



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): _____

Asignatura: _____

Grupo: _____

No de Práctica(s): Proyecto 3

Integrante(s): 322246320

322267567

322012051

322236688

322330872

*No. de lista o
brigada:* _____

Semestre _____

Fecha de entrega: _____

Observaciones: _____

CALIFICACIÓN: _____

Índice

1. Introducción	2
2. Marco Teórico	2
3. Desarrollo en serie de pasos	3
3.1. Planeación y organización:	3
3.2. Diseño:	3
3.2.1. Pokemones, ataques y efectos:	3
3.2.2. Manejo de archivos, música y diseño:	4
3.2.3. Interfaz gráfica y animaciones:	4
3.3. Implementación:	4
4. Desarrollo	4
4.1. Controlador	4
4.1.1. combate.dart	4
4.1.2. main.dart	6
4.2. Vista	6
4.2.1. finDeCombate.dart	6
4.2.2. interfazDeCombate.dart	6
4.2.3. menuPrincipal.dart	6
4.2.4. seleccionarPokemon.dart	6
4.2.5. tablaAtaques.dart	7
4.3. Modelo	7
4.3.1. ataque.dart	7
4.3.2. archivos.dart	7
4.3.3. ataques.dart	7
4.3.4. item.dart	7
4.3.5. pociones.dart	8
4.3.6. pokemon.dart	8
4.3.7. pokemones.dart	8
4.3.8. musica.dart	9
5. Aplicación de los objetivos	9
6. Diagramas UML	10
6.1. Diagrama estático:	10
6.2. Diagramas de secuencia:	11
7. Resultados relevantes	13
8. Conclusiones	16
9. Referencias	16

1. Introducción

- **Planteamiento del Problema:** Se busca diseñar un programa que simula el sistema de combate de los juegos pokémon, integrando parte de la lógica interna del juego original junto con una interfaz gráfica animada acorde al sistema de combates y que permita una correcta interacción con el usuario.
- **Motivación:** En este proyecto aplicaremos todos los temas vistos en el curso para la creación de una aplicación completa en Flutter. Reforzando los temas fundamentales de la programación Orientada a Objetos. Así como la creación de una interfaz gráfica funcional y completa.
- **Objetivos:** Crear un programa en Flutter que haga un uso eficiente del encapsulamiento, archivos, clases, polimorfismo e interfaz gráfica para diseñar una aplicación capaz de simular los combates del juego de Pokémon siguiendo parte de la lógica del juego original.

2. Marco Teórico

- **Patrón de diseño Singleton:** El propósito de este patrón es evitar que sea creado más de un objeto por clase. Esto se logra creando el objeto deseado en una clase y recuperándolo como una instancia estática.

Si se utiliza el patrón singleton para crear una instancia de una clase, entonces el patrón se asegura de que realmente sólo permanezca con esta instancia única. En los diferentes lenguajes de programación, hay diferentes métodos para lograrlo, pero principalmente se debe impedir que los usuarios creen nuevas instancias. Esto se logra mediante el constructor, declarando el patrón como 'privado'. Esto significa que sólo el código en el singleton puede instanciar el singleton en sí mismo, garantizando que sólo un mismo objeto puede llegar al usuario. [1]

- **Patrón de diseño MVC:** También llamado patrón modelo-vista-controlador, es un patrón de arquitectura de software en el cual se utilizan tres componentes, separando la lógica de la aplicación de la lógica de la vista, algo muy útil ya que permite realizar cambios en parte de la aplicación sin afectar a las demás, encargándose el modelo de cuestiones como manejo de datos, generalmente, aunque no obligatoriamente, consultando bases de datos para actualizaciones, consultas, búsquedas, etc.

El controlador es quien se encarga de procesar las solicitudes del usuario, pedirle los datos correspondientes al modelo y de comunicárselos a vista; vista se encarga de la representación visual de los datos, encargándose de toda la interfaz gráfica, no preocupándose el modelo ni el controlador por lo que sucede en esta parte. [2]

- **Sistema de combate Pokémon:** El sistema de combate de la franquicia Pokémon se basa en enfrentamientos por turnos entre pokemones que poseen

atributos numéricos y elementales. Cada Pokémon cuenta con un máximo de 4 movimientos, los cuales pueden causar daño directo, aplicar estados, o modificar estadísticas durante combate. La resolución de cada turno depende de la velocidad de los Pokémon.

Tabla de tipos: El sistema se basa en una tabla de efectividades entre tipos, que determina si un ataque es 'Superefactivo', 'Poco Efectivo' o 'Sin efecto'. Esto introduce un componente estratégico en cada combate.

A partir de estas reglas relativamente sencillas, el sistema puede adaptarse de una manera natural a un diseño de programación orientada a objetos. [3]

3. Desarrollo en serie de pasos

En esta sección describiremos el proceso para construir el juego de Pokémon, desde el planteamiento inicial de la idea, hasta el resultado final y las implementaciones.

3.1. Planeación y organización:

Empezamos analizando la manera en la que implementaríamos el juego, considerando la interfaz que más nos agradaba, los colores, música, animaciones, las ventanas que tendría, los ataques, efectos y los pokemones.

Dividimos los archivos `.dart` para facilitar el trabajo colaborativo, así como para mantener ordenado el código y que cada miembro del equipo pudiera trabajar sin generar problemas.

3.2. Diseño:

Realizamos la división de tareas para abordar todos los objetivos de la implementación entre todo el equipo.

3.2.1. Pokemones, ataques y efectos:

Dos de nosotros se encargaron de realizar toda la implementación de los pokemones, abordando los temas:

- Un pokemon de cada tipo.
- ataques personalizados.
- efectos al realizar un ataque (envenenamiento, quemadura, sueño, parálisis, congelamiento).
- pociones para curación y anti-efectos.
- efectividad de los movimientos sobre los pokemones según la tabla de tipos.
- prioridad de turno según la velocidad del pokemon y la prioridad del ataque.
- nivel de los pokemones al ganar combates.

3.2.2. Manejo de archivos, música y diseño:

Otro de nosotros se encargo de manejar los módulos esenciales del programa:

- manejo de archivos para almacenar los niveles.
- módulos de música para cada fase del juego, utilizando el patrón de *singleton*.
- manejo de turnos y ataques dentro del combate, utilizando el patrón *MVC*

3.2.3. Interfaz gráfica y animaciones:

Los últimos dos se encargaron de la interfaz gráfica:

- creación de varias ventanas según la fase de juego:
 - menú principal.
 - selección de pokémon.
 - tabla de ataques.
 - interfaz de combate.
 - ventana final de batalla.
- manejo de gifs, para simular animaciones dentro del juego.
- implementación de botones interactivos.

3.3. Implementación:

Esta división de tareas la pudimos juntar gracias al patrón de diseño **MVC**, que nos permitió juntar la interfaz gráfica con toda la lógica detrás de los pokémones. Organizar y manejar las interacciones del usuario. Y lo más importante de todo, mantener la modularidad del código.

4. Desarrollo

Dentro del código que implementamos, se trato de seguir una estructura de patrón MVC, haciendo distintos archivos para el modelo, el controlador y la vista; y encargándose cada archivo de cierta parte diferente del programa.

4.1. Controlador

4.1.1. combate.dart

En **combate.dart** implementamos la clase CombateController que controlará la lógica que ejecutan los pokémones durante la batalla y modificará solo mediante el llamado de métodos de otro *.dart* la información mostrada en la interfaz gráfica. Los métodos que contiene esta clase son:

- **iniciarCombate()** Recibe la información de los pokémon que pelearán.
- **intentarAplicarEstadoPorAtaque()** Este método se encarga, en caso de ser necesario y de caer en la probabilidad, de aplicar un efecto de estado a un pokémon.
- **actualizarEstadosPokemon()** En este método se va evaluando cuánto tiempo lleva el pokémon en ciertos estados para después de cierta cantidad de turnos quitar el estado.
- **aplicarDanioEstados()** Aquí se evalúa si el pokémon se encuentra bajo un efecto de estado como quemadura o envenenamiento, para calcular una cantidad de daño y aplicarsela antes de avanzar al siguiente turno.
- **curarPorAtaque()** Este método evalúa si se utilizó un ataque que cura al pokémon que lo usa, curándose un porcentaje del daño realizado, pero evitando que se sobrepase la vida máxima.
- **modificarVelocidadPorAtaque()** Aquí se evalúa si el ataque utilizado modifica de algún modo la velocidad ya sea del enemigo o del propio pokémon, para modificarlas en caso de ser necesario según un porcentaje de las velocidades base.
- **puedeAtacar()** Este método evalúa si el pokémon se encuentra bajo algún efecto de estado como sueño o congelamiento para devolver un booleano que se encarga de indicar si puede atacar o no.
- **verificarParalisis()** En este método se verifica si el pokémon está paralizado, para calcular la probabilidad de que no se mueva en el turno, en caso de ser necesario.
- **atacarJugador()** En este método se evalúa el ataque seleccionado para el combate para decidir quién ataca primero según velocidad o prioridad del ataque; verificando al final si los pokémon tienen vida para continuar el combate.
- **ejecutarAtaque()** En esta función es donde se aplica el daño que realiza cada ataque al pokémon que lo recibe y donde se mandan a llamar los métodos de aplicar estados.
- **usarObjeto()** Aquí se utilizan objetos de la clase *item* para eliminar efectos de estado o curarse.
- **turnoEnemigo()** Aquí se escoge un movimiento al azar del pokémon enemigo para utilizarlo en combate.
- **finalizarCombate()** Aquí se evalúa que pokémon tiene vida igual a cero, para mandar a un cambio de ventana con el dato de si se perdió o ganó el combate.

- **calcularDanio()** Aquí se aplica un multiplicador de daño a la potencia base del ataque a utilizar.
- **multiplicadorPorTipo()** Aquí se evalúa el tipo de ataque que recibirá el pokemon y el tipo del pokemon, para calcular un multiplicador según la relación entre los tipos.

4.1.2. main.dart

En **main.dart** se establece el título y las dimensiones de la ventana del programa, principalmente es la función que corre el programa.

4.2. Vista

4.2.1. finDeCombate.dart

Lo que hace **finDeCombate** es crear y mostrar un widget que indica si el jugador a perdido o ganado, en la parte superior derecha hay dos botones: 'Menú principal' y 'Volver a jugar'. En el centro se muestra la tarjeta del pokemon del jugador, con su nombre, tipo y nivel.

4.2.2. interfazDeCombate.dart

En **interfazDeCombate** se muestra el widget principal que muestra a ambos pokemones junto con sus datos de vida, nivel y estados; así como los botones mochila u ataques, que mostraran los ataques u objetos con los que se cuentan, para según lo seleccionado, mandar a llamar al método correspondiente de *combate.dart*

4.2.3. menuPrincipal.dart

Para nuestro **menuPrincipal.dart** se define el widget principal de la pantalla principal, en este se mostrará un fondo dinámico, una imagen con el título y tres botones los cuales son: 'Jugar', 'Música' y 'Comida'. El primero lleva a la pantalla de selección de pokemon, el segundo apaga o enciende la música y el ultimo botón termina la ejecución.

4.2.4. seleccionarPokemon.dart

En esta pantalla inicializamos todos los pokemones y se añade el seleccionado a una lista para que no se repita. Después definimos un widget llamado 'tarjetaPokemon' que es donde se muestra la información de cada pokemon a elegir. En la parte superior derecha tenemos el botón 'Ataques' que muestra los ataques registrados, su daño y los efectos que pueden causar determinados ataques. Cuando se termina la selección de pokemon, junto al botón 'Ataques' tenemos al botón 'Continuar' que nos lleva a la ventana de Combate.

4.2.5. tablaAtaques.dart

Aquí se recupera la información de todos los ataques creados, en una lista que se usará en la construcción del widget ventanaTablaAtaques, la cuál únicamente mostrará dichos datos.

4.3. Modelo

4.3.1. ataque.dart

Comenzamos creando la clase Ataque que contiene los atributos de:

- Nombre
- Tipo
- Potencia
- Prioridad
- Clase
- Estado secundario
- Probabilidad de estado

Después de inicializar el constructor con dichos atributos, se crearon las subclases de ataques para los 18 tipos de pokémon implementados en este proyecto.

4.3.2. archivos.dart

archivos.dart cuenta con tres métodos, encargándose en cada uno de algo distinto, uno intentar crear un archivo, en una ruta segura relacionada al lugar de memoria del juego, para simplemente escribir en distintas líneas el nivel de cada pokémon, otro se encarga de leer el archivo para almacenar en una lista, que se utiliza en otros **.dart** el nivel de cada pokémon, almacenando el nivel de los pokémones al ganar un combate, dentro de esa lista, para luego sobrescribir el archivo según lo que hay en la lista, asegurando que el nivel se guarde aún al cerrar el juego.

4.3.3. ataques.dart

En **ataques.dart** se crean 4 objetos de cada subclase de Ataque para cada tipo de pokémon, agregando los datos solicitados por el constructor, según información de los juegos originales.

4.3.4. item.dart

En **item.dart** creamos la clase abstracta Item que tiene los atributos nombre y descripción. Además de un método abstracto llamado usar(Pokémon pokémon).

4.3.5. pociones.dart

Aquí implementamos Item y creamos la clase **Potion** que curará 20 puntos de vida en el juego, tiene como atributo vidaMaxima, además de los extendidos.

También tenemos las clases: PotionQuitarQuem, PotionQuitarEnven, PotionQuitarCong, PotionQuitarPar y PotionDespertar, cada una la cuál se encargará deuitar los efectos de estado correspondientes.

4.3.6. pokemon.dart

Para la creación de **pokemon.dart** importamos la clase ataque.dart y después definimos la clase pokemon con los atributos de:

- Nombre
- Nivel
- Tipo
- Vida
- Velocidad
- Estado quemado (booleano)
- Estado congelado (booleano)
- Estado paralizado (booleano)
- Estado envenenado (booleano)
- Estado de sueño (booleano)
- Numero de turnos que duran ciertos estados
- Estado quemado (booleano)
- Ruta para Asset (Gif del pokemon de frente)
- Ruta para Asset (Gif del pokemon de espaldas)
- Lista de ataques
- Variable random (para aleatorizar el nivel del rival y su velocidad)

Dichos atributos son pedidos por el constructor. Además se agregaron los métodos: aplicarQuemadura(), aplicarCongelacion(), aplicarParalisis(), aplicarEnvenenamiento(), aplicarSuenio() y generarNivelRival(int nivelBaseJugador).

4.3.7. pokemones.dart

En **pokemones.dart** inicializamos un pokemon de cata tipo usando los atributos que nos pide el constructor.

4.3.8. musica.dart

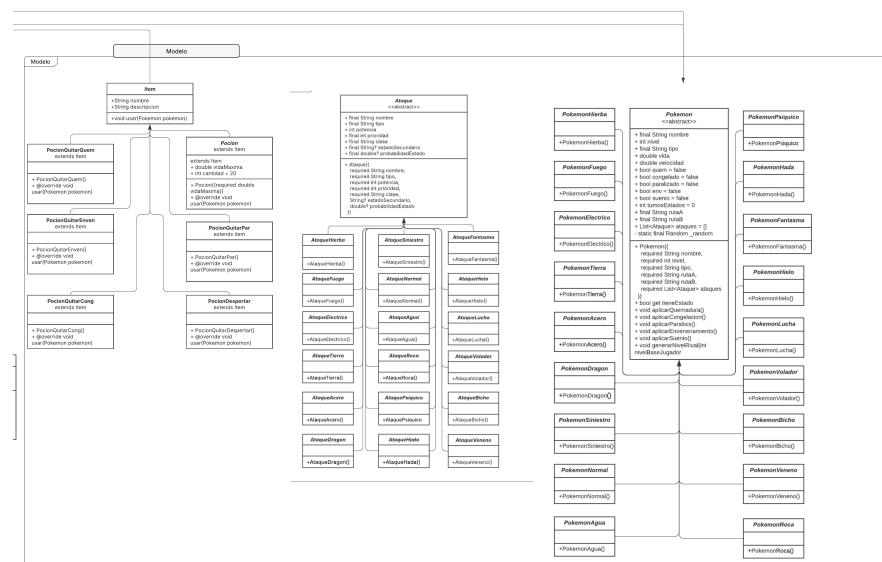
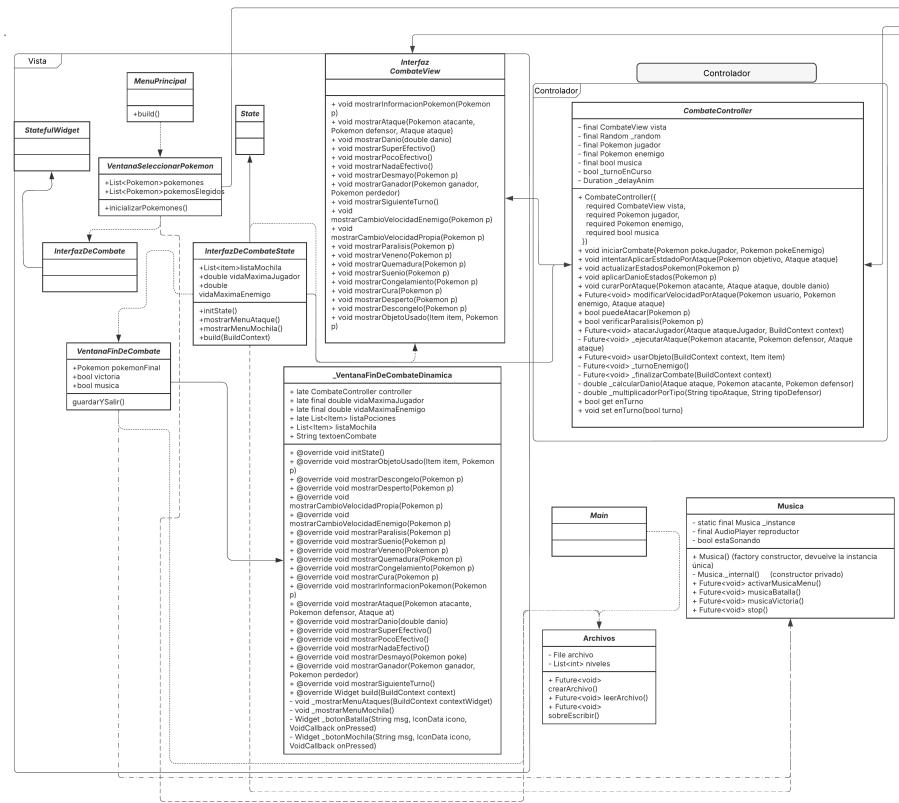
Para la parte del audio, se usa el paquete audioplayers definido previamente en el archivo pubspec.yaml. Después, se crea la clase Musica, la cual tiene como atributos instancia única (Singleton) que asegura que no haya audio sobrepuerto o duplicado durante la ejecución del juego. Después contiene una variable reproductor de tipo AudioPlayer, que es el objeto encargado de reproducir los archivos de audio, y una variable booleana estaSonando que permite llevar el control de si actualmente hay música activa. Por último se definieron los métodos: musicaMenu(), musicaBatalla() y musicaVistoria().

5. Aplicación de los objetivos

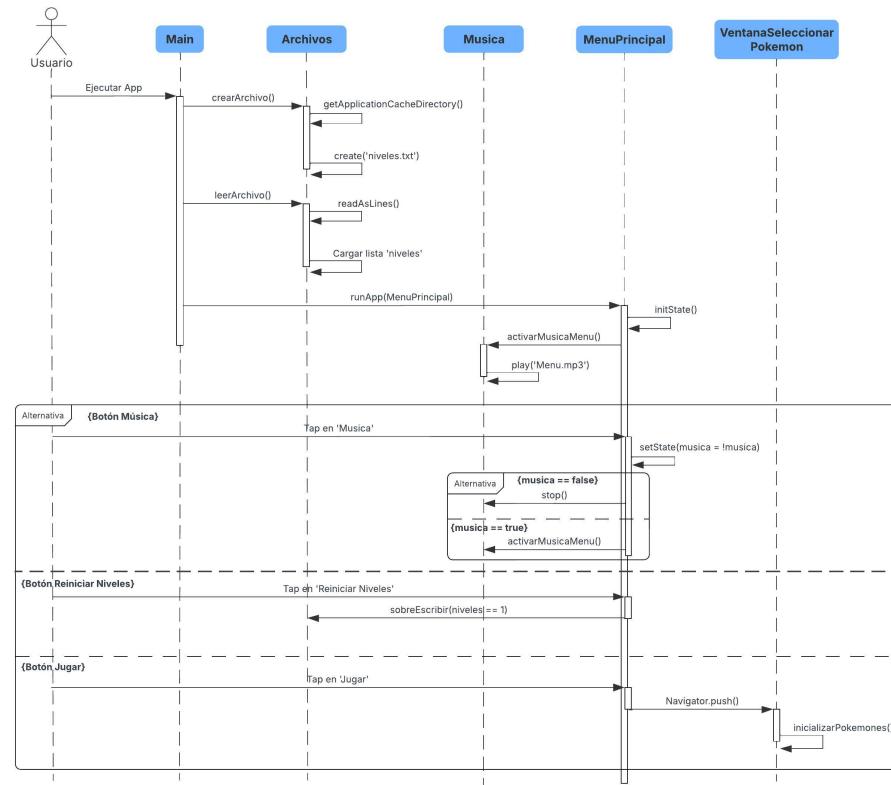
- **Encapsulamiento y paquetes:** El encapsulamiento es aplicado en el uso de las variables, métodos y clases privadas indicadas con '`_`' en *Dart*. El proyecto fue empaquetado siguiendo el esquema *Full Qualified Name*, lo que nos permitió mantener un orden en nuestro proyecto.
- **Herencia y polimorfismo:** Se definieron interfaces y clases abstractas para representar los elementos del juego: Pokemones, ataques e items. Mediante la herencia se crearon sus clases hijas respectivamente lo que nos permitió seguir un mismo modelo y la reutilización de código. El poliformismo permitió almacenar y manipular los objetos creados de estas clases derivadas como objetos de su clase padre, facilitando su manejo.
- **Excepciones:** Se consideraron las excepciones que se podían generar al intentar abrir una imagen, archivo, GIFs o reproducir música. El uso de *try-catch* nos permitió manejar de manera adecuada cualquier anomalía que se presente en estos casos.
- **Archivos:** Se creó un archivo de texto que actualiza y guarda el nivel de los pokemones cada vez que uno sube de nivel. Estos niveles son recuperables incluso cuando la aplicación es cerrada.
- **Hilos:** La aplicación de hilos está dada al usar las funciones asíncronas con retraso. Dentro del juego. Fueron utilizadas al actualizar la interfaz de combate para que se pudieran mostrar los mensajes de daño o efectividad cada vez que un Pokemon sufría daño.
- **Patrones de diseño:** Se utilizó el patrón de diseño **Singleton** para la clase *Musica*, así nos aseguramos que solo exista un objeto de música encargado de la música de todo el programa y evitando música duplicada. Se utilizó el patrón **MVC** para el diseño general del programa, lo que nos permitió implementar la interfaz gráfica con el sistema de combates sin mucha complicación y con una comunicación sencilla.

6. Diagramas UML

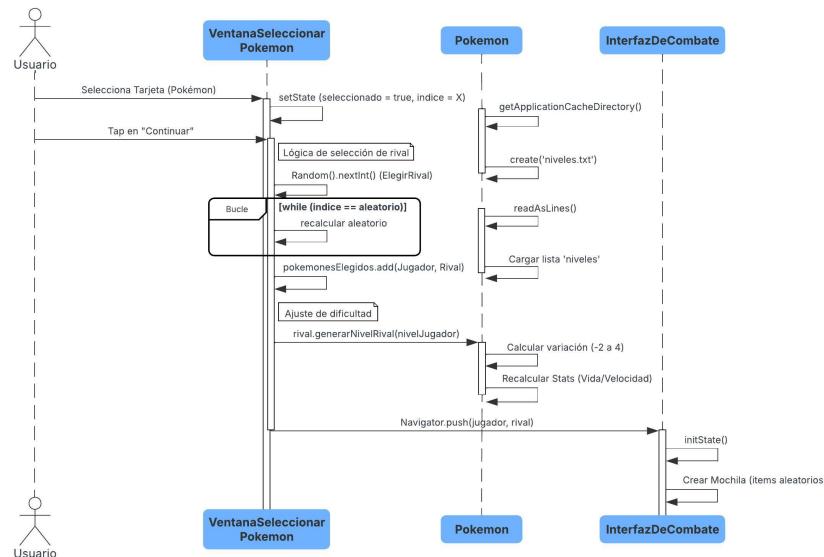
6.1. Diagrama estático:



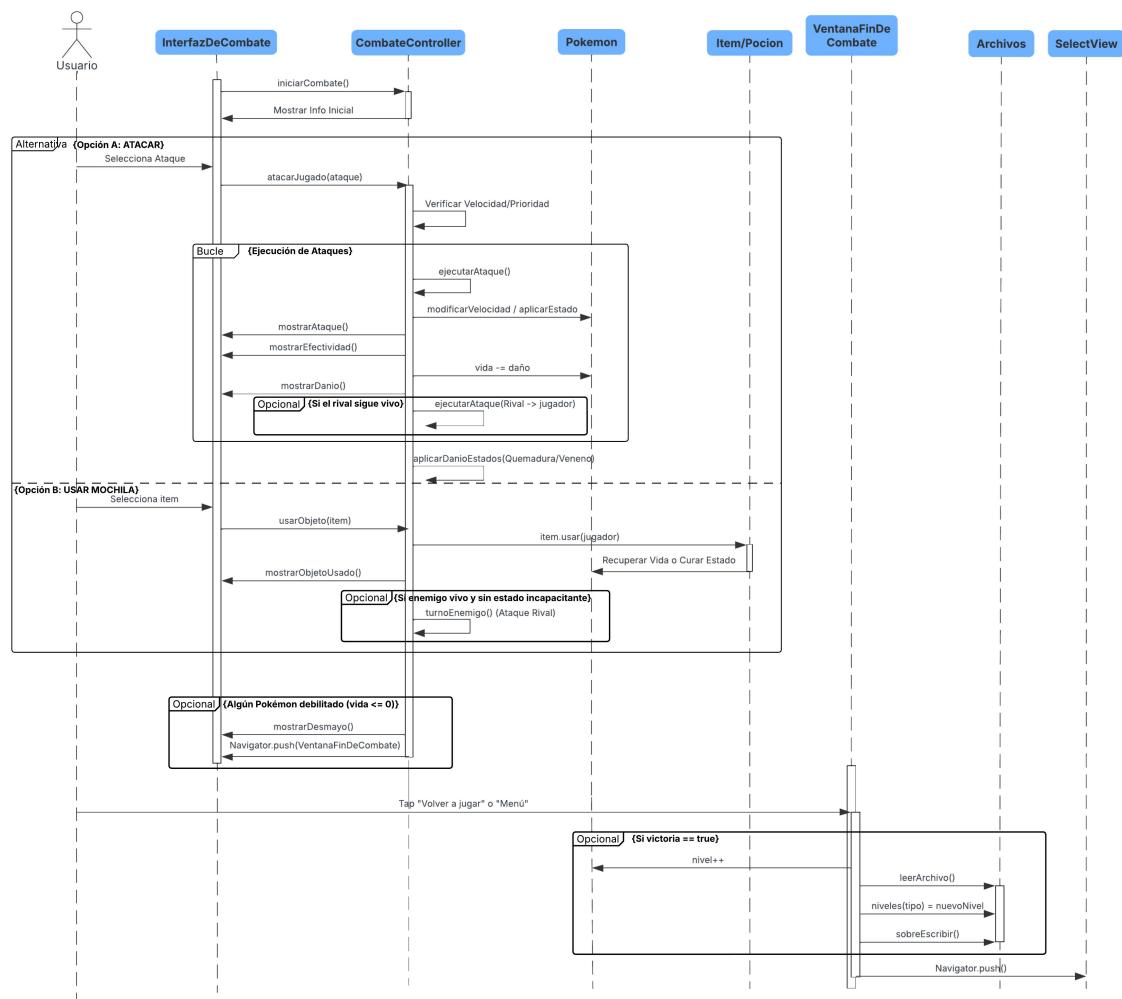
6.2. Diagramas de secuencia:



Inicio de App, Carga de Archivos y Menú Principal



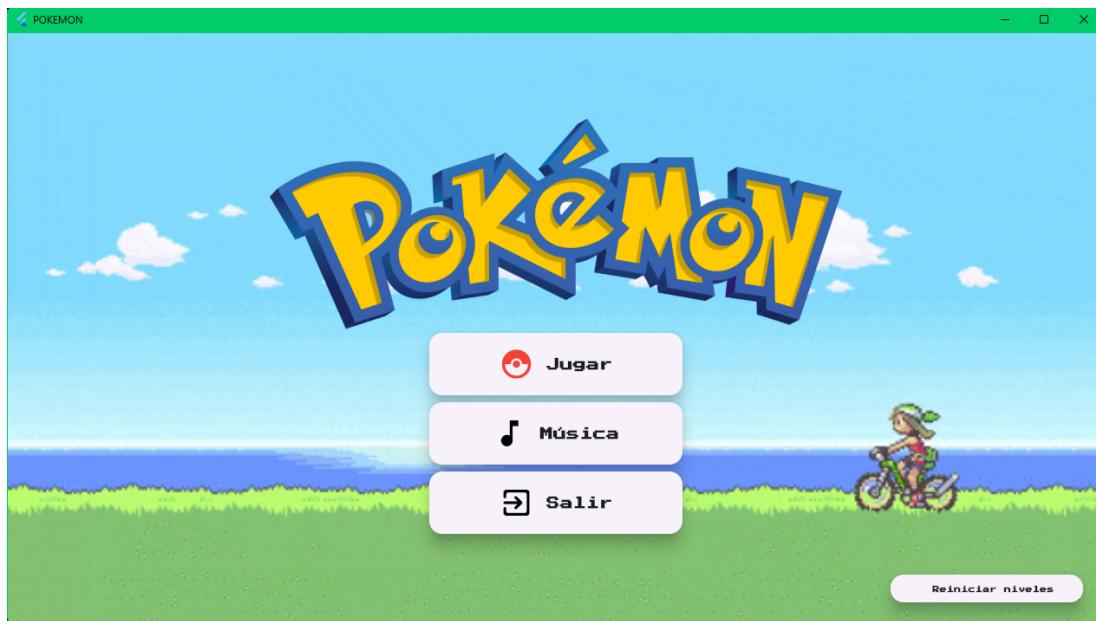
Selección de Pokémon y Generación de Rival



Flujo de Combate, Uso de Items y Guardado

7. Resultados relevantes

Menú principal: Es el inicio del juego, contiene los botones para jugar, activar/cancelar música, salir del programa y un pequeño botón en caso de querer reiniciar el sistema de niveles.



Selección de pokemones: Permite elegir un pokemon para jugar y ver los ataques que hay de cada tipo.

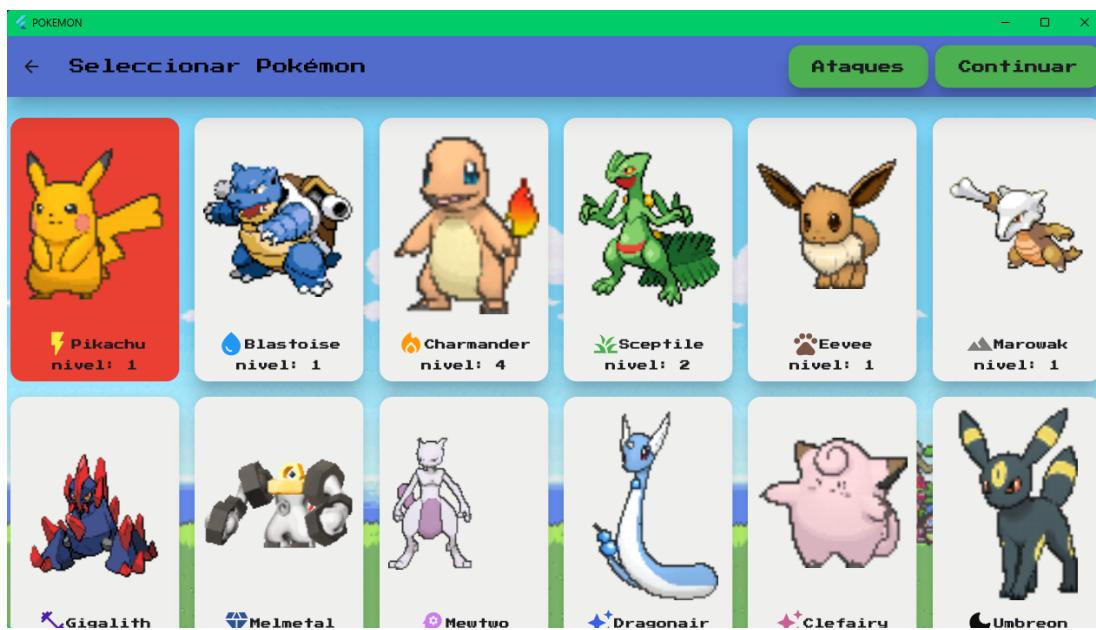


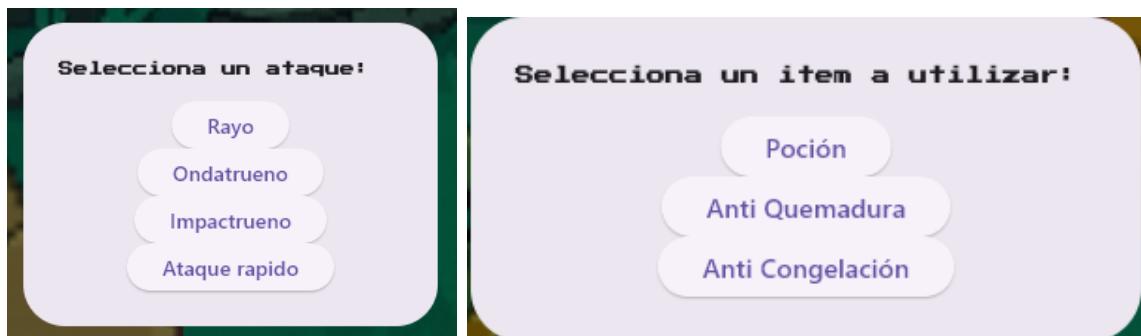
Tabla de ataques: Permite ver los ataques por tipo, su potencia y si pueden generar algún efecto.

Ataque	Potencia	Clase	Efecto
Rayo	90	Especial	—
Ondatrueno	0	Especial	paralisis
Impacttrueno	80	Especial	paralisis
Ataque rapido	40	Fisico	—
Hidro bomba	110	Especial	—
Surf	90	Especial	—
acuajet	40	Fisico	—
Escaldar	65	Especial	quemadura
Llamarada	110	Especial	quemadura
Nitrocarga	50	Fisico	velocidadPropia
Ascuas	40	Especial	quemadura
Latigazo	120	Fisico	—
Brazo pincho	60	Fisico	—
Hoja afilada	60	Fisico	—
Voto planta	50	Especial	—
Alboroto	90	Especial	—
Arañazo	40	Fisico	—
Atizar	80	Fisico	—
Corte	50	Fisico	—
Ataque rapido	40	Fisico	—
Terremoto	100	Fisico	—

Combate: Interfaz de combate, donde se puede visualizar vida, estado y turno del pokémon. En su turno se puede atacar o elegir un objeto de la mochila.



Ataques y mochila: Botones despegables que permiten elegir el ataque u objeto a usar en su turno. Los ataques cambian según el pokémon y los objetos son elegidos aleatoriamente.



Fin de batalla: Ventana de victoria o derrota mostrada al terminar el combate.



El sistema desarrollado permitió implementar un combate funcional entre dos Pokémon utilizando Flutter y Dart bajo una estructura inspirada en el patrón MVC. Realizando una buena división entre los trabajos del controlador, la vista y el modelo. Logrando que el controlador del combate coordine de manera adecuada el flujo del turno, determinando qué Pokémon ataca primero según su velocidad o el ataque usado, aplicando daño conforme a la potencia del movimiento seleccionado y mostrando mensajes de efectividad según las relaciones entre tipos; así como el uso de items, la aplicación de estados y la verificación de desmayos.

Las vistas implementadas en Flutter representan de manera correcta los datos enviados por el controlador, actualizando barras de vida, ataques disponibles, animaciones y mensajes de combate. Además, la clase Música funciona como un Singleton que controla globalmente la reproducción de audio dentro de la aplicación, evitando la creación de múltiples reproductores. Finalmente, el módulo Archivos permite guardar y leer los niveles de los Pokémon, confirmando la persistencia de datos.

8. Conclusiones

Este proyecto nos permitió integrar de manera práctica todos los conceptos estudiados a lo largo del curso de Programación Orientada a Objetos. El desarrollo del juego, nos permitió aplicar conceptos vistos individualmente, en un mismo proyecto, como lo son la herencia y polimorfismo, encapsulamiento y paquetes, archivos, excepciones, funciones asíncronas, patrones de diseño, así como la modularidad del código y la representación del mismo mediante diagramas UML, logrando una implementación completa del proyecto.

El trabajo en equipo, mediante la herramienta de GitHub, permitió la división de responsabilidades, y un control sobre las versiones del código. Además, ampliamos nuestros conocimientos sobre el lenguaje *Dart* y el framework *Flutter*, entendiendo mejor la estructura y organización de un proyecto real y completo, el manejo de dependencias mediante el archivo *pubspec.yaml*, y la importancia de mantener un orden claro por carpetas y archivos para garantizar la escalabilidad del sistema.

Finalmente, este proyecto reforzó nuestros conocimientos teóricos sobre la materia de POO, y nos permitió simular un proyecto completo, el trabajo colaborativo y el diseño de aplicaciones.

9. Referencias

Referencias

- [1] *Patrón singleton: una clase propia.* IONOS Digital Guide. 19 de feb. de 2021. URL: <https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/patron-singleton/> (visitado 01-12-2025).
- [2] Uriel Hernandez. *MVC (Model, View, Controller) explicado.* códigofacilito. URL: <https://codigofacilito.com/articulos/mvc-model-view-controller-explicado> (visitado 01-12-2025).
- [3] *Combate pokémon.* WikiDex. URL: https://www.wikidex.net/wiki/Combate_Pok%C3%A9mon (visitado 01-12-2025).