Índice

1. Introducción

1.1. Planteamiento del problema

El desarrollo de videojuegos como Pokémon implica la gestión de sistemas donde interactúan múltiples entidades con comportamientos diversos. El problema central consiste en modelar un sistema de batalla por turnos que gestione estadísticas dinámicas (vida, velocidad), mecánicas de daño basadas en tipos elementales (fuego, agua, planta, etc.) y condiciones de estado (quemado, paralizado). Además, se debe implementar una lógica de para el rival que sea capaz de tomar decisiones estratégicas, todo esto integrado bajo una arquitectura de software escalable orientada a objetos que sirva como backend para una futura interfaz gráfica en Flutter.

1.2. Motivación

La motivación principal es aplicar los conceptos de la Programación Orientada a Objetos (POO) en un escenario real. al simular las mecánicas de Pokemon, se destaca la necesidad de la herencia para compartir atributos comunes, el polimorfismo para diferenciar el comportamiento de ataques e ítems, y el encapsulamiento para proteger la integridad de los datos de la batalla. También, este proyecto permite explorar el manejo de colecciones y excepciones en Dart, preparandonos para el desarrollo de aplicaciones móviles.

1.3. Objetivos

- Implementar la mayor parte de los conceptos vistos durante la asignatura de Programación Orientada a Objetos
    - Definir una jerarquía de clases eficiente utilizando abstracción y herencia para representar Pokémons, Ataques e Ítems
    - Implementar el sistema de batalla: Desarrollar la lógica de turnos basada en la velocidad de los personajes, calculando el daño según la tabla de efectividad de tipos y aplicando condiciones de estado (envenenado, quemado, paralizado)
    - Implementar una lógica de decisión para el rival que priorice curar estados alterados o recuperar vida antes de atacar, utilizando manejo de excepciones para verificar la disponibilidad de recursos.
- Crear un juego funcional a partir de los conceptos teóricos de la POO

2. Marco Teórico

Clases: son modelos o abstracciones de la realidad que representan un elemento de un conjunto de *objetos* [3]
Objeto: es la estructura o entidad abstracta que se crea a partir de la definición de una *clase*, de tal manera, que ésta permite organizar y relacionar los *datos* de un programa o problema (*atributos*) con sus respectivos *métodos*. [6]

- **Atributos**: Conjunto de rasgos o características que posee un objeto [6]
  - **Encapsulamiento:** Es el principio que consiste en ocultar los detalles internos de una clase y exponer solo lo necesario.[6]
    Dart utiliza un guion bajo al inicio del nombre de una propiedad o clase para volverla privada.
  - **Inmutabilidad (final):** En Dart, la palabra clave **final** se utiliza para declarar atributos que solo pueden ser asignados una única vez. Una vez que un atributo final es inicializado, su valor no puede ser modificado. Esto es crucial para la integridad de datos que no deben cambiar durante la ejecución.[6]
  - **Inicialización Tardía (late):** En Dart la palabra clave **late** permite declarar una variable no nula que será inicializada después de su declaración, pero antes de su primer uso.
- **Métodos**: Conjunto de procedimientos o funciones para la manipulación o interacción con los datos de un objeto.[6]
  - **Constructor:** Método que tiene el mismo nombre que la clase y cuyo propósito es inicializar los atributos de un nuevo objeto. Se ejecuta automáticamente cuando se crea un objeto o instancia de la clase.[20]
  - **Método estático:** métodos que no requieren un objeto para ser invocados, se pueden ejecutar a través de la clase. [20]
  - **Método de instancia:** Estos métodos operan en instancias de una clase y pueden acceder a los campos y otros métodos de esa instancia. Son invocados a través de un objeto creado a partir de la clase. [15]

En Dart, todo es un objeto y todas las clases heredan de la clase Object. Las clases se definen mediante la palabra clave **class**. La instanciación de objetos no requiere necesariamente la palabra clave new (aunque es permitida), prefiriendo la sintaxis directa:

```
var objeto = Clase();
```

Herencia [5]
En términos simples, se puede definir como contruir "algo" a partir de cimientos ya existentes. Permite definir nuevas clases basadas en clases existentes, aprovechando y extendiendo su funcionalidad.

En Dart se utiliza la palabra clave extends. Dart soporta herencia simple (una clase solo puede heredar de una sola clase padre).

- **Super Clase**
  Se define una sola clase base, la cual contendrá los atributos y métodos comunes que deseamos compartir con las clases derivadas.

- **Clases derivadas**
  Se crea la clase hija, la cual como su nombre lo indica, será la que hereda de la clase padre.
  Con la *herencia* ahora se pueden crear objetos de la clase derivada y acceder a los atributos y métodos tanto de la clase derivada como de la clase base.

Ventajas:

- Reutilización del código
- Abstracción y generalización
- Uso de **polimorfismo**

Polimorfismo [5]
Es la habilidad de una clase para implementar métodos con el mismo nonbre, pero con comportamientos distintos dependiendo de la clase con la que se este interactuando.

- Crear clase base
  Se definen los métodos que deamos que las clases derivadas implementen.

- Clase derivada
  Se crean las clases que hereden de la clase padre. Cada clase debe proporcionar su propia implementación del método definido en la clase padre.
  Ahora se pueden crear objetos de las clases derivadas y tratarlos como objetos de la clase base.

Sobrescritura [16]
Permite que una sub-clase proporcione su propia implementación del método para adaptarlo a su comportamiento.
Ocurre cuando una sub-clase redefine un método heredado de su clase padre con:

- Mismo nombre
- Mismo tipo y número de parámetros.
- Mismo nivel de acceso.
- Mismo tipo de valor de retorno.

@Override *(se indica que se está sobrescribiendo un método)*
Para acceder:
La palabra clave **super** se utiliza para comunicar métodos y atributos de la clase base a una clase derivada, permitiendo llamar a los contructores, métodos o acceder a los atributos de la clase padre desde la clase hija.
Esto es útil si deseamos agregar funcionalidad adicional en la sub-clase, pero también queremos preservar el comportamiento original del método de la clase base.

Diagrama UML (Lenguaje Unificado de Modelado) [4]
El **Lenguaje Unificado de Modelado** (UML, por sus siglas en inglés) es un lenguaje gráfico estándar y formal para la industria del software. No es un lenguaje de programación, sino un conjunto de *diagramas* que actúan como planos. Se utiliza para visualizar, especificar, construir y documentar los diferentes artefactos y vistas de un sistema de software complejo, facilitando la comunicación entre desarrolladores, analistas y clientes.

- **Diagrama de Clases**:
  Es el diagrama más común de UML y pertenece a los diagramas de **estructura** (estáticos). Describe la estructura del sistema mostrando sus clases, atributos (datos), operaciones (métodos) y las relaciones estáticas entre ellas (como herencia, asociación, agregación o composición). [4]

- **Diagrama de Secuencia:**
  Este es un diagrama de **comportamiento** (dinámico), enfocado en la interacción. Muestra cómo los objetos colaboran entre sí a lo largo del *tiempo*. Se centra en el orden cronológico (la secuencia) de los mensajes que se envían y reciben entre diferentes objetos (representados como "líneas de vida") para realizar una tarea o un caso de uso específico. [4]

Enumeraciones
[9] Un tipo de dato especial que permite definir un conjunto de constantes con nombre. Implementación en Dart: Se usa la palabra clave **enum**, lo que mejora la legibilidad y reduce errores comparado con el uso de cadenas de texto o enteros.

Excepciones [14]
Una excepción es una condición anormal que cuando ocurre altera el flujo normal del programa en ejecución. Estos errores pueden ser generados por la lógica del programa, como un índice de un arreglo fuera de su rango, una división entre cero o errores generados por los propios objetos que denuncian algún tipo de estado no previsto o condición que no pueden manejar.
Beneficios del manejo de excepciones:

- *Separación de lógica*: Permite separar el código de lo que el programa debe hacer del código de manejo de errores

- Las excepciones pueden ”subir” a través de la pila de llamadas (call stack) hasta encontrar un método que sepa cómo manejarlas, centralizando la gestión de errores.

- Evita que el programa colapse ante datos de entrada incorrectos o fallos de recursos externos (como un archivo no encontrado)

Manejo de excepciones en Dart [8]
En Dart, las excepciones son objetos. El lenguaje proporciona las palabras clave *try, catch, on, finally y throw*:

- *throw:* Se utiliza para detener la ejecución y generar una excepción explícitamente cuando se detecta un problema.
- *try:* Define un bloque de código donde podría ocurrir una excepción.
- *catch:* Bloque que captura la excepción lanzada en el try y permite manejarla (por ejemplo, imprimiendo un error).
- *on* Se usa cuando se quiere capturar un tipo específico de excepción.

Concurrencia y Asincronía
A diferencia de lenguajes tradicionales que usan múltiples hilos (multi-threading) por defecto para tareas pesadas, Dart es un lenguaje **Single-Threaded** (de hilo único). Esto significa que ejecuta las instrucciones una por una en un bucle principal llamado **Event Loop**. [2]

Si una tarea pesada (como cargar un archivo de música grande) se ejecutara en este hilo principal de forma síncrona, la interfaz gráfica se congelaría, dejando de responder a los toques del usuario. Para solucionar esto, se utilizan conceptos de asincronía: [1]

- **Future**: Es un objeto que representa el resultado de una operación asíncrona que se completará en el futuro (puede ser un valor o un error). Es una "promesa" de que el dato llegará
- **async / await**: Son palabras clave que permiten escribir código asíncrono que parece síncrono
  - **async:** Marca una función para indicar que realizará operaciones que toman tiempo
  - **await:** Pausa la ejecución de esa función específica (sin bloquear el resto de la app) hasta que la tarea (el *Future*) se complete.

Patrones de Diseño
Los patrones de diseño son soluciones probadas y reutilizables para problemas comunes en el desarrollo de software. En la implementación de la interfaz gráfica y la lógica de batalla, se destacan los siguientes:

- **Patrón Observer (Observador):**
  Es el patrón fundamental de la arquitectura de Flutter. Define una dependencia "uno-a-muchos" entre objetos, de modo que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente [18]

- **Patrón Command (Comando):** Este patrón encapsula una solicitud como un objeto, permitiendo parametrizar a los clientes con diferentes solicitudes. [17]
- **Patrón State (Estado):** Permite que un objeto altere su comportamiento cuando su estado interno cambia [19]

pubspec.yaml

El archivo pubspec.yaml es el centro de configuración y gestión de metadatos para cualquier proyecto desarrollado en Dart y Flutter. [12]. Utiliza el formato **YAML** (*Yet Another Markup Language*), que es legible por humanos y jerárquico. Este archivo cumple tres funciones críticas en el ciclo de vida del software:

- **Gestión de Dependencias:** Especifica qué librerías externas (paquetes) necesita el proyecto para funcionar. En este caso, se define la dependencia de *audioplayers*, permitiendo al gestor de paquetes de Dart descargar e integrar el código necesario para la reproducción de audio
- **Gestión de Recursos (Assets)**: Flutter no incluye imágenes o archivos de sonido en la compilación final por defecto. Es necesario declarar explícitamente en la sección flutter: assets: las rutas de las carpetas para que el compilador las empaquete dentro del binario de la aplicación. [7]
- **Metadatos y Versionado:** Define el nombre de la aplicación, su descripción, y la versión semántica del proyecto, así como las restricciones del entorno (versión del SDK de Dart requerida).

Colecciones

Las colecciones son estructuras de datos que permiten agrupar múltiples objetos en una sola unidad. Son esenciales en POO para manejar conjuntos de elementos dinámicos.

- Listas (**List**): Es una colección ordenada de objetos. [10]
- Mapas (**Map**): Es una colección de pares clave-valor, donde cada clave es única.[11]

Seguridad Nula (Null Safety): Es una característica moderna de los lenguajes de programación que garantiza que las variables no puedan contener un valor nulo (null) a menos que se declare explícitamente. Esto elimina una categoría entera de errores comunes en tiempo de ejecución.
[13] En Dart: Por defecto, las variables son no-nulas (int a).
La seguridad nula nos permite usar late para prometerle a Dart que esas variables tendrán valor antes de usarse, evitando tener que verificar si son nulas.

3. Desarrollo

3.1. Explicación de Códigos Fuente

A continuación, detallamos la estructura del proyecto, usando el paradigma orientado a objetos

1. **Base.dart**
   Aquí es donde definimos qué son las cosas (Abstracción) y cómo se comportan (Polimorfismo)

   a) **Interfaz Combate**
   - **Función:** Define el "contrato" obligatorio para cualquier pokemon que participe en la pelea
   - **Lógica:**
     Al ser una interfaz (implementada mediante *abstractclass* en Dart), no contiene código funcional, sino solo la definición de qué debe saber hacer un luchador. Esto garantiza el polimorfismo: el sistema puede tratar a cualquier objeto que implemente *Combate* por igual.
   - **Métodos:**
     - $atacar(objetivo, movimiento)$: Método abstracto, obliga a definir cómo se ejecuta una acción ofensiva
     - $recibirDanio(cantidad, prob\_estado)$: Método abstracto void, obliga a definir cómo se procesa el daño entrante y los efectos secundarios
     - $mostrarAtaque()$: Método para visualizar la preparación de la acción

   b) **Clase Abstracta Pokemon** (Implementa Combate)
   - **Función:** Sirve como la plantilla para todos los pokemones del juego
   - **Lógica:**
     Define los atributos de datos que todos los pokémon comparten: nombre (identidad), tipo (para la tabla de efectividad), velocidad (para los turnos) y vida/vidaMax (para la salud). *Implementa la interfaz* ***Combate***, pero al ser abstracta, delega la implementación detallada de los métodos de pelea a sus hijos
   - **Métodos:**
     $Pokemon(...)$: Constructor que inicializa los atributos y la vida máxima.

   c) **Clase Ataque**
   - **Función:** encapsula las propiedades de una acción ofensiva de los pokemones
   - **Lógica:**
     Almacena el daño base y el tipo elemental. Incluye la propiedad $prob\_estado$ para definir si el ataque causa efectos secundarios.

- **Métodos:**
  calcularDanio(): Retorna el daño final (*int*) multiplicando el valor base por el factor de efectividad (*double*).

d) **Clase Efectividad**

- **Función:** es el "arbitro" de las reglas de la batalla.
- **Lógica:**
  Usa un $Map < String, Map < String, double\check{}$ (un mapa con un mapa dentro) para representar la tabla de multiplicadores (x2.0, x0.5, x0.0). *Esto evita el uso de múltiples condicionales anidados, optimizando la búsqueda*
- **Métodos:**
  $calcularEfectividad(tipoAtaque, tipoEnemigo)$: Busca en el mapa la intersección de tipos. Si no existe la clave, retorna 1.0 por defecto (daño neutro).

e) **Clase PokemonBatalla** (Hereda de Pokemon)

- **Función:** Representa a la entidad en el combate. Hereda atributos base (nombre, vida) e implementa la interfaz *Combate*.
- **Lógica:**
  Gestiona el estado del personaje. Aplica el *patrón State* mediante el ***enum*** *EstadoCondicion* para alterar su comportamiento (por ejemplo reducir daño si está quemado)
- **Métodos:**
  - $atacar(objetivo, movimiento)$: Primero verifica si el estado actual permite moverse (si está congelado, retorna). Calcula el daño usando la clase *Efectividad* y llama al método *recibirDanio* del objetivo
  - $recibirDanio(cantidad, prob\_estado)$: Método void. Resta la cantidad a la vida. Genera un número aleatorio; si es menor a $prob\_estado$, cambia el *estadoActual* del *Pokémon* (por ejemplo a EstadoCondicion.quemado)
  - $finDeTurno()$: sw ejecuta al cerrar el ciclo de ataque, aplica daño (resta HP fija) si el Pokémon está quemado o envenenado, y calcula la probabilidad de curarse de estados como parálisis.

f) **Clase Abstracta Item**

- **Función:** Plantilla base para el inventario, aplicando **Polimorfismo**
- **Lógica:**
  Define *usar()*, e incluye para la parte gráfica *imagenPath* para vincular la lógica con los *assets* gráficos definidos en el *pubspec.yaml.*
- **Métodos:**
  $usar(objetivo)$: Método abstracto que Obliga a las subclases a definir cómo afecta el ítem al Pokemon

g) **Clases Pocion y CuradorEstado** (Heredan de Item)

- **Función:** Implementaciones concretas de ítems
- **Lógica:**

  - **Pocion**: Su método *usar* suma una cantidad numérica a la vida del objetivo, sin exceder *vidaMax*
  - *CuradorEstado:* Su método *usar* verifica si el *estadoACurar* coincide con el del objetivo. Si es así, restablece el estado a *normal*.

2. **SistemaBatalla.dart**

   a) **Clase Batalla**

   - **Función:** Controla el flujo de juego
   - **Lógica:**
     Reacciona a la entrada del usuario y devuelve una lista de textos ($List < String > logTurno$) para actualizar la UI. Implementa el patrón Command mediante la clase *AccionTurno* para tratar ataques e ítems
   - **Métodos:**
     - $ejecutarTurnoJugador(ataque)$: Recibe el ataque seleccionado, solicita la decisión de la IA rival y llama a $\_determinarYEjecutar$, y retorna el log actualizado
     - $\_determinarYEjecutar(accionUsuario, accionRival)$: Compara la *velocidad* de ambos (aplicando penalización si hay parálisis). Ejecuta la acción del más rápido primero, verifica si alguien se debilitó, y luego ejecuta al segundo.
     - $\_convertirDecisionRival(decision)$: Transforma la instrucción en texto del rival (ej. "$USAR\_POCION$") en un objeto *AccionTurno* ejecutable.
     - $\_verificarFinBatalla()$: Método booleano, revisa si la *vida* de alguno llegó a 0. Si es así, cambia *enCurso* a *false*.

3. **ComportamientoRival.dart**: Representa la "inteligencia artificial (IA)" del rival
   Decidimos nombrarlo de esta manera ya que es la computadora quien está "jugando"

   a) **Clase ComportamientoRival**

   - **Función:** Toma decisiones estratégicas para el turno del rival
   - **Lógica:**
     Utiliza un sistema de prioridades y Manejo de Excepciones para la gestión de recursos.
   - **Métodos:**

- *tomarDecision*(*rival*, *items*): Retorna un String. Evalúa condiciones en orden: 1) Curar estado, 2) Curar vida, 3) Atacar.
  ->**Uso de Excepciones**: Utiliza un bloque *try-catch* al buscar ítems con *.firstWhere()*. Si el rival no tiene el ítem necesario, la excepción StateError es capturada, permitiendo que la "IA" degrade su decisión a "ATACAR" en lugar de cerrar la aplicación por error
- *elegirAtaqueAleatorio*() : Selecciona un movimiento al azar de la lista de ataques

4. **Interfaz Gráfica**

   - **SelectionScreen.dart**
     - **Función:** Pantalla de selección de personajes y música.
     - **Lógica:**
       Implementa **Hilos y Asincronía** para la carga de recursos pesados.
     - **Métodos:**
       - *_iniciarMusica*(): Función marcada como **async**. Utiliza la palabra clave **await** para cargar el archivo de audio desde los *assets* sin bloquear el Event Loop (hilo principal) de la interfaz, asegurando que la app no se congele durante la carga.
   - **BattleScreen.dart**
     - **Función:** Vista principal del combate.
     - **Lógica:**
       Implementa el patrón **Observer** a través de la gestión de estado de Flutter.
       - *realizarAtaque*(): Invoca a la lógica de Batalla. Usa *setState*(()...) para notificar que los datos han cambiado, forzando a los widgets (barras de vida, texto) a redibujarse con la nueva información

5. **Configuración (pubspec.yaml)**

   - **Función:** Archivo de configuración y *metadatos* del proyecto.
   - **Lógica:**
     Gestiona la inyección de dependencias y el acceso a archivos locales
     - *dependencies* : *audioplayers*: Importa código externo para manejo de hilos de audio
     - *assets* :: Declara las rutas *assets/images/* y *assets/music/* para que sean empaquetadas en la aplicación final

3.2. Interfaz

En POO la interfaz de usuario se programa como un árbol de objetos que interactúan. Todo lo que vemos en la pantalla (botones, una barra de vida, uma imagen,

etc.) es un **widget (un objeto)**, en Dart la mayoría de los widgets heredan de dos clases principales:

- **Widget sin estado (StatelessWidget)**: se usan para elementos que nunca cambian después de ser construidos, por ejemplo, una imagen fija.
  Heredar de esta clase es implementar **abstracción**, ya que el widget solo se preocupa por como se ve.

- **Widget con estado (StatefulWidget)**: se usan para elementos dinámicos que cambian durante la batalla, por ejemplo, la barra de vida de un pokémon. El widget hereda la capacidad de **mutar** su apariencia según las respuestas a eventos como un ataque.

  - **Composición:**
    La pantalla de batalla completa es un widget grande, este widget está compuesto por widgets más pequeños (objetos) que trabajan juntos. Este uso de la composición nos permite reutilizar código, por ejemplo, el *WidgetBarraDeVida* es un solo objeto que se puede utilizar dos veces (uno para el jugador y otro para el rival), pero con datos distintos.

- Carpeta **assets**, la cual es donde guardamos todos los recursos estáticos que necesitamos como imágenes de los pokémon, iconos, fuentes personalizadas o cualquier archivo de datos.

  1. Se creo una carpeta llamada **assets** y colocamos allí todas las imágenes de los pokémon.
  2. Para que el sistema de compilación de la aplicación (Flutter) sepa que debe incluir estos archivos, es necesario declarar la ruta en el archivo *pubspec.yaml*
  3. Una vez declarado, cualquier widget en el código puede acceder a la imagen usando la ruta completa.

- **Gestión del estado**
  Es la forma en que el objeto de la interfaz **(StatefulWidget)** sabe cuándo y cómo debe actualizar la pantalla:

  - Método **setState**: es el mecanismo de **encapsulamiento y notificación** que garantiza que todos los cambios de datos realizados por la lógica se reflejen eficientemente y en tiempo real en la interfaz gráfica
    - Marca el objeto: le dice al framework (Flutter) que el objeto de estado (State del widget) ha cambiado internamente.
    - Solicita redibujo: fuerza a que el método **build** del widget (el método que realiza la interfaz) se ejecute de nuevo.
    - El framework compara el nuevo resultado del **build** con lo que ya está en la pantalla, solo modifica las partes de la interfaz que realmente cambiaron como la barra de vida y el texto de daño.

3.3. Puntualización de Aspectos de la POO implementados

A continuación, describiremos la implementación de la lógica del sistema de batallas en lenguaje Dart, relacionando cada parte con los conceptos teóricos fundamentales de la POO.

- **Abstracción:**
  Se modelaron las clases base ***Pokemon*** e ***Item*** para definir las plantillas esenciales (vida, nombre, método usar), ocultando la complejidad interna y obligando a las clases hijas a definir los detalles específicos

- **Herencia**
  La clase ***PokemonBatalla*** extiende de *Pokemon*, heredando sus atributos pero agregando lógica de estados. Igualmente, ***Pocion*** hereda de *Item*, reutilizando la propiedad de cantidad y nombre

- **Polimorfismo: Método usar():** Gracias a la sobrescritura de métodos (*@overrride*), el método usar() se comporta distinto dependiendo del objeto. Si el objeto es una Pocion, el método restaura puntos de vida numéricos, si es un CuradorEstado, verifica y limpia una condición específica (como EstadoCondicion.envenenado). El sistema de batalla simplemente llama a item.usar(), sin necesitar saber qué tipo específico de ítem es, y esto nos permite demostrar de manera efectiva el concepto del polimorfismo

- **Encapsulamiento y Gestión (SistemaBatalla):** La lógica principal reside en la clase **Batalla** (archivo *SistemaBatalla2.dart*), donde se aplicó **Encapsulamiento** para proteger el estado del juego.

  - **Atributos Privados:** Variables como:

    `\_jugador, \_rival, \_itemsRivalActuales y \_enCurso`

    se declararon privadas (usando el guion bajo), garantizando que solo la clase **Batalla** pueda modificar el estado de la partida.
  - **Gestión de Turnos:** El método **determinarTurno** compara la velocidad de los objetos **PokemonBatalla**. Si un Pokémon está paralizado, su velocidad efectiva se reduce a la mitad para ese cálculo, alterando el orden de ataque *sin modificar el atributo base para el resto*

- **Rival y Manejo de Excepciones:** Para el comportamiento del rival (*ComportamientoRival.dart*), se implementó una toma de decisiones lógica apoyada en el Manejo de Excepciones:

  - Lógica de Prioridades: Se evalúa su estado en orden:
    1. **Curar Estado**: Si tiene un estado anormal (como Quemado)
    2. **Curar Vida**: Si su vida es menor al 50 %.
    3. **Atacar**

- **Uso de try-catch:** Para verificar si el rival tienw el item necesario en su mochila, se utiliza el método **firstWhere** dentro de un bloque try.
  Si el ítem no existe (lanzando una excepción de "elemento no encontrado"), el catch captura el error y permite que la continue a la siguiente prioridad, evitando que el juego se cierre y ya.

- **Instancias:**
  en el archivo **Instancias.dart**, creamos los objetos en los que se basa la pelea. Definimos instancias finales de **Ataque** (como lanzallamas, terremoto) y de **PokemonBatalla** (como charizard, pikachu), asignándoles sus tipos, vida y listas de ataques de cada uno. *Con esto separamos la definición de los datos de la lógica del negocio*

- **Manejo de hilos**
  Una aplicación tiene un solo hilo principal que es el encargado de dibujar la interfaz de usuario y responder a los inputs del usuario.
  Si ejecutamos una tarea muy larga como calcular la lógica completa de la batalla, que incluye el daño y efectos de estado, el hilo principal queda "bloqueado", esto significa que no puede dibujar la pantalla ni responder al usuario.

  - La asincronía resuelve lo anterior utilizando el concepto de **Future** y **Event Loop**:
    Una función que realiza una operación que lleva tiempo debe ser marcada como **async** y devolver un objeto **Future**.
    - **Future** es un objeto que produce un resultado (un valor o un error).
  - **await** es la instrucción clave que maneja la ejecución para que no se bloquee:
    1. Cuando el flujo de ejecución del código se encuentra con **await**, la función async que lo contiene se pausa inmediatamente en esa línea.
    2. En lugar de que el hilo principal se quede esperando, el control se cede al Event loop. El bucle de eventos ahora puede usar el hilo principal para realizar otras tareas pendientes.
    3. Cuando el **Future** (la lógica de la batalla que se ejecuta en el fondo) finaliza y produce un resultado, el Event loop coloca un evento "completado" en la cola de tareas. Cuando el hilo principal está libre, toma ese evento y continua con la función pausada justo en la línea siguiente al await.

4. Diagramas UML

4.1. Diagrama Estático

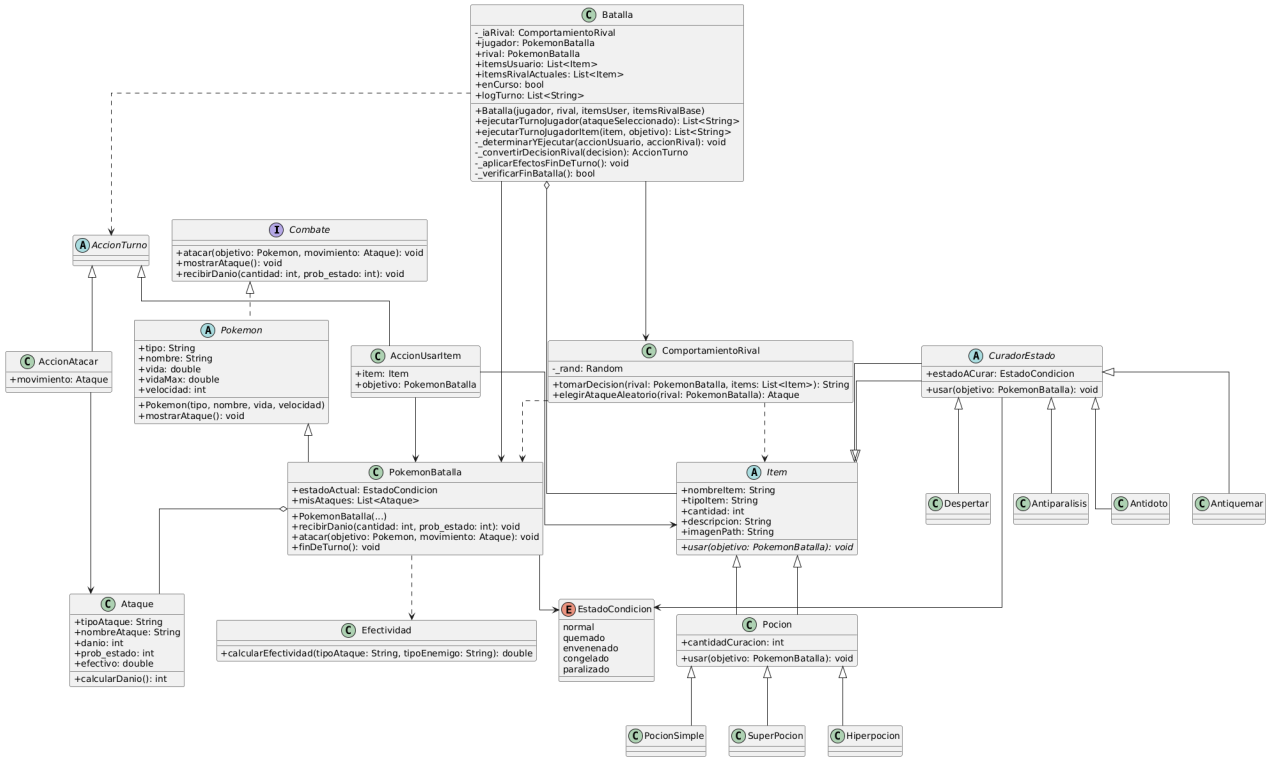

Figura 1: Diagrama de clases

Este diagrama nos permite ilustrar los elementos que contienen las clases y cómo usan la herencia de elementos así como las llamadas de estos.

Nota: Para la realización de este diagrama usamos la Herramienta PlantUML. Significado de cada letra en el Diagrama:

- ***C**: Clase*
- ***A**: Clase Abstracta*
- ***I**: Interfaz*
- ***E**: Enum*

4.2. Diagrama Dinámico

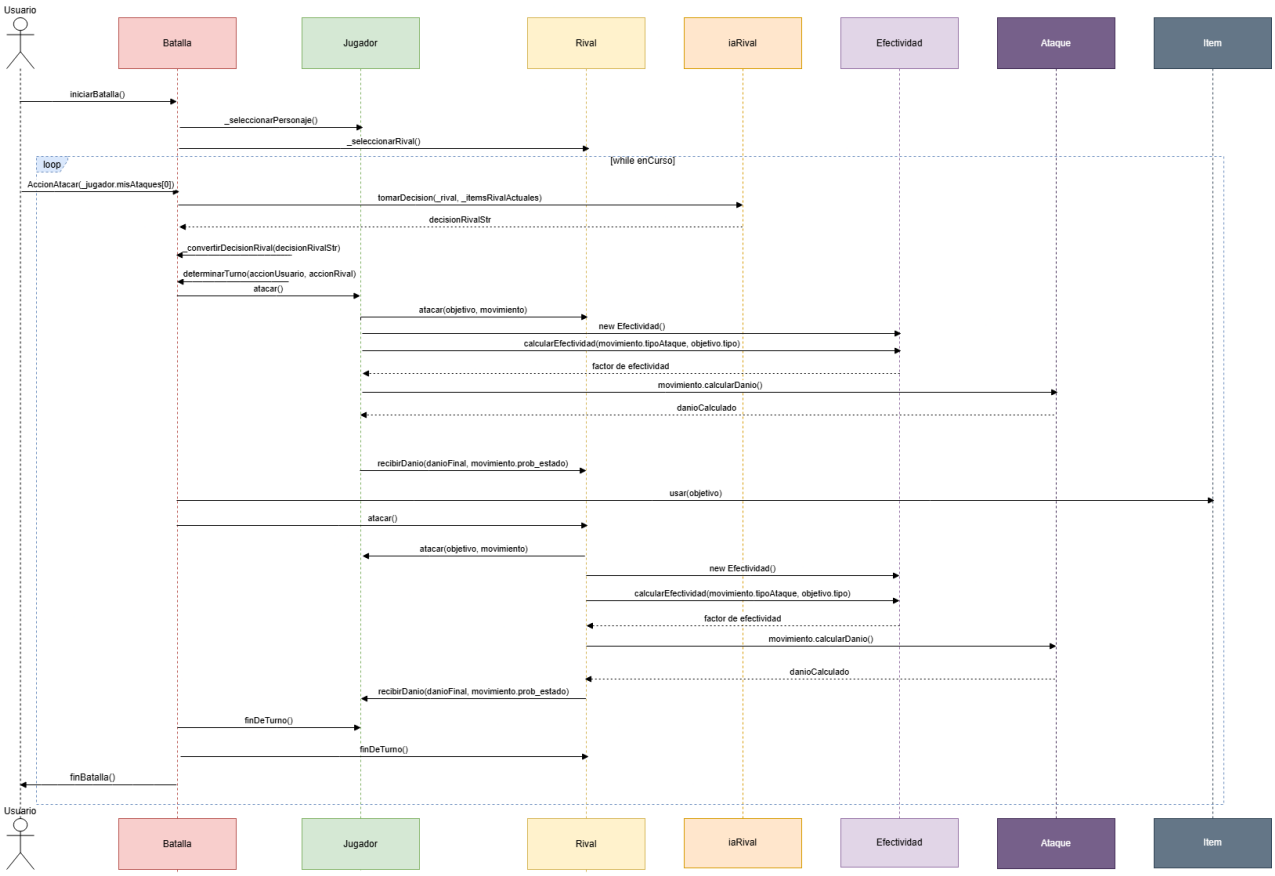

Figura 2: Diagrama de secuencia

5. Resultados

Tipo	Pokémon	Ataque 1 (Daño)	Ataque 2 (Daño)	Ataque 3 (Daño)
Steel	Lairon	Cola Férrea (100)	Garra Metal (50)	Cuerpo Pesado (80)
	Aggron	Cuerpo Pesado (80)	Cola Férrea (100)	-
	Aron	Cabeza de Hierro (80)	Garra Metal (50)	-
Dark	Zorua	Finta (60)	Pulso Umbrío (80)	Juego Sucio (95)
	Zweilous	Mordisco (60)	Pulso Umbrío (80)	Tajo Umbrío (70)
	Zarude	Mordisco (60)	Pulso Umbrío (80)	Tajo Umbrío (70)
Dragon	Flygon	Cola Dragón (80)	Garra Dragón (80)	-
	Rayquaza	Cola Dragón (80)	Enfado (120)	Vasto Impacto (80)
	Dragapult	Vasto Impacto (80)	Enfado (120)	Cola Dragón (80)
Ghost	Gengar	Bola Sombra (80)	Púas Tóxicas (20)	-
	Aegislash	Tajo Umbrío (70)	Sombra Vil (90)	Escudo Real (10)
	Chandelure	Fuego Fatuo (50)	Infortunio (65)	Rayo Confuso (40)
Rock	Tyranitar	Roca Afilada (100)	Triturar (80)	Danza Dragón (20)
	Golem	Trampa Rocas (15)	Explosión (150)	Avalancha (75)
	Rhyperior	Romperrocas (40)	Cuerno Taladro (50)	Cabezazo Roca (150)
Bug	Scizor	Ida y Vuelta (70)	Puño Bala (40)	Danza Espada (20)
	Volcarona	Zumbido (90)	Danza Aleteo (20)	Vendaval (110)
	Heracross	Tijera X (80)	Megacuerno (120)	Bocajarro (110)
Psycho	Alakazam	Psíquico (90)	Confusión (50)	Psico Corte (70)
	Mewtwo	Onda Mental (90)	Confusión (50)	Psico Corte (70)
	Baltoy	Confusión (50)	Psicorrayo (65)	Paranormal (65)
Flying	Staraptor	Tornado (40)	Ataque Ala (60)	Pájaro Osado (120)
	Yveltal	Ala Mortífera (80)	Vendaval (110)	Tornado (40)
	Pidgeot	Aire Afilado (75)	Pájaro Osado (120)	Danza Pluma (20)
Ground	Diglett	Bofetón Lodo (55)	Disparo Lodo (55)	Bomba Fango (65)
	Golem (T)	Terremoto (100)	Disparo Lodo (55)	Bofetón Lodo (55)
	Sandslash	Arenas Ardientes(70)	Ataque Arena (20)	Terratemblor (60)
Poison	Ekans	Picotazo Ven. (45)	Deslumbrar (30)	Ácido (40)
	Nidorina	Doble Patada (70)	Picotazo Ven. (45)	Mordisco (60)
	Seviper	Cola Veneno (50)	Niebla (15)	Colmillo Ven. (90)
Fight	Mankey	Patada Baja (60)	Arañazo (40)	Golpes Furia (50)
	Machop	Golpe Karate (50)	Mov. Sísmico (80)	Sumisión (80)
	Hitmonlee	Doble Patada (70)	Patada Salto (100)	Patada Giro (60)
Ice	Jynx	Puño Hielo (75)	Beso Amoroso (15)	Doble Bofetón (65)
	Lapras	Rayo Hielo (90)	Canto (20)	Neblina (40)
	Dewgong	Rayo Aurora (65)	Descanso (20)	Derribo (90)
Grass	Bulbasaur	Látigo Cepa (45)	Drenadoras (20)	Hoja Afilada (55)
	Oddish	Absorber (20)	Polvo Veneno (10)	Danza Pétalo (120)
	Tangela	Constricción (15)	Paralizador (10)	Latigazo (80)
Electric	Pikachu	Impactrueno (40)	Voltio Cruel (90)	Chispazo (80)
	Magnemite	Impactrueno (40)	Chispazo (80)	Rayo (90)
	Jolteon	Voltiocambio (70)	Impactrueno (40)	Electrocañón (120)
Water	Squirtle	Burbuja (40)	Acua Cola (90)	Hidropulso (60)
	Poliwhirl	Burbuja (40)	Rayo Burbuja (65)	Escaldar (80)
	Vaporeon	Acua Cola (90)	Hidropulso (60)	Escaldar (80)
Fire	Charmander	Ascuas (40)	Lanzallamas (90)	Nitrocarga (50)
	Arcanine	Lanzallamas (90)	Llamarada (110)	Colmillo Ígneo(65)
	Rapidash	Llamarada (110)	Nitrocarga (50)	Calcinación (60)
Normal	Rattata	Ataque Rápido (40)	Placaje (40)	Hipercolmillo(80)
	Meowth	Arañazo (40)	Golpe Cuerpo (85)	Frustración (60)
	Eevee	Ataque Rápido (40)	Placaje (40)	Golpe Cuerpo (85)

Figura 3: Tabla de daños Basada en la Figura proporcionada en las instrucciones

5.1. Prueba de Ejecución

6. Conclusiones

Se concluyó la práctica dando cumplimiento a los objetivos planteados, logrando desarrollar un sistema de batallas de Pokemon, siendo aplicando los conocimientos adquiridos en el curso de programación orientada a objetos. Se implementó un sistema de turnos que controla la batalla partiendo de elementos basicos como la definición de un Pokemon o Items con sus respectivas propiedades y estadísticas que permitieron darle dinamismo a la mecánica del juego. En el codigo se aplicaron los conceptos de abstracción, herencia, polimorfismo y encapsulamiento para definir la estructura del sistema y organizar su comportamiento. Permitiendo crear clases base, extender sus funcionalidades y sobrescribir métodos. Igualmente pudimos aplicar los conceptos del Manejo de Excepciones, Hilos y patrones, pudimos noter que estas implementaciones se vieron más reflejadas en la parte de la interfaz gráfica. La práctica permitió comprender de manera más clara cómo se aplican estos conceptos en un proyecto real, mostrando la estructura y desarrollo de proyectos de este tipo, lo que reforzó los conceptos vistos en el curso de programación orientada a objetos, demostrando asi el potencial de alcance que puede tener el correcto uso de estos.

7. Referencias

[1] The Dart Project Authors. Asynchrony support. dart.dev. (https://dart.dev/language/async), 2025. Consultado el 29-11-2025.

[2] The Dart Project Authors. Concurrency in dart. dart.dev. (https://dart.dev/language/concurrency), 2025. Consultado el 29-11-2025.

[3] Bailón J. Avila, J. Clases y objetos. en análisis y diseño en poo. (https://portalacademico.cch.unam.mx/cibernetica1/analisis-y-diseno-en-poo/clases-y-objetos), 2022. Consultado el 20-11-20255.

[4] J. Avila, J. y Bailón. Diagramas uml. en análisis y diseño en poo. (https://portalacademico.cch.unam.mx/cibernetica1/analisis-y-diseno-en-poo/diagramas-UML), 2022. Consultado el 20-11-2025.

[5] J.L. Blasco. Introducción a poo en java: Herencia y polimorfismo. (https://openwebinars.net/blog/introduccion-a-poo-en-java-herencia-y-polimorfismo/), 2023. Consultado el 20-11-2025.

[6] Alianza BUNAM. Conceptos básicos de programación orientada a objetos. (https://alianza.bunam.unam.mx/cch/conceptos-basicos-de-programacion-orientada-a-objetos/), 2022. Consultado el 20-11-2025.

[7] Google Developers. Adding assets and images. flutter.dev. (https://docs.flutter.dev/ui/assets/assets-and-images), 2025. Consultado el 29-11-2025.

[8] The Dart Project Authors (Google). Dart language documentation: Error handling. (https://dart.dev/language/error-handling), 2024. Consultado el 20-11-2025.

[9] The Dart Project Authors (Google). Enumerated types. (https://dart.dev/language/enums), 2025. Consultado el 20-11-2025.

[10] The Dart Project Authors (Google). List<e>class. (https://api.dart.dev/dart-core/List-class.html), 2025. Consultado el 20-11-2025.

[11] The Dart Project Authors (Google). Map<k, v>class. (https://api.dart.dev/dart-core/Map-class.html), 2025. Consultado el 20-11-2025.

[12] The Dart Project Authors (Google). The pubspec file. dart.dev. (https://dart.dev/tools/pub/pubspec), 2025. Consultado el 20-11-2025.

[13] The Dart Project Authors (Google). Sound null safety. (https://dart.dev/null-safety), 2025. Consultado el 20-11-2025.

[14] J. A. Lozano. Excepciones y errores. (https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3085/mod$_r esource/content/1/UAPA-Excepciones-Errores/index.html), 2020. $Consultadoel 20-11-2025$.

[15] Nascor. Introducción a los métodos en java: Tipos y funcionamiento. (https://cursosnascor.com/blog-detalle/introduccion-los-metodos-en-java-tipos-y-funcionamiento), 2023. Consultado el 20-11-2025.

[16] Nivardo. Sobrescritura de métodos en java. (https://oregoom.com/java/sobrescritura-de-metodos/), 2024. Consultado el 20-11-2025.

[17] Refactoring.Guru. Patrón command. (https://refactoring.guru/es/design-patterns/command), 2025. Consultado el 29-11-2025.

[18] Refactoring.Guru. Patrón observer. (https://refactoring.guru/es/design-patterns/observer), 2025. Consultado el 29-11-2025.

[19] Refactoring.Guru. Patrón state. (https://refactoring.guru/es/design-patterns/state), 2025. Consultado el 29-11-2025.

[20] J. A. Solano. Clases y objetos. (https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3068/mod$_r esource/content/1/UAPA-Clases-Objetos/index.html), 2020. $Consultadoel 20-11-2025$.