates contra NPC predefinidos (indiceNPC < 4).

2. Gestión del Estado de Objetos (Clonación)
Evitar Mutación Global: La decisión más crítica es el uso del método estático crearNuevoPokemon antes de cada batalla.

Crea copias frescas de los objetos Pokemon y sus Stat (específicamente, restableciendo el HPActual y el PP) a partir de los prototipos originales.

Esto asegura que los cambios de estado por el combate (daño, consumo de PP) no persistan, permitiendo que el jugador y el NPC comiencen la siguiente batalla con sus Pokémon completamente restaurados.

3. Acoplamiento e Interfaz de Usuario
Alto Acoplamiento a Utilerías: La clase Main está enlazada a las utilidades de consola, utilizando Scanner directamente para pausas (s.nextLine()) y llamando a utilidades de presentación como b.limpiarPantalla() de la clase Batalla.

INICIALIZAR

Decisiones de Diseño en la Clase Inicializar
1. Patrones y Estructura Principal
Patrón de Clase de Servicio Estática (Utility/Data Factory): La clase está diseñada casi en su totalidad con miembros y métodos static. Esta decisión elimina la necesidad de crear una instancia de Inicializar en la clase Main (aunque Main la instancia, no es necesario) y la establece como un contenedor global de datos y funciones de inicialización.

Encapsulamiento de Datos Maestros: La clase es responsable de crear y almacenar todos los datos base del juego (elementos, ataques, Pokémon, estadísticas y entrenadores) en pools (arreglos estáticos). Esto centraliza la definición de los objetos del mundo del juego.

Separación de Responsabilidades: Cumple con el Principio de Responsabilidad Única al aislar la compleja lógica de configuración inicial del juego del flujo de control principal (que reside en Main).

2. Gestión del Estado de Objetos (Datos Maestros)
Uso de Variables static para Pools: Todos los pools de datos (poolElementos, poolAtaques, poolPokemonesNPC, etc.) son declarados static. Esta decisión asegura que existe solo una copia de estos datos en la memoria para toda la aplicación, actuando como las tablas de datos maestros del juego.

Estrategia de Inicialización de Stat: Se crean instancias separadas de Stat para los Pokémon NPC y los Pokémon del jugador (e.g., statsCharizardNPC y statsCharizardJ), aunque tienen los mismos valores iniciales. Esto es una decisión defensiva para garantizar que los cambios de estado (como HP actual) durante la selección del jugador en inicializarJugador no afecten a las plantillas de los NPC, o viceversa, previniendo efectos secundarios no deseados.

Acoplamiento y Dependencia Estructural: La secuencia de los métodos de inicialización es estrictamente dependiente (por ejemplo, inicializarAtaques necesita que inicializarElementos haya poblado poolElementos). El diseño impone un orden en el que se deben llamar los métodos.

3. Acoplamiento e Interfaz de Usuario
Manejo de I/O en la Inicialización (inicializarJugador): La decisión de incluir el flujo de selección de Pokémon directamente en el método inicializarJugador acopla la creación del objeto Entrenador Jugador con la interfaz de usuario de la consola (Scanner, System.out.println).

Dependencia de Batalla para I/O: El método inicializarJugador llama a b.limpiarPantalla(), lo que crea una dependencia innecesaria entre la clase Inicializar y la clase Batalla para una tarea puramente cosmética de la interfaz de usuario. Idealmente, las utilidades de I/O no deberían residir en una clase de Factory de datos.

Acoplamiento con Arrays.toString: Se utiliza Arrays.toString en el loop de selección de Pokémon para mostrar el contenido de los arreglos de Elemento y Ataque. Esto es una decisión de diseño para la presentación de datos complejos en la consola.

POKEMON

1. La clase Pokemon está diseñada como un modelo de datos central en el paradigma de Programación Orientada a Objetos (POO).

Encapsulamiento Fuerte: Todos los campos de instancia (atributos) se declaran como private (nombre, nivel, stats, elementos, ataques, debilitado). Esta es una decisión clave para proteger el estado interno del objeto y forzar el acceso/modificación a través de métodos públicos (getters y setters).

Composición de Datos (Relación "Tiene un"): Un objeto Pokemon no solo contiene datos primitivos, sino que compone otras estructuras de datos como Stat, Elemento[] y Ataque[]. Esto modela la relación de la vida real del juego (un Pokémon tiene estadísticas, tiene tipos, y tiene ataques).

Sobrescritura de toString(): El método toString() está sobrescrito para devolver solo el nombre. Esta es una decisión simple de diseño para facilitar la representación en cadena del objeto en contextos de impresión o depuración, haciéndola concisa y legible.

2. Constructor
El constructor está diseñado para forzar la inicialización completa de la instancia en un estado válido.

Constructor con Argumentos Obligatorios: El constructor público requiere todos los atributos de la clase como parámetros. Esto asegura que no se pueda crear una instancia de Pokemon que carezca de su nombre, nivel, estadísticas, tipos o ataques, garantizando la integridad de la información.

Asignación Directa (this): El uso de la palabra clave this. en el constructor es la decisión estándar para diferenciar entre las variables de instancia y las variables locales de los parámetros.

3. Métodos
La estructura de métodos se basa en el principio de separar el acceso del comportamiento.

Patrón Getter / Setter: Se proporciona un par de métodos public de acceso (getNombre, setNombre, etc.) para cada campo private.

Getters: Permiten la lectura externa de los valores de los atributos.

Setters: Permiten la modificación externa del estado del objeto.

Control de Estado Específico (isDebilitado): Se utiliza el prefijo is en lugar de get para el getter del campo booleano (debilitado). Esta es una convención de diseño de Java para métodos que consultan el estado booleano.

Falta de Métodos de Comportamiento: Es una decisión de diseño notable que la clase no contenga métodos de comportamiento (como atacar(), recibirDano(), curarse()). Esto

sugiere que la lógica de la batalla (el comportamiento) se manejará externamente, muy probablemente en la clase Batalla, manteniendo a Pokemon centrado únicamente en su estado (modelo de datos).

4. Instancias
Las decisiones sobre las instancias se centran en la gestión de su identidad, estado y la memoria que consumen sus colecciones.

Inicialización de Colecciones: Los campos elementos y ataques se inicializan con arreglos de tamaño fijo (new Elemento[2], new Ataque[4]) en la declaración de la clase, aunque esta inicialización se sobrescribe inmediatamente en el constructor. Esta es una decisión de diseño para limitar la cantidad máxima de tipos (a 2) y ataques (a 4) que un Pokémon puede tener.

Instancias Mutables: La inclusión de todos los setters hace que una instancia de Pokemon sea completamente mutable. El estado del Pokémon (como su nombre, nivel, o si está debilitado) puede cambiarse en cualquier momento después de su creación, lo cual es necesario para simular eventos de juego como subir de nivel o ser derrotado.

Dependencia de Instancias: Cada instancia de Pokemon depende de una instancia externa de Stat y de colecciones de instancias de Elemento y Ataque. Esto significa que la vida útil y la mutabilidad de estos objetos anidados afectarán al estado del Pokemon.

STAT

1. Patrones y Estructura Principal
Modelo de Datos Compuesto: La clase Stat está diseñada como un objeto valor que agrupa todas las estadísticas numéricas clave de un Pokémon (HP, ATK, DEF, SPD) en una sola entidad. Esta decisión simplifica el modelo Pokemon, que solo necesita referenciar un objeto Stat en lugar de cinco campos de estadísticas individuales.

Encapsulamiento Fuerte: Todos los campos de instancia se declaran como private, lo que constituye una decisión clave para proteger los valores numéricos y forzar que la modificación se realice a través de los métodos setter.

2. Gestión del Estado de Objetos (Atributos y Mutabilidad)
Seguimiento Dual de HP: La decisión más crucial es la inclusión de dos atributos para los Puntos de Salud (HP y HPActual).

HP (HP Máximo): Modela el valor estático o base de la salud máxima del Pokémon.

HPActual (HP Actual): Modela el valor mutable que disminuye durante el combate.

Esta distinción es fundamental para la simulación de batalla, permitiendo que el daño se aplique a HPActual sin perder el valor de referencia de la salud total.

Mutabilidad Necesaria: La inclusión de setters para todos los campos hace que una instancia de Stat sea mutable.

La mutabilidad de HPActual y las estadísticas de combate (ATK, DEF, SPD) es necesaria para simular los efectos dinámicos del juego, como recibir daño, curarse, o aplicar aumentos/disminuciones temporales de estadísticas.

Inicialización Completa: El constructor exige todos los valores (HP, HPActual, ATK, DEF, SPD) al momento de la creación. Esto garantiza que una instancia de Stat siempre se inicialice con un conjunto completo de valores válidos.

3. Métodos y Comportamiento
Patrón Getter / Setter: La clase implementa el patrón de acceso más simple, proporcionando un getter y un setter para cada atributo, lo que permite el control total sobre la lectura y escritura de cada estadística individual.

Falta de Lógica de Negocio: La clase se enfoca estrictamente en almacenar datos. Es notable que no contiene métodos que implementen lógica de negocio (ej. aplicarDaño(int daño), aumentarAtaque(int stages)). Esto adhiere al Principio de Responsabilidad Única al mantener la lógica de manipulación de estadísticas (el comportamiento) fuera de la clase de modelo de datos, residiendo esta lógica, por ejemplo, en la clase Batalla.

ATAQUE

1. Patrones y Estructura Principal
Modelo de Datos de Dominio (POO): La clase está diseñada para modelar la entidad Ataque dentro del contexto del juego, encapsulando todas sus propiedades clave (nombre, potencia, PP, etc.).

Encapsulamiento de Estado: Todos los atributos se declaran private, lo que constituye una decisión de diseño para proteger los datos internos y gestionar su acceso únicamente a través de métodos públicos (getters y setters).

Relación de Composición (Tipo/Elemento): El ataque compone una instancia de Elemento. Esto modela la relación real de que un ataque tiene un tipo elemental, delegando la complejidad de las interacciones de tipo a la clase Elemento.

2. Gestión del Estado de Objetos (Atributos)
Seguimiento Dual de PP: La decisión de incluir dos atributos para los Puntos de Poder (PPMaximo y PPActual) es fundamental.

PPMaximo modela la capacidad intrínseca del ataque (un valor estático).

PPActual modela el estado mutable del ataque durante el juego (el contador de uso).

Esta distinción es crucial para la simulación precisa del estado de batalla y la posterior restauración.

Modelo de Ataque Físico: Los atributos potencia y precision son decisiones de diseño que definen los valores numéricos directos utilizados en el cálculo de daño y la probabilidad de acierto.

Mutabilidad Completa: Al igual que otras clases de modelo, la presencia de setters para todos los campos hace que una instancia de Ataque sea completamente mutable, lo que es esencial para actualizar el PPActual después de cada uso del ataque.

3. Acoplamiento e Interfaz de Usuario
Constructor con Asignación Directa: El constructor requiere la inicialización de todos los atributos, asegurando que cada instancia se cree con un estado de batalla válido (incluyendo PPActual). Es importante notar que, en el código Inicializar, la decisión de diseño fue que, al crear nuevos ataques, siempre se inicializa PPActual igual a PPMaximo, indicando un ataque "recargado".

Sobrescritura Concisa de toString(): Se sobrescribe toString() para devolver solo el nombre. Esto simplifica la presentación del objeto en listas o mensajes de consola.

Separación de Comportamiento: La clase se adhiere al rol de modelo de datos al no contener lógica de comportamiento de batalla (como usarAtaque() o reducirPP()). Se espera que estas operaciones sean gestionadas por la clase Batalla, manteniendo a Ataque enfocado en sus propiedades.

BATALLA

1. Patrones y Estructura Principal
Clase de Servicio de Dominio (Motor de Reglas): La clase Batalla encapsula la totalidad de la lógica de la simulación del combate. Su propósito principal, definido por el método cicloBatalla, es gestionar las reglas complejas (cálculo de daño, efectividad, turnos) del juego, adhiriéndose al Principio de Responsabilidad Única para la simulación.

Flujo de Control de Bucle Infinito (While Loop): El método cicloBatalla utiliza un bucle while(enCombate) con múltiples puntos de retorno (return true/return false). Esta es una decisión de diseño estándar para modelar un proceso continuo (la batalla) que solo termina al alcanzar una condición terminal (victoria o derrota).

Dependencia Fuerte del Scanner: El método cicloBatalla crea una instancia de Scanner y llama repetidamente a sc.nextLine() o sc.nextInt(). Esto genera un alto acoplamiento entre la lógica del juego y la interfaz de usuario de consola, haciendo la clase menos reutilizable en un entorno que no sea de consola.

2. Gestión del Estado de Objetos (Comportamiento y Mutación)
Concentración de la Mutación de Estado: El método atacar() es el punto central donde se implementa la mutación de estado de los objetos de modelo. Dentro de este método:

Se reduce el PPActual del ataque usado.

Se reduce el HPActual del Pokémon defensor.

Se establece el estado debilitado = true del defensor si su HP llega a cero.

Métodos de Verificación de Estado: Se utilizan métodos como comprobarSalud, comprobarPokemonesJugador, y comprobarPokemonesNPC para consultar el estado actual de la batalla (si un Pokémon activo cayó o si un entrenador se quedó sin Pokémon). Esto separa la consulta de la mutación.

Fórmula de Daño Personalizada (calcularDaño): La decisión de reimplementar la fórmula de daño de Pokémon es clave, incluyendo elementos de diseño del juego original como:

La dependencia de Nivel, Potencia, Ataque y Defensa.

Multiplicadores fijos para Efectividad y Crítico.

Un Factor Aleatorio (Random Factor) de (0.85 + Math.random() * 0.15) para simular la variabilidad natural del daño.

3. Métodos y Lógica Estática/Dinámica
Métodos Estáticos de Utilidad: Los métodos limpiarPantalla(), esDebilidad(), esFortaleza(), y obtenerEfectividad() son declarados static. Esta decisión los establece como funciones de utilidad pura que operan sobre sus parámetros sin requerir el estado de una instancia de Batalla.

limpiarPantalla() intenta resolver una necesidad de la interfaz de usuario específica del sistema operativo.

La Lógica de Tipos (obtenerEfectividad) se centraliza aquí, decidiendo que solo existe efectividad x0.5, x1.0, o x2.0, simplificando la complejidad de los tipos duales (ya que el código suma los efectos: 1.0×2.0=2.0).

Implementación de IA Simple (ataqueIA): La decisión de diseño de la IA se reduce a una selección de ataque completamente aleatoria (rand.nextInt(4)). Esto asegura que el NPC siempre ataque, pero no implementa ninguna lógica estratégica (como elegir ataques super efectivos).

Control de Prioridad (Velocidad): El método compararVelocidades introduce el concepto de prioridad de turno basado en el atributo de velocidad (SPD), una característica fundamental en el diseño de los juegos de Pokémon, determinando qué bloque de ataque se ejecuta primero en el cicloBatalla.

GIMNASIO

1. Patrones y Estructura Principal
Modelo de Agregación de Alto Nivel: La clase Gimnasio es un modelo de datos que actúa como un contenedor o agregador de instancias de Entrenador. Modela un concepto de juego de alto nivel, decidiendo que una instalación, como un gimnasio, se define principalmente por su nombre y los oponentes que contiene.

Límite Fijo de Agregación: La inicialización del campo entrenadores con un arreglo de tamaño fijo (new Entrenador[6]) es una decisión de diseño para imponer un límite máximo de 6 entrenadores que pueden pertenecer al gimnasio. Aunque la inicialización en Inicializar usa un arreglo más pequeño, el modelo prevé una capacidad máxima de 6.

2. Gestión del Estado de Objetos (Atributos)
Constructor Basado en Identidad y Contenido: El constructor exige el nombre y el arreglo de entrenadores al crearse. Esta es una decisión de diseño para garantizar que cada gimnasio esté completamente definido y poblado con su identidad y contenido inmediatamente después de su instanciación.

Mutabilidad de Contenido: La inclusión del setEntrenadores() y el getEntrenadores() que devuelve la referencia directa al arreglo hace que el contenido del gimnasio sea mutable. Esto permitiría cambiar completamente la lista de oponentes en tiempo de ejecución, lo cual es útil para escenarios de juego dinámicos o restablecimiento de contenido.

Falta de Encapsulamiento Estricto: Los atributos se declaran con el modificador de acceso por defecto (package-private) en lugar de private. Si bien esto permite el acceso dentro del mismo paquete, la provisión de getters y setters públicos anula cualquier beneficio estricto de ocultación de datos que podría haberse logrado con private.

3. Métodos y Representación
Representación Concisa (toString()): El método toString() se sobrescribe para devolver solo el nombre del gimnasio. Esta es una decisión de diseño que prioriza la legibilidad al referirse a la instalación en la consola.

Falta de Comportamiento Lógico: La clase no contiene ningún método de lógica de juego (ej. iniciarDesafio(), otorgarMedalla(), comprobarProgreso()). Esto mantiene la clase puramente como un modelo de datos, delegando toda la lógica de interacción y control del desafío (quién se enfrenta a quién y en qué orden) a una clase controladora externa, presumiblemente a la clase Main o una clase de Juego (no mostrada).

ENTRENADOR

1. Patrones y Estructura Principal
Modelo de Agregación (Relación "Tiene un"): La clase Entrenador modela al usuario o al oponente mediante la agregación de la clase Pokemon. La decisión de diseño clave es que un entrenador contiene una colección de Pokémon (Pokemon[] pokemones = new Pokemon[3];), que se establece en un límite fijo de 3, simplificando la gestión de equipos a un tamaño predefinido.

Encapsulamiento de Identidad: Los campos de instancia (nombre, esJugador, pokemones) se mantienen implícitamente a nivel de paquete (por defecto, no son private ni public), aunque las prácticas modernas de POO sugieren el uso de private para un encapsulamiento más estricto. A pesar de esto, se proporcionan getters y setters explícitos para gestionar el acceso.

Identificación de Rol: El campo boolean esJugador es una decisión de diseño fundamental para distinguir en tiempo de ejecución entre el jugador humano y la Inteligencia Artificial (IA) (NPCs), permitiendo a otras clases (como Batalla) aplicar la lógica de turno, I/O, o IA adecuada.

2. Gestión del Estado de Objetos (Atributos)
Constructor de Inicialización Completa: El constructor es una decisión de diseño para forzar la inicialización de todos los atributos (nombre, esJugador, y la colección de Pokemon) en el momento de la creación de la instancia, garantizando un estado inicial válido para el entrenador.

Mutabilidad del Estado: La inclusión de setters para todos los atributos hace que la instancia de Entrenador sea completamente mutable. Esto permitiría, teóricamente, cambiar el nombre, el rol, o incluso el equipo completo del entrenador después de la creación, aunque en el flujo de juego actual solo se utiliza para la inicialización.

Manejo de Colección: La decisión de exponer el arreglo de Pokémon directamente a través de getPokemones() (y setPokemones()) es simple de implementar, pero rompe el encapsulamiento. Esto permite que el código externo acceda y modifique directamente el contenido del arreglo (el estado de los Pokémon) sin pasar por lógica de control dentro de la clase Entrenador.

3. Métodos y Representación
Sobrescritura de toString(): El método toString() se sobrescribe para devolver el nombre. Esta es una decisión de diseño para una representación concisa y significativa en contextos de presentación o depuración (por ejemplo, mostrando quién es el oponente).

Convención Booleana: Se utiliza la convención de Java isJugador() para el getter de la propiedad booleana, mejorando la legibilidad del código.

Falta de Comportamiento: La clase se enfoca puramente en el modelo de datos y la identidad. Es notable que no contenga ningún método de comportamiento de batalla (ej. elegirPokemon(), lanzarAtaque()). Al igual que con Pokemon, la lógica de decisión y la interacción se delega completamente a la clase Batalla, manteniendo a Entrenador como un simple portador de datos y rol.

ELEMENTO

1. Patrones y Estructura Principal

Modelo de Datos de Clasificación: La clase Elemento está diseñada para modelar el sistema de tipos o categorías elementales del juego (ej., "Fuego," "Agua"). Su propósito es definir las reglas de interacción para cada tipo, delegando la lógica de efectividad a la clase Batalla.

Encapsulamiento del Sistema de Reglas: Los atributos se declaran private (nombre, debilidades, fortalezas), encapsulando la información de las reglas de tipo. Esto protege los datos base y asegura que la lógica de efectividad (Batalla.obtenerEfectividad) trabaje con datos consistentes a través de los getters.

Representación de Reglas por Cadenas: La decisión de diseño clave es que los atributos debilidades y fortalezas son arreglos de cadenas de texto (String[]) en lugar de arreglos de objetos Elemento. Esto simplifica la inicialización (solo se necesita el nombre del tipo, como se ve en la clase Inicializar) y simplifica la comparación de tipos en el método Batalla.esDebilidad().

2. Gestión del Estado de Objetos (Atributos)
Constructor con Carga de Reglas: El constructor requiere el nombre y las reglas de interacción (debilidades y fortalezas) al crearse. Esta es una decisión de diseño para asegurar que cada instancia de tipo se inicialice con sus reglas de efectividad definidas inmediatamente, garantizando un estado completo.

Mutabilidad de los Tipos (Teórica): La inclusión de setters para todos los atributos (incluyendo los arreglos de reglas) hace que la instancia de Elemento sea mutable. Esto permitiría, teóricamente, cambiar las reglas del juego en tiempo de ejecución. Sin embargo, en el contexto de la clase Inicializar, estas instancias actúan como datos maestros estáticos, por lo que su mutabilidad rara vez se utiliza en el flujo de batalla.

Sobrescritura Concisa de toString(): Se sobrescribe toString() para devolver solo el nombre, lo cual es esencial para una representación legible del tipo cuando se usa dentro de la presentación de los Pokémon en la consola.

3. Arreglos y Acoplamiento
Exposición de Arreglos en Getters: Los métodos getDebilidades() y getFortalezas() devuelven directamente referencias a los arreglos internos (String[]). Esta decisión simplifica el código de Batalla (que itera sobre ellos directamente) pero rompe el encapsulamiento, ya que el código externo podría modificar el contenido de estos arreglos de reglas sin que la clase Elemento lo sepa.

Acoplamiento con la Lógica Externa: La clase Elemento es un modelo de datos que está altamente acoplado a la lógica del juego implementada en Batalla (específicamente obtenerEfectividad). Se limita a proporcionar los datos brutos para que la clase Batalla pueda ejecutar sus comparaciones y aplicar los multiplicadores de daño.

Estructura de paquetes

La estructura de instancias, basada en las 9 clases enviadas y cómo se utilizan o definen sus objetos, se presenta a continuación, agrupada por clase.

Estructura de Instancias por Clase

1. Clase `Main` (Orquestador Principal)

Clase Instanciada	Nombre de la Instancia	Cantidad	Propósito
`Batalla`	**`b`**	1	Motor de reglas de combate.
`Inicializar`	**`i`**	1	Creador de datos base (Factory).
`Scanner`	**`s`** (en `main`), **`y`** (en `showStats`)	2	Manejo de entrada de consola.
`Entrenador`	**`jugadorOriginal`**	1	Plantilla de datos del jugador (base inmutable).
`Entrenador`	**`jugador`**, **`npc`**	2 (por ciclo)	Copias mutables usadas en cada batalla.
`Pokemon`	**`pokemonesJugador`**, **`pokemonesNPC`**	6 (3 por cada arreglo, por ciclo)	Copias mutables de los Pokémon (Deep Copy).

---

2. Clase `Inicializar` (Fábrica de Datos)

Clase Instanciada	Nombre de la Instancia	Cantidad	Propósito
`Batalla`	**`b`**	1	Uso estático para `limpiarPantalla()` y otros métodos.
`Scanner`	**`scanner`** (en `inicializarJugador`)	1	Manejo de entrada para la selección inicial.
`Elemento`	**`poolElementos`**	7	Creación de tipos de Pokémon.
`Ataque`	**`poolAtaques`**	14	Creación del repertorio de movimientos.
`Stat`	**`stats...NPC`**, **`stats...J`**	20	Creación de todas las estadísticas base (10 NPC, 10 Jugador).
`Pokemon`	**`poolPokemonesNPC`**	10	Plantillas de Pokémon para NPC.
`Pokemon`	**`poolPokemonesJugador`**	10	Plantillas de Pokémon para Jugador.

Entrenador	**poolEntrenadores**	4	Creación de oponentes NPC.
Entrenador	(Devuelto por inicializarJugador)	1	El objeto Entrenador del jugador.
Gimnasio	**poolGimnasios**	3	Contenedores de grupos de entrenadores (aunque se usa el mismo pool 4 veces).

---

3. Clase Batalla (Motor de Combate)

Clase Instanciada	Nombre de la Instancia	Cantidad	Propósito
Random	**random** (en esCritico), **rand** (en ataqueIA)	2 (en métodos de instancia)	Generación de números aleatorios (crítico, ataque NPC).
Scanner	**s** (en atacar), **scan** (en cambiarPokemonJugador), **x** (en cambiarPokemonNPC), **sc** (en cicloBatalla)	4 (local en métodos)	Manejo de entrada y pausas durante la batalla.
ProcessBuilder	(Anonima en limpiarPantalla)	1 (en método estático)	Ejecución del comando de limpieza de pantalla.

---

4. Clases de Modelo (Solo Proporcionan la Estructura de Datos)

Las siguientes clases son de modelo y **no crean instancias de otras clases** dentro de sí mismas, sino que **componen** o **agrupan** instancias que les son pasadas por otras clases (principalmente inicializan):

Clase	Instancias Creadas Internamente	Estructura de Composición (Recibidas)
Pokemon	0	Compone: 1 Stat, 2 Elemento[], 4 Ataque[].
Ataque	0	Compone: 1 Elemento.
Entrenador	0	Agrega: 3 Pokemon[].
Gimnasio	0	Agrega: Entrenador[] (máx. 6).
Stat	0	Contiene solo tipos primitivos (int).
Elemento	0	Contiene solo tipos primitivos y arreglos de String.

Limitaciones

Las limitaciones de este programa se basan principalmente en, dentro de la parte logica, que no hemos podido optimizar al maximo el programa, y ya en la parte de experiencia del

usuario, se basan en que no existe una interfaz gráfica, es únicamente texto plano, por lo que la experiencia es mediocre en cuanto al interés del usuario a usar la aplicación.

Chat con la IA

El recurso de la inteligencia artificial fue empleado mayormente para saber en el momento en que ocurría un error en el código, el significado del error, pero sin la solución del error. Y se usó también para cuando el código no funcionaba, buscar la lógica de sobre cómo debería funcionar el código.

Futuro

Para el futuro se podría implementar una interfaz gráfica.
Que el jugador pueda tener más de tres pokemones o pueda elegir la cantidad de pokemones con los que quiere combatir, ya sea uno o seis.
Se podría mejorar la IA para que escoja ataques más basados en una estrategia, como querer aplicar más daño por efectividad de elementos por ejemplo.