



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: _____ René Adrián Dávila Pérez _____

Asignatura: _____ Programación Orientada a Objetos _____

Grupo: _____ 01 _____

No. de proyecto: _____ 03 _____

322276824

Integrante(s): _____ 322258516 _____

425037384

320108116

322221415

No. de brigada: _____ 01 _____

Semestre: _____ 2026-1 _____

Fecha de entrega: _____ 01 de diciembre de 2025 _____

Observaciones: _____

CALIFICACIÓN: _____

Índice

1. Introducción	2
2. Marco Teórico	2
2.1. Dart	2
2.2. Clases y Objetos	2
2.3. Herencia y Clases Abstractas	2
2.4. Polimorfismo	3
2.5. Encapsulamiento	3
2.6. Constructores y Parámetros Nombrados	3
2.7. Colecciones y Listas	3
2.8. Manejo de Estados y Lógica de Negocio	3
2.9. Uso de la Biblioteca <code>dart:math</code>	3
2.10. Flutter	3
2.11. Mecánicas de Combate Pokémon	4
2.12. Patrones de Diseño	4
3. Desarrollo	5
3.1. Main	5
3.2. Modelo	5
3.2.1. Pokemon	5
3.2.2. Ataques	5
3.3. Vista	5
3.4. Controlador	6
3.4.1. CombateController	6
3.5. Páginas	6
3.5.1. batalla_page	6
3.5.2. seleccion_page	7
4. Resultados	15
5. Conclusiones	18

1. Introducción

- **Planteamiento del problema:**

Desarrollar una aplicación que simule el sistema de batallas de Pokémon. La aplicación debe gestionar un combate por turnos entre usuario y rival, determinando el orden de ataque con base en la velocidad y finalizando el encuentro cuando la salud de alguno llegue a cero. Asimismo, el sistema debe implementar la lógica de efectividad de tipos de acuerdo con la tabla de tipos y ofrecer una interfaz gráfica interactiva que permita seleccionar ataques y visualizar el estado actual del combate.

- **Motivación:**

La simulación de un sistema de batalla por turnos representa un escenario ideal para la implementación avanzada de POO, ya que requiere la interacción constante entre múltiples objetos con estados complejos. Utilizar Flutter permite aplicar lógica de backend y comprender cómo los objetos se vinculan con una interfaz gráfica reactiva, usando el patrón de diseño MVC y acercando el desarrollo a un entorno de producción de software moderno y funcional.

- **Objetivos:**

Implementar una jerarquía de clases que modele correctamente las entidades del juego aplicando conceptos fundamentales como herencia, polimorfismo, clases abstractas y el patrón de diseño MVC, así como integrar la lógica con la interfaz de usuario mediante el manejo de estados en Flutter, asegurando que las reglas del juego se ejecuten de manera consistente y transparente para el usuario.

2. Marco Teórico

2.1. Dart

Dart es un lenguaje de programación moderno, desarrollado por Google, que combina el poder de la programación orientada a objetos con la facilidad y eficiencia de los lenguajes de programación basados en scripts [1].

2.2. Clases y Objetos

Una clase es una plantilla que define atributos y comportamientos, mientras que un objeto es una instancia concreta de dicha clase. En Dart, las clases permiten representar entidades del sistema como Pokémon o ataques, y facilitan la modularidad del código [2].

2.3. Herencia y Clases Abstractas

Son los mecanismos que permiten crear jerarquías donde una clase base define comportamientos comunes que son extendidos y concretados por subclases específicas. Esto facilita la reutilización de código y el polimorfismo [3].

2.4. Polimorfismo

El polimorfismo permite que un mismo método pueda tener comportamientos diferentes dependiendo del objeto que lo invoque. En el sistema de batalla, ataques de distintos tipos pueden sobrescribir métodos para calcular daño de forma específica.

2.5. Encapsulamiento

Consiste en ocultar los detalles internos de una clase y exponer solo lo necesario mediante métodos públicos. Esto permite gestionar de forma segura atributos como puntos de salud (HP) o velocidad, evitando modificaciones no controladas.

2.6. Constructores y Parámetros Nombrados

Los constructores permiten inicializar objetos y, con los parámetros nombrados, Dart facilita la legibilidad y claridad al crear entidades complejas como ataques o Pokémon con múltiples atributos.

2.7. Colecciones y Listas

Dart proporciona listas dinámicas que permiten almacenar colecciones de objetos, como los movimientos de un Pokémon.

2.8. Manejo de Estados y Lógica de Negocio

La lógica de turnos, aplicación de daño, cambios de estado y verificación de condiciones de victoria se implementan mediante métodos dentro de las clases del sistema. El uso de estructuras condicionales y funciones permite controlar cada etapa del combate.

2.9. Uso de la Biblioteca `dart:math`

Se emplea para generar valores aleatorios en ataques, o variaciones en el daño, mediante la clase `Random`.

2.10. Flutter

Flutter es un *framework* basado en widgets, componentes visuales inmutables que describen la estructura de la interfaz. Cada pantalla o elemento gráfico (selección de ataques, barra de vida, menús) se construye con widgets [4].

Stateful Widgets. Se utilizan cuando la interfaz debe reaccionar a cambios en el estado interno, como la actualización del HP o el avance del combate por turnos.

Gestión de Estado. El método `setState()` permite actualizar variables y reflejar cambios en la interfaz. Esto es fundamental para mostrar en tiempo real la vida restante, los mensajes de combate o la disponibilidad de acciones.

2.11. Mecánicas de Combate Pokémon

HP (Health Points): “Es la condición física del Pokémon, representada con un valor numérico. Estos son reducidos normalmente mediante los ataques del oponente en combate, los efectos del veneno, las quemaduras, o varios climas entre otros medios” [5].

Velocidad: “La velocidad es la propiedad del Pokémon de atacar, antes o después, que el oponente. A la hora de atacar el Pokémon con un mayor valor de velocidad, por lo general, siempre atacará primero”.

Efectividad por Tipos. Cada tipo de ataque puede ser más o menos efectivo dependiendo del tipo del oponente. Esta efectividad se expresa como multiplicadores: neutral (1), supereficaz (2), poco eficaz (0.5) o sin efecto (0) [6].

	Atacante															
Tipo	Normal	Lucha	Volador	Veneno	Tierra	Roca	Bicho	Fantasma	Acero	Fuego	Aqua	Planta	Eléctrico	Psíquico	Siniestro	Hada
Normal	1	x2						x0								
Lucha		1	x2												x2	x½ x2
Volador			1	x2	x0	x2	x½						x½	x2	x2	
Veneno				1	x2	x2	x½						x½	x2		x½
Tierra					1	x½	x½						x2	x2	x0	x2
Roca						1	x½									
Bicho							1	x2	x2	x½	x2	x2	x2	x2	x2	
Fantasma								1	x2	x½	x2	x2				
Acero									1	x2	x½	x2	x2			
Fuego										1	x½	x2	x2	x2		
Aqua											1	x½	x2	x2	x2	
Planta												1	x2	x2	x2	
Eléctrico													1	x2	x2	x2
Psíquico														1	x2	x2
Siniestro															1	x½ x2
Hada																1

Normal Fantasma
Lucha Acero
Volador Fuego
Veneno Aqua
Tierra Planta
Roca Eléctrico
Bicho Psíquico
Siniestro
Hada

<https://pokemonalpha.xyz/>

Figura 1: Tabla de tipos de Pokémon. [6]

2.12. Patrones de Diseño

Modelo–Vista–Controlador (MVC). El patrón MVC separa la lógica de negocio (modelo), la interfaz de usuario (vista) y el flujo de datos (controlador). En Flutter, esta separación se logra mediante la distribución de clases y widgets, permitiendo un proyecto organizado, escalable y sostenible [7].

3. Desarrollo

La aplicación dentro del paquete `mx.unam.fi.poo.p910` permite la ejecución de un programa que simula un juego de batalla PokéMón donde se aplicaron los conceptos principales de programación orientada a objetos tales como herencia, polimorfismo, manejo de errores y el patrón de diseño MVC. A continuación se explica a detalle el funcionamiento de cada parte del código.

3.1. Main

Es el punto de partida del programa, importa la librería `material` para el uso de widgets y `seleccion_page` de `paginas` siendo la primera página que se mostrará en la ejecución, poniendo como widget raíz a `MartialApp`.

3.2. Modelo

En el modelo se encapsula la información de la aplicación y se define la lógica con la que se manipulan los datos, por lo que en esta capa se definieron los siguientes archivos:

3.2.1. Pokemon

Se define la clase **Pokemon** con los atributos básicos que heredan los pokémon de todos los tipos, siendo estos: nombre, nivel, tipo, imagen, velocidad, vida, su lista de ataques y se define un generador aleatorio `Random` para asignar diferentes valores de velocidad y vida a cada pokémon, haciendo al juego más dinámico variando los valores de las estadísticas de los PokéMón. Finalmente se genera una lista `listaPokemon`s que genera un pokémon de cada tipo siendo los pokémon que estarán disponibles dentro de la aplicación.

3.2.2. Ataques

Se genera la clase **Ataque** que incluye los atributos de nombre, tipo y potencia, siendo generadas posteriormente clases hijas que definen el tipo de ataque, así se crean listas que contengan 3 ataques de un tipo específico permitiendo asignar movimientos coherentes a los pokémon de la clase **Pokemon**.

3.3. Vista

El objetivo de los objetos definidos en la capa de vista es mostrar la información del objeto del modelo y habilitar la edición de información enviando la información al controlador, es así que en esta capa se controla el cómo se muestra la información al usuario creando la clase abstracta **CombateView** en la que sus métodos señalan los mensajes que deben mostrarse cuando ocurra una batalla, tales como `mostrarSuperEfectivo()`, `mostrarPocoEfectivo()`, `mostrarNoAfecta()`, etc.

Los métodos son definidos en la clase **FFlutterCombateView** donde todos los métodos se apoyan en `_append` que usa `notifyListeners()` y un atributo `log` para guardar el texto de lo que ocurre en el combate.

3.4. Controlador

El controlador hace de intermediario entre la capa del modelo y la capa de la vista, por lo que esta capa se encarga de decidir los efectos de las acciones en el combate y comunicárselo a la vista. Para esto se define un map **tablaTipos** donde una clave es el tipo de ataque y la otra el tipo del defensor tomando como valor un multiplicador que se define para el daño que un pokémon le puede causar a otro dependiendo del tipo de ambos. Este map es usado en la función `_atacar()` para calcular el daño multiplicando el multiplicador por la potencia del ataque, mostrando el daño hecho y el mensaje definido en `view` sobre la efectividad del ataque.

3.4.1. CombateController

La clase **CombateController** coordina la secuencia de los turnos con ayuda del método `iniciarTurnoFlutter()` que verifica la vida de los dos jugadores y en base al atributo de velocidad ordena los turnos, haciendo uso de la función `_atacar()` y de los mensajes definidos en `combate_view`.

3.5. Páginas

En esta carpeta se precisan las dos páginas que se mostrarán en la aplicación, siendo la de selección de pokemones y la combate, detallando cada una a continuación.

3.5.1. batalla_page

Esta clase implementa la pantalla que visualiza el usuario en el sistema de combate de la aplicación por lo que gestiona la interfaz gráfica y la interacción con el usuario, para esto la clase se extiende de `StatefulWidget` ya que la interfaz cambia con el tiempo, se definen los pokemones.

Se crea un método `initState()` que crea el controlador del combate y la vista de la batalla, añadiendo un método escucha para `flutterView` que notifica a `setState` en caso de algún cambio para bajar el scrollbar hasta el final. Empleando también una función `_reproducirMusica` que usa un `try/catch` y un `async` para reproducir una canción cuando se abra la página.

Para la interfaz se declara el método `_getIconoTipo()` que recibe un tipo de pokémon y devuelve un tipo de ícono a usar. Posteriormente se declara un método `ejecutarTurno()` que llama al controlador de combate para procesar el turno una vez que el usuario ha usado un ataque.

Con respecto a los widgets usados para la interfaz se tienen los siguientes:

- **build()**
Es el método responsable de generar la pantalla de batalla, para organizar la estructura de la pantalla se utiliza un **Scaffold**, una **appBar** que señala la batalla pokemon, el registro de la batalla contenido dentro de **SingleChildScrollView** y un **bottomNavigationBar** que alterna entre los botones principales y los de ataque en la parte inferior de la pantalla.
- **buildPokemonInfo()**
El método construye la tarjeta en la que se muestra la información del pokémon usado y el rival, tornando la vida de rojo cuando es menor a los 500 puntos de vida y organizando la información a través de **Row** para mostrar la información del rival del lado derecho y la del pokémon propio del lado izquierdo.
- **imagenPokemon()**
El método se encarga de mostrar la visualización de la imagen del pokémon, decidiendo su tamaño, utilizando **SizedBox** y mostrando una equis de color opaco cuando los puntos de vida del pokémon sean menores a 0.
- **retroButton()**
El método encapsula el estilo de creación de botones, recibiendo el texto y la función que hará el botón.
- **buildBotonesPrincipales()**
El método revisa la vida de los pokémon, siendo el caso que si los dos siguen con vida se da la opción de huir o luchar, y en caso de que el pokémon propio haya muerto solo se da la opción de ir hacia atrás, generando la página de selección.
- **buildBottomBar()**
El método construye una barra inferior que muestra los ataques cuando la variable **mostrarAtaques** es verdadera, es decir se ha presionado el botón de Lucha, de lo contrario se muestran los botones principales.
- **buildBarraAtaques()**
El método construye la barra donde se muestran los ataques disponibles del pokémon, utilizando el método del botón retro para ejecutar los ataques.
- **buildListaAtaques()** El método construye la lista completa de ataques del pokémon del jugador cuando se elige la opción Lucha. Para organizar su contenido, utiliza un **Row** y muestra cada ataque como un elemento independiente lo que ejecuta el ataque mediante **ejecutarTurno()**.

3.5.2. seleccion_page

Es una clase definida como un **StatefulWidget**, es el punto de entrada a la interacción lógica del usuario y se encarga de gestionar el flujo de elección de los combatientes antes de iniciar la batalla.

Se construyó mediante un Scaffold que aloja un GridView.builder, el cual renderiza la colección de objetos Pokemon disponibles en la lista del modelo. Cada celda de la rejilla encapsula la información del Pokémon (imagen, nombre y tipo) dentro de un widget Card, para una presentación organizada y uniforme.

La lógica de selección se implementó mediante el manejo de estado interno de la clase. Se declaró un atributo nullable Pokemon? jugadorSeleccionado, el cual determina el comportamiento del evento onTap detectado por el GestureDetector. El funcionamiento se divide en dos fases: en la primera, si el atributo es nulo, el objeto seleccionado se asigna al usuario y se invoca a setState() para actualizar la interfaz, cambiando el color de la tarjeta a verde para indicarle al usuario que el Pokémon fue seleccionado, además, la barra superior modifica su título según la fase de selección en la que se encuentre el usuario.

Después, la transición hacia el combate se gestiona en la segunda fase de interacción. Al detectar una segunda selección, la clase instancia la navegación mediante el método Navigator.push, aquí es donde se crea la BatallaPage, inicializándola con las referencias tanto del objeto jugadorFinal como del rivalFinal.

Diagrama de clases (UML estático)

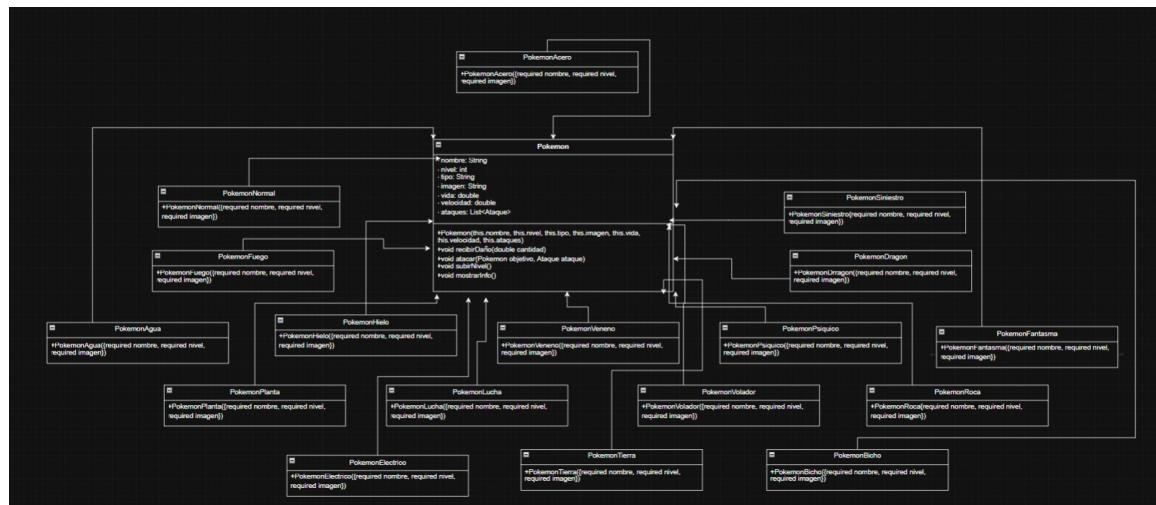


Figura 2: UML Estático: Diagrama de clases de Pokemon.

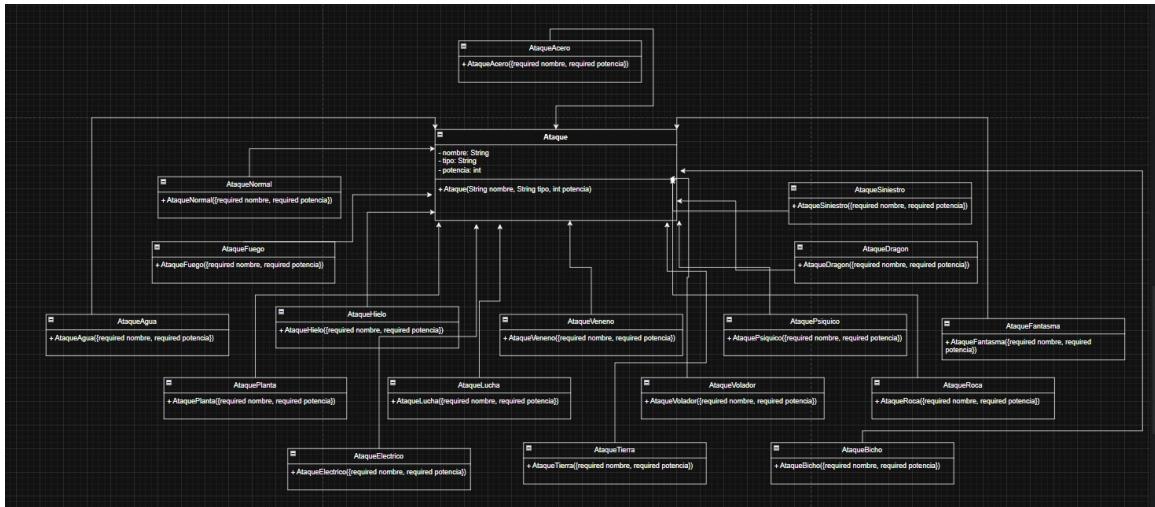


Figura 3: UML Estático: Diagrama de clases de Ataque.

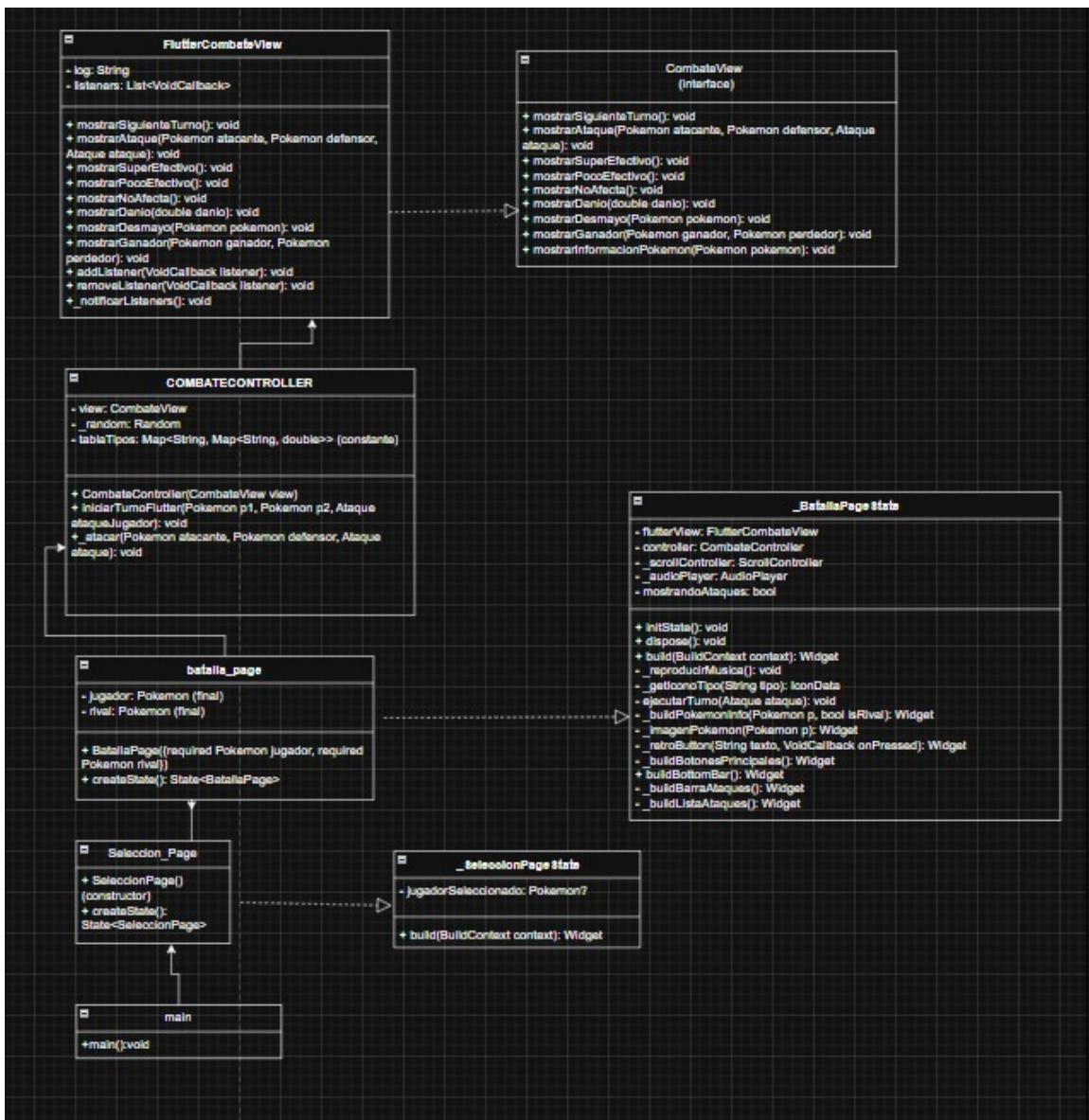


Figura 4: UML Estático: Diagrama de clases de main.

Diagrama de secuencia (UML dinámico)

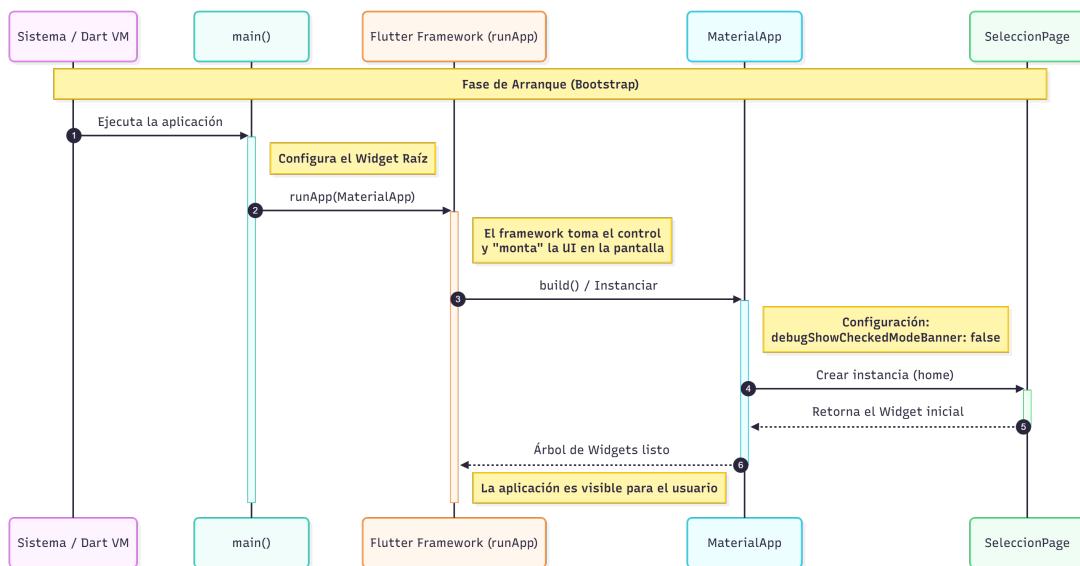


Figura 5: UML Dinámico: Diagrama de secuencia del main.

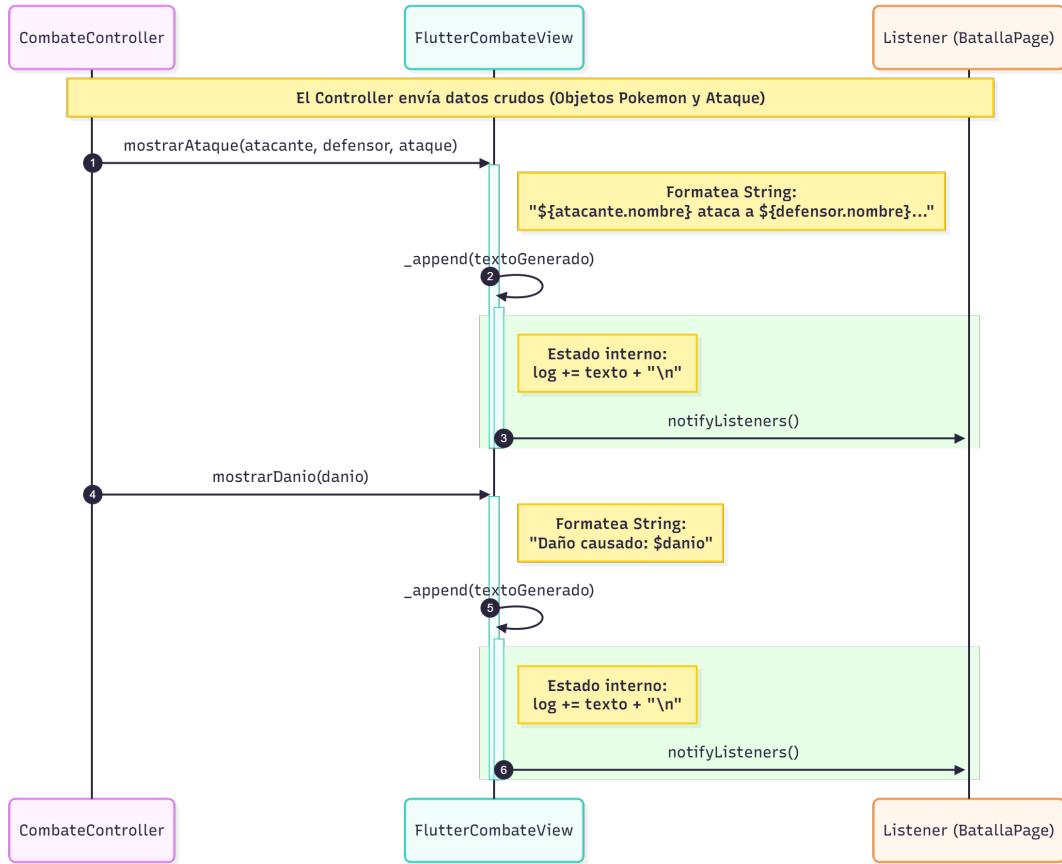


Figura 6: UML Dinámico: Diagrama de secuencia de combate view.

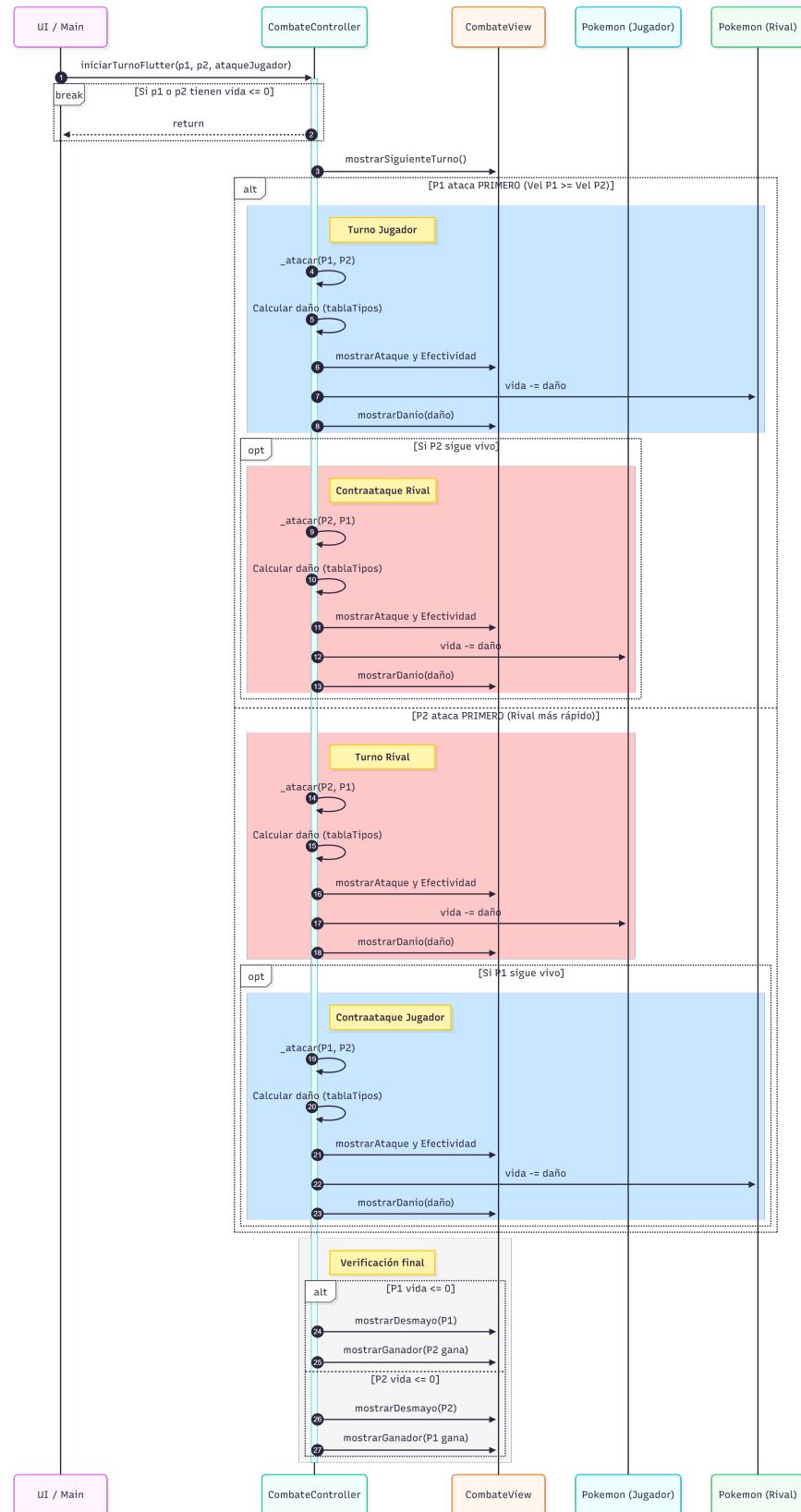


Figura 7: UML Dinámico: Diagrama de secuencia de combate controller.

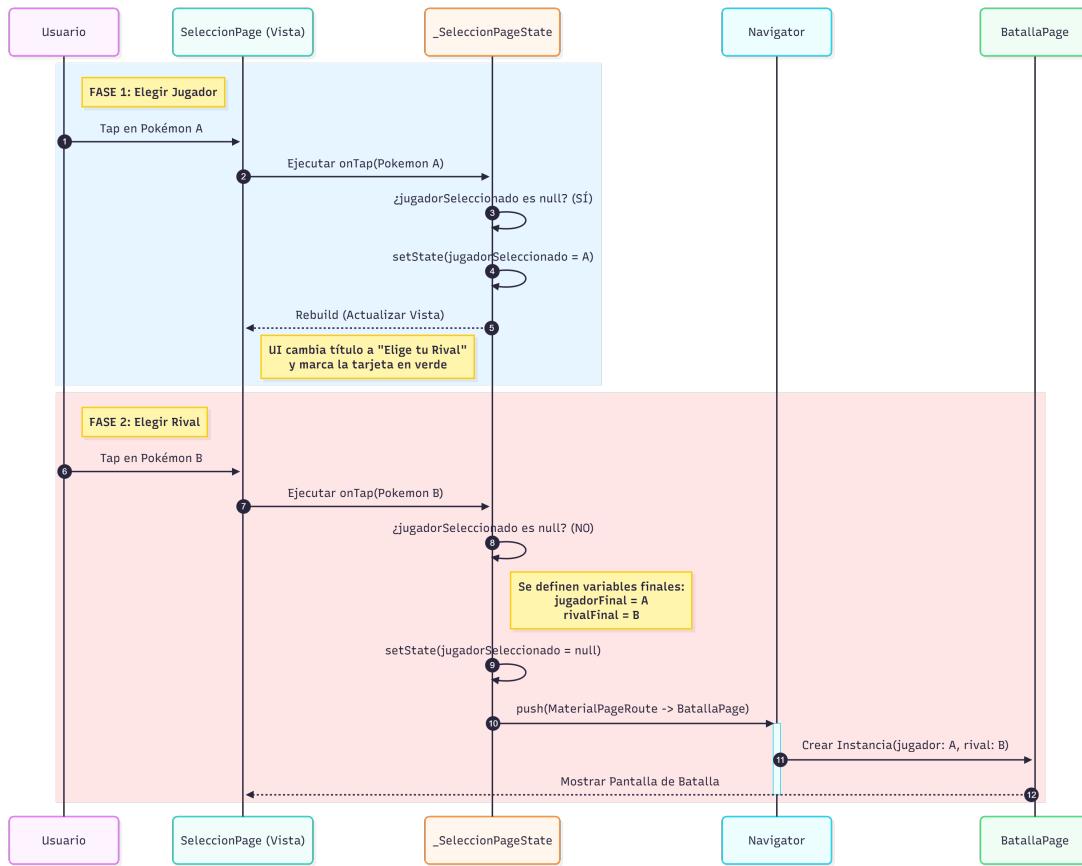


Figura 8: UML Dinámico: Diagrama de secuencia de seleccion page.

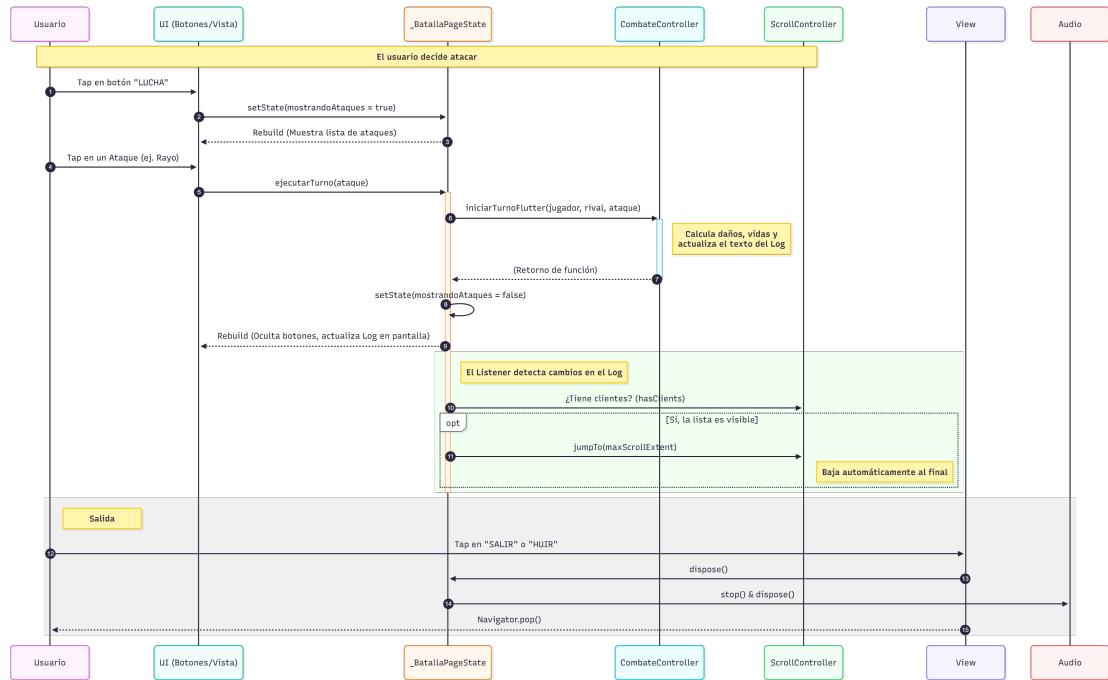


Figura 9: UML Dinámico: Diagrama de secuencia de batalla page.

4. Resultados

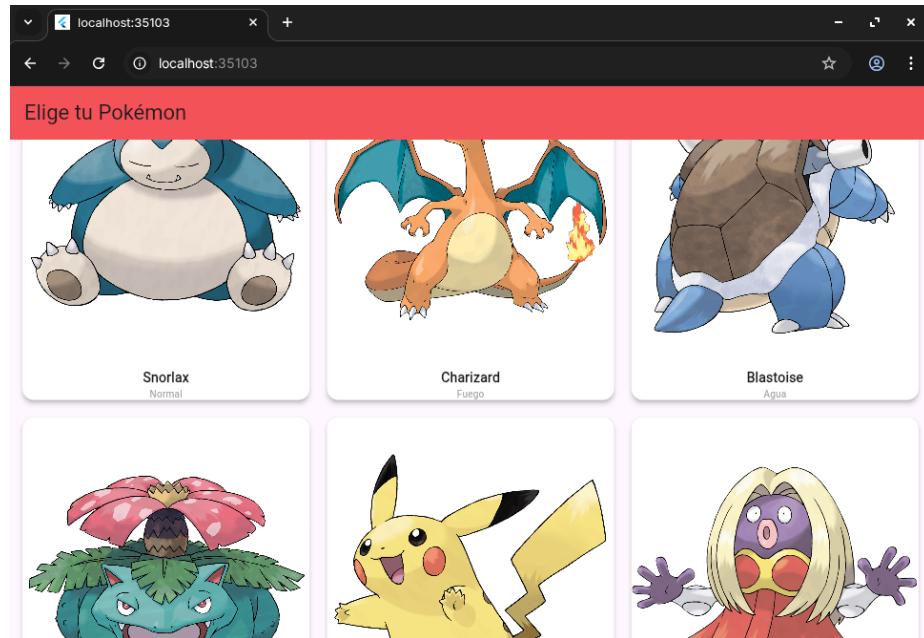


Figura 10: Pantalla inicial donde se escoge el Pokémon del usuario y el Pokémon rival.

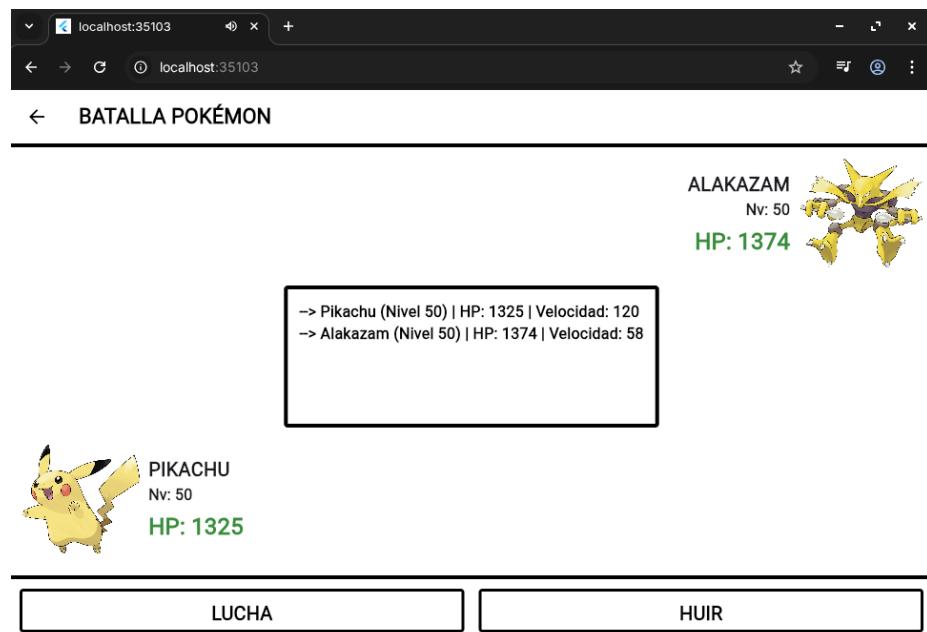


Figura 11: Pantalla de batalla inicial; batalla de exhibición.

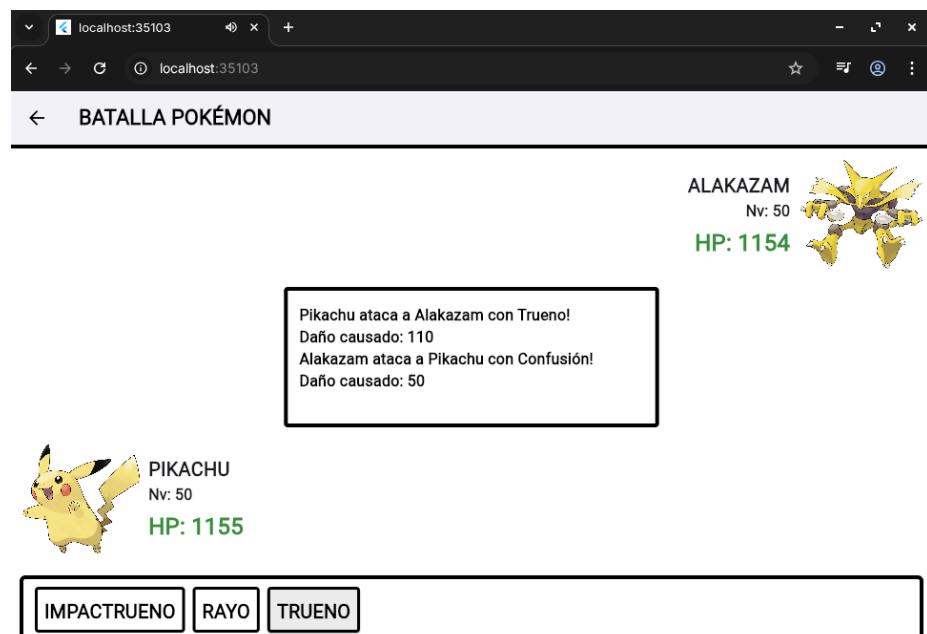


Figura 12: Pantalla de batalla, menú de ataques (mostrando solo los ataques disponibles del Pokémon elegido).

```
Daño causado: 110
Alakazam ataca a Pikachu con Psíquico!
Daño causado: 90
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
Alakazam ataca a Pikachu con Premonición!
Daño causado: 120
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
Alakazam ataca a Pikachu con Psíquico!
Daño causado: 90
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
Alakazam ataca a Pikachu con Confusión!
Daño causado: 50
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
Alakazam ataca a Pikachu con Psíquico!
Daño causado: 90
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
Alakazam ataca a Pikachu con Premonición!
Daño causado: 120
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
Alakazam ataca a Pikachu con Premonición!
Daño causado: 120
--- Siguiente turno ---
Pikachu ataca a Alakazam con Trueno!
Daño causado: 110
¡Alakazam se ha desmayado!
¡Alakazam ha sido derrotado!, ¡Pikachu gana el combate!
```

Figura 13: Pantalla de batalla, historial de batalla que muestra los movimientos realizados por ambos pokemones.

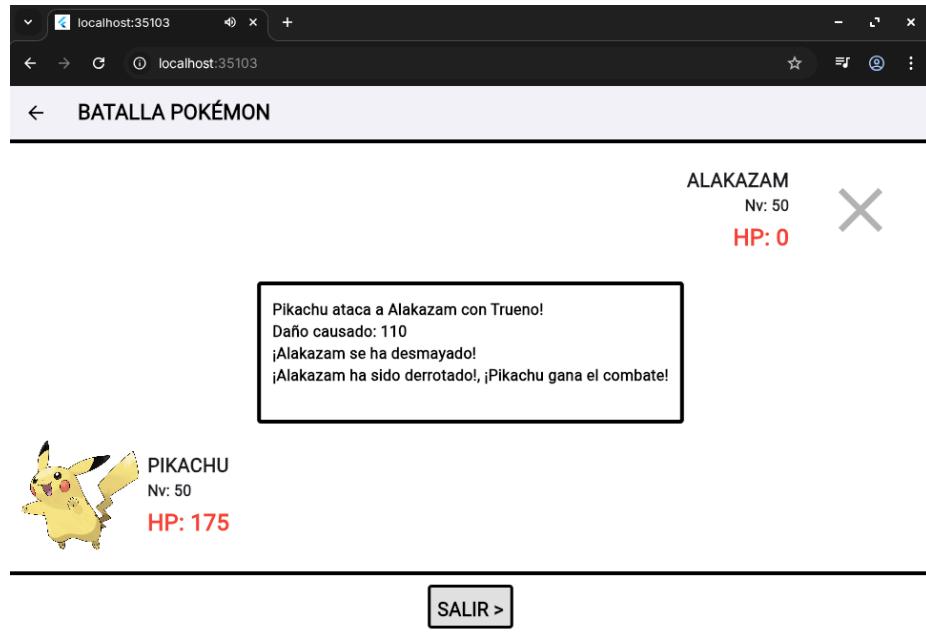


Figura 14: Pantalla de batalla, fin de la batalla con los resultados correspondientes.

5. Conclusiones

El desarrollo de este proyecto permitió integrar de manera efectiva el paradigma de la Programación Orientada a Objetos con la lógica de desarrollo de aplicaciones en Flutter. Se comprobó que el uso de clases abstractas y herencia es esencial para gestionar la diversidad de los elementos de un programa sin duplicar código, permitiendo que el sistema sea escalable y sostenible.

La implementación de la tabla de tipos y el cálculo de daño evidenció la importancia de centralizar las reglas en valores finales para evitar inconsistencias en el estado de la batalla. Además, la gestión de turnos basada en la velocidad reforzó la comprensión del flujo de control y la manipulación de objetos en tiempo de ejecución. Finalmente, el uso de Flutter y el patrón de diseño MVC demostró cómo la programación orientada a objetos sirve como base sólida para construir interfaces gráficas dinámicas, donde cada elemento visual responde a los cambios en los modelos de datos.

Referencias

- [1] Google. **A Tour of the Dart Language**. URL: <https://dart.dev/language>
- [2] Dart Language Documentation. **Classes**. Nov. 2025. URL: <https://dart.dev/language/classes>
- [3] Dart Language Documentation. **Object-Oriented Programming in Dart**. Nov. 2025. URL: <https://dart.dev/language/extend>

- [4] Flutter Documentation. **Architectural Overview**. Dic. 2025. URL: <https://docs.flutter.dev/resources/architectural-overview>
- [5] Wikidex. **Mecánica - Características**. Nov. 2025. URL: <https://www.wikidex.net/wiki/Caracter%C3%ADsticas>
- [6] Wikidex. **Mecánica - Tipos**. Nov. 2025. URL: https://www.wikidex.net/wiki/Tipo#Efectividades_de_los_tipos
- [7] Rhm, F. (2023). **Understanding MVC Architecture in Flutter: A Comprehensive Guide with Examples**. Medium. URL: https://medium.com/@Faiz_Rhm/understanding-mvc-architecture-in-flutter-a-comprehensive-guide-with-examples-5d1