



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorios de docencia

# Laboratorio de Computación Salas A y B

*Profesor(a):* \_\_\_\_\_ M.C. RENE ADRIAN DAVILA PEREZ

*Asignatura:* \_\_\_\_\_ PROGRAMACIÓN ORIENTADA A OBJETOS

*Grupo:* \_\_\_\_\_ 01

*No de Proyecto:* \_\_\_\_\_ 03

*Integrante(s):* \_\_\_\_\_ 322118311

\_\_\_\_\_ 322094028

\_\_\_\_\_ 322092842

\_\_\_\_\_ 322078673

\_\_\_\_\_ 322067738

*No. de lista o  
brigada:* \_\_\_\_\_ 03

*Semestre:* \_\_\_\_\_ 2026-1

*Fecha de entrega:* \_\_\_\_\_ 01/12/2025

*Observaciones:* \_\_\_\_\_

**CALIFICACIÓN:** \_\_\_\_\_

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Marco teorico:</b>	<b>2</b>
<b>3. Desarrollo</b>	<b>4</b>
3.1. Diagrama Estatico. . . . .	10
3.2. Diagrama de Secuencia. . . . .	11
<b>4. Resultados</b>	<b>12</b>
<b>5. Conclusiones</b>	<b>13</b>
<b>6. Bibliografía</b>	<b>14</b>

# 1. Introducción

- **Planteamiento del problema.** El reto de este proyecto consiste en realizar un juego al estilo clásico de pokémon, con sus mecánicas establecidas las cuales son atacar por turnos, observar tu vida y la del rival, por otro lado tambien la variedad de personajes así como sus habilidades,etc ,la cuestión de aquí, es que será realizado en un lenguaje de programación diferente llamado flutter, la dificultad aquí sera reflejada principalmente por la falta de experiencia del lenguaje.
- **Motivación.** La aplicación práctica de la POO para modelar un juego clásico como Pokémon, usando herencia y polimorfismo en cosas como personajes y ataques. Es un desafío práctico atractivo que demuestre nuestras habilidades de desarrollo y creatividad que mejore nuestra capacidad de trabajar en un proyecto en equipo que nos apasiona.
- **Objetivos.**

1. Definir las características principales de cada Pokémon, como los puntos de vida (HP) y la velocidad.
2. Implementar un sistema de batallas por turnos basado en la velocidad: el Pokémon con mayor velocidad realiza el primer ataque. Despues, el segundo Pokémon responde para completar el turno. La batalla finaliza cuando alguno de los dos llega a cero puntos de vida.
3. Definir los métodos necesarios para mostrar un menú similar al presentado en la figura de referencia.
4. Crear al menos un Pokémon de cada tipo y al menos un movimiento por tipo, siguiendo las relaciones de daño mostradas en la Figura 1. Los círculos verdes indican daño doble, los cuadros amarillos indican daño reducido y las X rojas representan inmunidad.
5. Mostrar en pantalla al Pokémon rival y al Pokémon controlado por el usuario.
6. Incluir en el menú la opción ATACAR, la cual debe desplegar una lista de movimientos disponibles, tal como se muestra en la Figura 2.

## 2. Marco teórico:

### Objetivo 1

Para este objetivo se emplearon los siguientes conceptos de Programación Orientada a Objetos:

- Clase: Describe a todos los objetos de un tipo. [1]
- Objeto: Encapsulación genérica de datos y de los procedimientos para manipularlos. [1]

- Atributos: Aquellas que dan identidad a las variables.
  - Encapsulamiento: Es una especie de capa protectora con el cual brinda un nivel de seguridad, para acceder a los objetos encapsulados se hacen por medio de gets(metodos consultores) y setters(metodos modificadores. [1]
  - Métodos: Especifican el comportamiento de la clase y sus objetos. [1]
  - Constructores: Es un método que tiene el mismo nombre que la clase cuyo propósito es inicializar los atributos de un nuevo objeto. Se ejecuta automáticamente cuando se crea un objeto de una clase. Siempre hay cuando menos un constructor. [1]
- Si no se ha escrito un constructor en la clase, el compilador proporciona un constructor por defecto, el cual no tiene parámetros e inicializa los atributos a su valor por defecto.
- Polimorfismo: • Capacidad de un objeto de decidir qué método aplicar, dependiendo de la clase a la que pertenece – Una llamada a un método sobre una referencia de un tipo genérico (clase base) ejecuta la implementación correspondiente del método dependiendo de la clase del objeto que se creó • Permite diseñar e implementar sistemas extensibles – Los programas pueden procesar objetos genéricos (descritos por referencias de la superclase) – El comportamiento concreto depende de las subclases [1]

## Objetivo 2

- Objeto
- Abstracción: La abstracción en la programación orientada a objetos (POO) es un concepto que se utiliza para reducir la complejidad del código y mejorar la diseño de software al ocultar detalles específicos de implementación. La abstracción ayuda a los desarrolladores a enfocarse en los conceptos clave y en la funcionalidad de alto nivel de una aplicación sin distraerse con los detalles de bajo nivel del código. En POO, la abstracción se implementa comúnmente a través de clases abstractas e interfaces, lo que permite definir un conjunto de comportamientos o propiedades que una clase derivada debe implementar. [1]

## Objetivo 4

Conceptos aplicados:

- Archivos: Las aplicaciones puedan comunicarse con su entorno, para obtener datos e información que se busca procesar o almacenar. dar un resultado. o bien devolver el resultado, volverlo persistente.

El manejo de archivos se realiza a través de flujos de datos desde una fuente hacia un repositorio. La fuente inicia el flujo de datos, a lo que llamamos flujo

de entrada. El repositorio termina el flujo de datos, a lo que llamamos el flujo de salida. En otras palabras, tanto la fuente como el repositorio representan nodos de flujo de datos [2]

### 3. Desarrollo

#### 1. Preparación del proyecto

- a) Creación de un nuevo proyecto Flutter.

Para comenzar con el desarrollo de este proyecto, primero procedimos a crear un nuevo proyecto utilizando el entorno de Flutter. Abrimos la terminal y ejecutamos el comando *flutter create pokemonpro*, lo cual permitió que Flutter preparara automáticamente los directorios y archivos necesarios para comenzar a programar.

- b) Creación de las carpetas para organizar el código:

- */source*: Aquí colocaremos los archivos .dart que se encargarán de la lógica de nuestro videojuego.
- */views*: Aquí colocaremos los archivos .dart que corresponderán a la construcción de la interfaz gráfica del juego.
- */sprites*: Aquí colocaremos las imágenes .png para cada Pokémon.
- */audio*: Aquí estarán los archivos .mp3 (la música que se utilizará para cada pantalla).

En ese sentido, este paso también involucró el proceso creativo, el cual consistió en una lluvia de ideas entre todo el equipo y la recopilación de material que podríamos usar para este proyecto (música, imágenes de Pokémon, fondos, etc.).

#### 2. Clases base

- a) Crear la clase **Tablatipos** con sus listas de efectividades, resistencias e inmunidades.

Posteriormente procedimos a implementar la clase que modelará la tabla de daño. Para ello se creó el archivo **tablatipos.dart**, el cual contiene la clase **Tablatipos**. Esta permite determinar cómo interactúan los ataques de cierto tipo con los tipos del Pokémon defensor dentro del combate.

La clase contiene tres listas principales:

- **numeroefectivo**: contiene los identificadores de los tipos a los cuales se les causa un daño aumentado ( $\times 2$ ).
- **numeroresiste**: almacena los tipos que reducen el daño recibido ( $\times 0.5$ ).
- **numeroinmune**: agrupa los tipos completamente inmunes al daño ( $\times 0$ ).

Se creó el atributo `idtipo`, encargado de identificar el tipo correspondiente a cada instancia. A partir de esta estructura se definieron los 17 tipos de Pokémon como objetos estáticos dentro de la misma clase, cada uno asignado con sus relaciones específicas de efectividad, resistencia e inmunidad según la tabla de daños.

b) Implementar funciones para:

- obtener tipo por ID,
- calcular daño super efectivo,
- calcular daño resistido,
- identificar inmunidad.

Para complementar la estructura de los tipos definidos previamente, se implementaron diversas funciones auxiliares encargadas de gestionar la lógica de interacción entre los tipos de Pokémon durante una batalla. Estas funciones permiten consultar las relaciones de efectividad sin necesidad de acoplar la lógica directamente dentro de las clases de combate.

La función `obtenerTipoPorId()` devuelve la instancia correspondiente de `Tablatipos` en función del identificador numérico del tipo.

La función `multiplicadorSupereflectivoPor()` determina cuándo un movimiento resulta muy efectivo contra el Pokémon rival. Por otro lado, `multiplicadorResistidoPor()` calcula si el movimiento es resistido.

Finalmente, la función `inmune()` identifica los casos donde el movimiento no genera daño debido a inmunidades según la tabla de tipos.

c) Crear la clase `Pokemon` con atributos de vida, tipos, velocidad y estados:

Para representar a cada Pokémon, se desarrolló la clase `Pokemon`, la cual encapsula las características básicas necesarias para participar en una batalla: puntos de vida, tipos, velocidad, número en la Pokédex y estado alterado.

También se incluyeron los métodos `damage()`, encargado de reducir la vida, y `muerto()`, que determina si el Pokémon ha sido derrotado.

d) Implementar la clase `Movimiento` para representar ataques:

Finalmente, se implementó la clase `Movimiento`, la cual modela los ataques utilizados por los Pokémon durante una batalla. Cada movimiento cuenta con un identificador, un nombre, un tipo asociado y un valor de ataque que determina el daño base que puede causar.

### 3. Implementación de colecciones

a) Crear la lista de Pokémon (`pokedex`):

Una vez definidas las clases `Pokemon` y `Movimiento`, procedimos a crear una colección que contuviera a todos los Pokémon que formarían parte del proyecto. Para ello implementamos el archivo correspondiente donde declaramos la lista `pokedex`, la cual almacena instancias de la clase `Pokemon`.

- b) Implementar una función para obtener Pokémon por ID:

Para tener facil el acceso a los Pokémon almacenados en la `pokedex`, implementamos la función `obtenerPokemonPorId()`. Esta función recibe como parámetro el identificador del Pokémon y devuelve la instancia correspondiente.

- c) Crear los movimientos en la clase `Movimientos`:

Diseñamos una estructura para gestionar los movimientos disponibles en el juego. Para ello creamos la clase `Movimientos`, en la cual cada objeto estático representa un ataque con sus propiedades : nombre, tipo y poder.

- d) Función para obtener movimientos por ID:

Finalmente, agregamos la función `porId()` dentro de la clase `Movimientos`, la cual se encarga de devolver el movimiento correspondiente al identificador. Esta función permite seleccionar ataques específicos durante el combate.

#### 4. Implementación del sistema de combate:

- a) Crear la clase `Pelea`. Implementamos la clase `Pelea`, que se encargara de hacer los cálculos de daño entre dos Pokémon.

- b) Implementar el constructor que recibe los IDs de los dos Pokémon y los dos movimientos. El constructor de la clase `Pelea` recibira como parámetros los identificadores de los dos Pokémon y los dos movimientos seleccionados. Con estos valores, la clase obtiene las instancias correspondientes desde las colecciones mediante las funciones `obtenerPokemonPorId()` y `Movimientos.porId()`.

- c) Determinar quién ataca primero mediante la velocidad.

Para esto se compararon los atributos `velocidad` de ambos Pokémon. El Pokémon con mayor velocidad se asigna como atacante inicial (primer turno).

- d) Calcular el daño basándose en:

- poder del movimiento,
- tipo del ataque,
- tipos del Pokémon defensor,
- multiplicadores por efectividad,
- inmunidades.

Para el calculo de daño se incorporan los atributos del movimiento utilizado y los tipos del Pokémon defensor. En este procedimiento se utilizan las funciones previamente definidas: `multiplicadorSuperefactivoPor()`, `multiplicadorResistidoPor()` y `inmune()`. Dichas determinaran si el ataque resulta súper efectivo, poco efectivo o completamente ineficaz debido a inmunidades. En caso de inmunidad, el daño final se reduce a cero sin necesidad de cálculos adicionales.

e) Aplicar daño al Pokémon atacado.

Mediante el método `damage()` se aplica el daño calculado al Pokémon defensor. Esta función actualiza la vida del Pokémon, controlando que no disminuya por debajo de cero.

f) Implementación de efectos de estado (veneno, quemadura, parálisis).

Además del daño directo, algunos movimientos podrán causar estados en los pokémon. Para ello se evalúa si el movimiento posee un valor distinto de cero en el atributo `estadoAlterado`. En caso afirmativo, se genera una probabilidad para determinar si la condición se aplica al Pokémon rival.

g) Aplicar daño por estado al final del turno.

Finalmente, se implementó el comportamiento de daño residual provocado por estados como veneno y quemadura. Este daño se calcula como un porcentaje de la vida máxima del Pokémon afectado y se aplica al finalizar el turno. En el caso de la parálisis, se evalúa si el Pokémon pierde su turno debido a esta condición. Estos efectos agregan realismo al sistema y enriquecen la estrategia dentro del combate.

## 5. Colocar música de fondo

a) Crear la clase `AudioManager`

Para gestionar de forma centralizada todos los audios del proyecto, se desarrolló la clase `AudioManager`, implementada mediante el patrón de diseño *Singleton*. Este patrón garantiza que solo existe una única instancia responsable de controlar la música y los efectos de sonido durante toda la ejecución del juego.

b) Implementar música de fondo y efectos de sonido.

Dentro de la clase `AudioManager` se utilizaron las funciones proporcionadas por el paquete `audioplayers` para reproducir música de fondo y efectos cortos en distintos puntos del juego. Se configuró la música de cada pantalla para que se reproduzca automáticamente al acceder a ella, y se implementaron métodos específicos para reproducir efectos puntuales, como seleccionar un menú o abrir la mochila.(que serán utilizados en pasos más adelante).

c) Agregar control de loop, pausa, reanudación y volumen:

Finalmente, se añadieron controles que permiten ajustar el comportamiento de la música. Entre estos se encuentran la reproducción en bucle para la música del menú y la batalla, funciones para pausar o reanudar la música según el flujo del juego y un control de volumen adaptable.

## 6. Diseño de la interfaz

a) Crear la vista `MenuView`.

Para iniciar con el diseño de la interfaz del proyecto, se creó la vista `MenuView`, la cual funcionara como pantalla principal del juego. Esta interfaz fue desarrollada inspirándose en el estilo de los juegos de Pokémon para Game Boy Advance (GBA), utilizando fondos previamente creados en el proceso creativo, tipografía retro y colores saturados. Para mantener la estética retro del proyecto, se utilizó la tipografía *Press Start 2P*, obtenida mediante el paquete `googlefonts`.

b) Opciones LUCHA y MOCHILA.

Se incorporaron las dos opciones principales del menú: LUCHA y MOCHILA. Cada una de ellas actúa como un botón que permite al usuario acceder a diferentes secciones del juego. La opción LUCHA redirige a la pantalla de combate, mientras que la opción MOCHILA abre una interfaz donde el usuario puede visualizar su colección de Pokémon.

c) Agregar el selector animado: Se implementó un selector animado utilizando un carácter en forma de flecha cuya posición cambia dependiendo de la opción del menú seleccionada. Para lograrlo, empleamos el widget `AnimatedSwitcher`, el cual permite animar transiciones suaves cuando el valor de `selectedIndex` cambia. Cada vez que el usuario pasa el cursor sobre una opción o hace clic en ella, la variable `selectedIndex` se actualiza dentro de `setState()`, provocando que la flecha se mueva automáticamente hacia la posición correspondiente.

d) Crear la pantalla de la mochila:

La sección MOCHILA incluyó una vista que muestra una lista completa de los Pokémon disponibles. Cada entrada en la lista contiene el nombre del Pokémon y un número de identificación visual. Además, al seleccionar uno de ellos, se despliega una segunda vista donde se muestra una imagen ampliada del Pokémon, así como información asociada. (cabe recalcar que únicamente muestra una imagen con los datos del pokémon, la cual fue creada y editada previamente).

e) Implementar la navegación hacia la batalla:

Finalmente, se configuró la navegación desde la vista del `MenuView` hacia la pantalla de combate. Al seleccionar la opción LUCHA, se genera un cambio de pantalla (con un fondo creado previamente en el proceso creativo) utilizando el sistema de rutas de Flutter. Además, se añadió la lógica necesaria para activar la música correspondiente al entrar a una batalla y restaurarla al regresar al menú principal.

## 7. Diseño de la pantalla de combate

a) Crear la vista `BattleView`.

Para representar el escenario de batalla del juego, se creó la vista `BattleView`, la cual actúa como la interfaz principal durante un enfrentamiento. Esta pantalla incluye un fondo creado previamente y los elementos

necesarios para mostrar tanto a los Pokémon involucrados como las opciones de acción disponibles durante el turno. La vista fue implementada utilizando `Stack` para superponer capas visuales

- b) Posicionar los sprites del Pokémon aliado y enemigo. Dentro de la vista de combate se colocaron dos sprites: el del Pokémon enemigo en la parte superior derecha y el del Pokémon aliado en la parte inferior izquierda. Para lograr colocarlos correctamente se utilizamos widgets `Positioned`.
- c) Agregar botones de acción: ATACAR, CAMBIAR, MOCHILA, HUIDA.  
En la parte inferior de la pantalla se implementó un menú interactivo compuesto por cuatro botones: ATACAR, CAMBIAR, MOCHILA y HUIDA. Estos botones fueron organizados mediante un `GridView` de dos columnas.

## 8. Integración del sistema de combate con la UI

- a) Se implementó la conexión directa entre la interfaz gráfica y la lógica de combate. Al presionar la opción ATACAR dentro de la vista de batalla, la interfaz despliega una cuadrícula con los movimientos disponibles del Pokémon aliado. Al seleccionar uno, la vista invoca el método `_ejecutarAtaque()`, el cual crea una instancia de la clase `Pelea` enviando los identificadores del Pokémon aliado, del enemigo, y los movimientos seleccionados para ambos combatientes.
- b) Se desarrolló un sistema de turnos completo controlado por la interfaz, donde la vista ejecuta el flujo de la batalla mediante el método `_procesarTurnoBatalla()`. Este proceso incluye: mostrar mensajes, aplicar pausas temporales para simular secuencias de ataque, ejecutar la lógica interna de `Pelea`, actualizar estados alterados (veneno, quemadura, parálisis) y reproducir los mensajes de daño generado..
- c) Después de recibir el resultado de `Pelea`, la interfaz actualiza las barras de vida de ambos Pokémon empleando `AnimatedContainer`, lo que permite mostrar transiciones al disminuir los puntos de vida. Además, los valores internos de `vidaActual` en cada instancia de `Pokemon` se mantienen sincronizados con la representación gráfica mostrada al usuario.
- d) Se incluyó detección de condiciones de derrota, victoria o empate. La UI revisa si alguno de los Pokémon ha llegado a cero puntos de vida y, en ese caso, despliega un cuadro de diálogo final con el mensaje correspondiente. También se integró la evaluación de estados alterados persistentes.
- e) Finalmente, al concluir la batalla, la interfaz retorna automáticamente al menú principal mediante `Navigator.pop()`, restaurando la música original del menú con `AudioManager`.

### 3.1. Diagrama Estático.

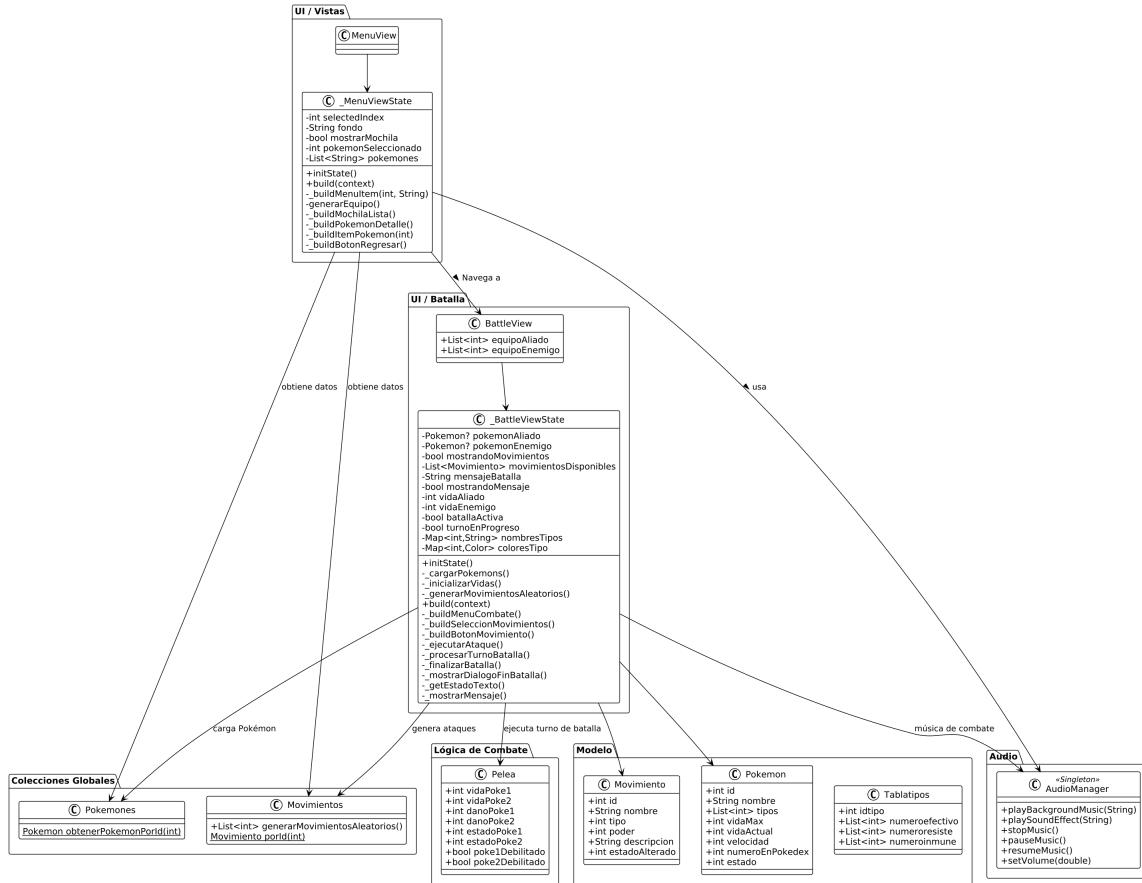


Figura 1: Diagrama de clases

### 3.2. Diagrama de Secuencia.

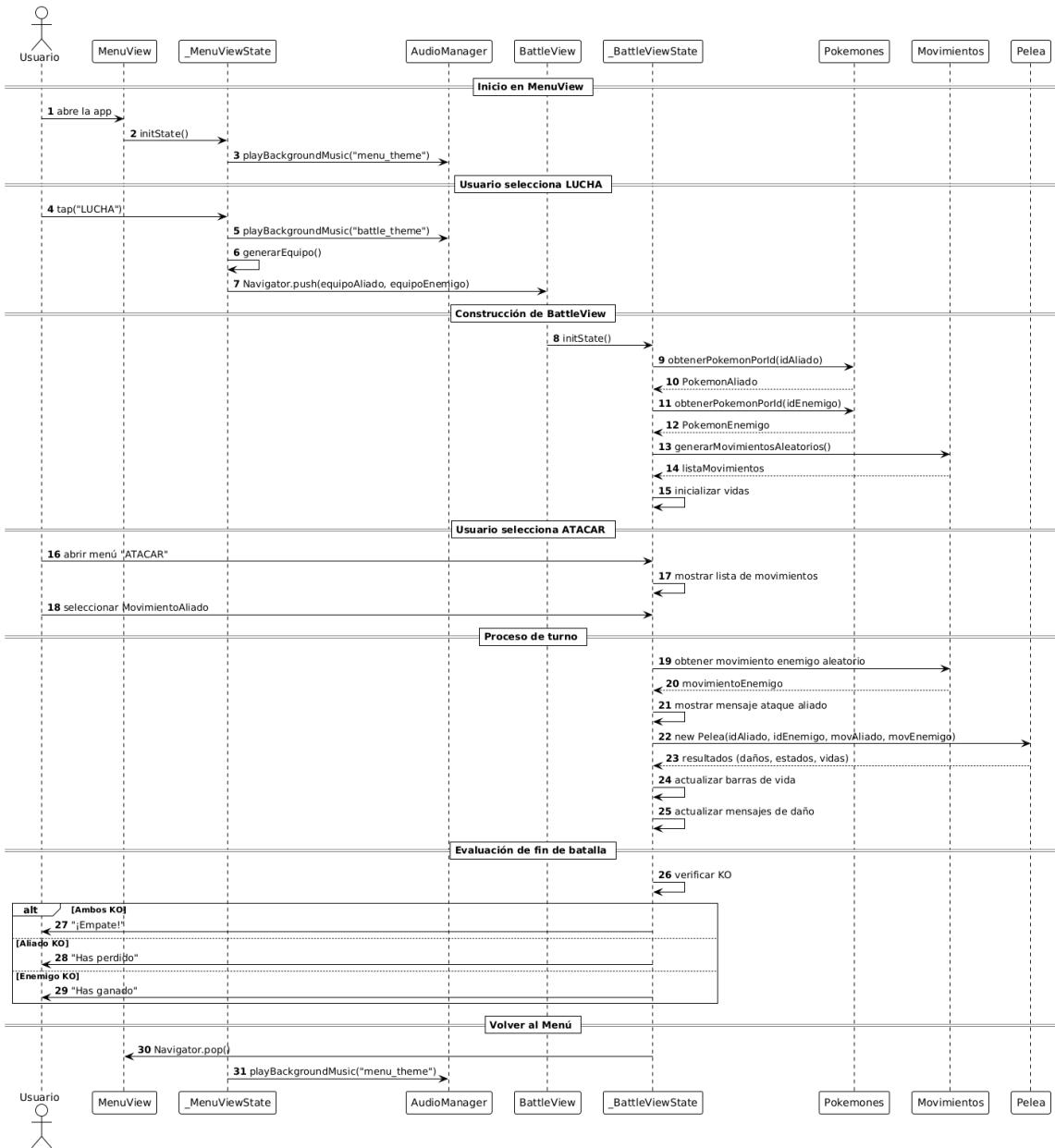


Figura 2: Enter Caption

## 4. Resultados



Figura 3: Menu principal con el requerimiento del objetivo 3.



Figura 4: Colección de pokemons al seleccionar la opción mochila.



Figura 5: Ejemplo de imagen que se muestra al seleccionar un Pokemon.

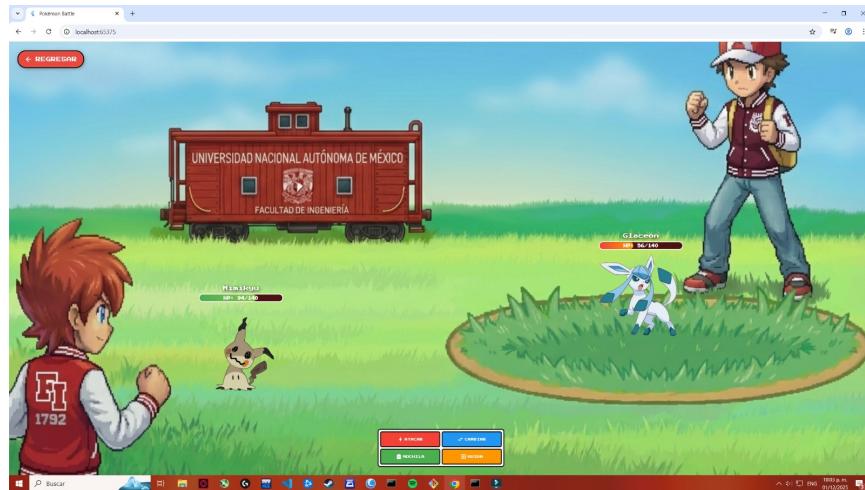


Figura 6: Pantalla de combate, con los pokemon ya colocados, cumpliendo el objetivo 5.

## 5. Conclusiones

Realizar este proyecto fue una experiencia que nos permitió poner en práctica todo lo aprendido sobre la materia de Programación Orientada a Objetos gracias a todos estos conocimientos que adquirimos en el semestre pudimos hacer un juego que simula a Pokémon en específico a sus características batallas por turnos, para poder realizarlo implementamos atributos para cada Pokémon como lo fue la vida, velocidad, distintos tipos de ataques, estados alterados y un sistema de turnos que dependía de las características de cada Pokémon Como este fue nuestro primer proyecto usando Flutter utilizamos los materiales que se nos compartió por en medio del Classroom lo cual no ayudo muchísimo a despejar dudas y hacer que el proceso no se sintiera tan pesado. Los diagramas UML nos permitieron ordenar de mejor manera la estructura

del programa y tener una idea más clara de lo que queríamos lograr antes de ejecutarlo. Trabajar por etapas y dividir tareas dentro del equipo ayudó a que el proyecto avanzara de forma óptima y que todos pudiéramos aportar. En general, este trabajo no solo nos permitió reforzar conceptos como herencia, polimorfismo, encapsulamiento y todo lo aprendido, sino que también entender la importancia de planear bien las ideas antes de empezar a programar.

## 6. Bibliografía

- [1] Notas de la asignatura de Programación Orientada a Objetos, Facultad de Ingeniería, Universidad Nacional Autónoma de México (UNAM). Material proporcionado en el curso, 2025.
- [2] R. A. Dávila, Programación Orientada a Objetos – Serie de videos sobre Archivos, Hilos e Isolates en Dart, Facultad de Ingeniería, Universidad Nacional Autónoma de México (UNAM). Material audiovisual proporcionado en el curso, 2025.
- [3] Google LLC, "Dart Language Tour,"[dart.dev](https://dart.dev/language), 2025. Accedido: Nov. 2025. Disponible en: <https://dart.dev/language>