	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: POO

Grupo: 01

No de Práctica(s): Proyecto 3

Integrante(s): 322125337

322080869

322059179

323629814

322113536

No. de lista o brigada: 6

Semestre: 2025

Fecha de entrega: 01/12/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introduccion 2

2. Marco Teórico 2

3. La aplicacion de los conceptos aplicados en cada objetivo. Con referencias 4

4. Desarrollo paso a paso 8

5. Resultados 11

6. Conclusiones 15

7. Referencias bibliográficas 15

1. Introduccion

- **Planteamiento del problema**: Necesitamos desarrollar una aplicación interactiva que simule un sistema de batalla Pokémon donde el usuario pueda seleccionar un compañero asi como también elegir a un rival y ejecutar una batalla por turnos, para esto debemos modelar distintos elementos esenciales del combate: Pokémon con atributos propios, ataques con diferentes tipos y daños, objetos de curación y una tabla de efectividades entre elementos, además es necesario implementar estas entidades utilizando clases, herencia y composición, además de administrar correctamente la interacción entre ellas dentro de una interfaz gráfica construida en Flutter.

- **Motivación**:El uso de clases, herencia y encapsulamiento nos permite representar de forma estructurada conceptos como Pokémon, ataques y objetos, facilitando la reutilización del código y permitiendo un combate más flexible, asi como también modelar comportamientos como daño, velocidad, efectividad y curación demuestra la importancia de aplicar programación orientada a objetos en aplicaciones interactivas, además implementar todo en Flutter permite reforzar la comprensión del desarrollo multiplataforma, la navegación entre pantallas y la gestión de estados dentro de una aplicación móvil moderna.

- **Objetivos**:

  - Aplicar herencia y composición para construir clases como Pokemon, Ataque y Consumibles comprendiendo cómo se relacionan y colaboran en un sistema más grande.
  - Implementar una interfaz en Flutter permitiendo al usuario seleccionar Pokémon, visualizar estadísticas, elegir acciones y observar el resultado del combate en tiempo real.
  - Demostrar el uso del polimorfismo mostrando cómo distintas clases de consumibles o ataques pueden ser procesadas de forma uniforme durante la batalla.
  - Simular una batalla por turnos incorporando elementos como ataques, daño, efectividad, velocidad y registro de acciones con el fin de reforzar el uso de métodos, atributos y lógica orientada a objetos.

2. Marco Teórico

Trabajado por Bautista Benitez Edgar Saul

Flutter: Flutter es un framework multiplataforma desarrollado por Google que permite crear aplicaciones nativas para Android, iOS, web y escritorio desde un único código base. Su arquitectura se centra en widgets, lo que facilita la construcción de

interfaces reactivas y altamente personalizables. Además, su motor gráfico propio permite que la interfaz no dependa del sistema operativo, otorgando consistencia visual y mejor rendimiento, cualidades ideales para aplicaciones interactivas como simuladores de batalla.[1]

Manejo de archivos: El manejo de archivos se refiere al conjunto de operaciones que permiten leer, escribir, almacenar o modificar información de forma persistente. En Dart, se utiliza la librería dart:io, la cual proporciona clases como File y Directory para manipular datos del sistema de archivos cuando la plataforma lo permite. Este mecanismo es esencial en aplicaciones que requieren almacenar configuraciones, progreso o recursos dinámicos, asegurando que la información permanezca disponible incluso después de cerrar la aplicación.[2]

Excepciones: Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de instrucciones. Dart ofrece un manejo estructurado de excepciones mediante try, catch y finally, lo que permite capturar errores y evitar fallos inesperados. Esto es especialmente importante en aplicaciones que dependen de recursos externos, como imágenes de internet o datos dinámicos, ya que permite que la aplicación continúe ejecutándose de manera controlada, preservando la estabilidad del sistema. [3]

Hilos de ejecucion:La programación asíncrona permite ejecutar tareas de forma no bloqueante, lo que mantiene fluida la interacción con el usuario incluso cuando se realizan operaciones de larga duración. En Flutter, el uso de Future, async/await y Stream permite coordinar procesos asincrónicos como animaciones, cálculos y tiempos de espera sin bloquear el hilo principal. Esto resulta fundamental en el desarrollo de una batalla por turnos, donde es necesario ejecutar secuencias temporizadas sin afectar la interfaz gráfica.[4]

Patrones de Diseño en Flutter:Los patrones de diseño permiten organizar el código de manera modular y escalable. En Flutter, uno de los patrones más utilizados es MVVM (Modelo–Vista–VistaModelo), donde el modelo gestiona los datos, la vista representa la interfaz y el ViewModel contiene la lógica de presentación que comunica ambos niveles. Este patrón facilita la separación de responsabilidades, mejora la mantenibilidad y permite construir interfaces reactivas mediante gestión de estado. Flutter también soporta arquitecturas similares a MVC, donde se separa la lógica de negocio, la vista y el controlador, permitiendo organizar estructuras de proyectos incluso en aplicaciones pequeñas. [5] [6] [7]

3. La aplicacion de los conceptos aplicados en cada objetivo. Con referencias

Trabajado por Bautista Benitez Edgar Saul

El proyecto realizado implementa de manera correcta el paradigma de la Programación Orientada a Objetos estructurando su lógica con clases como Pokemon, Ataque, Consumibles, entre otras, esta organización nos muestra bien el concepto de objeto como entidad que encapsula estado (atributos como vida, velocidad, tipo) y comportamiento (métodos para atacar, curarse, usar objetos), esta definición de objeto como combinación de estado y comportamiento es una de las bases de POO. [9]

También tenemos que dentro de la logica del combate se utilizan tipos primitivos (números, cadenas), estructuras de datos complejas (listas, conjuntos, mapas) y control de flujo (condicionales, bucles, lógica de turnos). Como por ejemplo el sistema compara la velocidad de los Pokemon para definir el orden de ataque, calcula daño a través de expresiones numéricas y verifica cuándo la vida de un Pokémon llega a cero para terminar la batalla, además el uso de estos tipos, las expresiones y el control de flujo dentro de un contexto orientado a objetos demuestra una correcta aplicación de estos conceptos tal como describen los manuales de POO. [9]

El proyecto también usa herencia y polimorfismo, lo podemos observar en las clases del tipo consumible que están organizadas jerárquicamente: Consumibles es la clase base y de esta derivan ConsumiblesCuracion y ConsumiblesEstado, añadiendo atributos y comportamientos específicos, además gracias a la herencia se reutilizan datos y métodos comunes y gracias al polimorfismo estas subclases pueden ser tratadas de manera genérica cuando se manipula una colección de consumibles, siendo asi que coincide con la definición formal de polimorfismo en POO: un mismo tipo base puede tener múltiples implementaciones concretas y la ejecución del método depende del tipo real del objeto. [9]

Aparte aunque el proyecto no utiliza bloques explícitos de manejo de excepciones (try/catch) sí implementa validaciones internas para mantener la integridad del estado, como por ejemplo al actualizar la vida de un Pokémon el setter asegura que no sobrepase la vida máxima ni baje de cero asi como también al usar objetos consumibles se verifica que la cantidad exista y se reduce correctamente excluyendo objetos cuando su cantidad llega a cero, estas prácticas representan un manejo preventivo de errores que evita estados inválidos en tiempo de ejecución, este enfoque aunque es algo básico es consistente con recomendaciones de diseño orientado a objetos cuando se busca robustez frente a entradas erróneas o estados límite. [9]

Por otro lado en la parte de almacenamiento de datos aunque no se utilizan archivos externos el proyecto emplea colecciones estáticas (listas de Pokémon, ataques, consumibles) como repositorio interno esta solución es válida cuando no es necesario

persistir datos entre sesiones y además permite al sistema operar con un conjunto fijo de entidades, con esta forma de hacerlo el "repositorio en memoria" se encuentra contemplado dentro de los paradigmas de diseño de software orientado a objetos como una estrategia simple de gestión de datos cuando la persistencia no es crítica. [9]

Después tenemos lo hilos de ejecución que si bien no se maneja explícitamente, el uso de Flutter permite al programa gestionar la lógica del juego y la interfaz sin bloquearse y con facilidad podría adaptarse a asincronía (por ejemplo, delays entre turnos, animaciones, actualizaciones), esto no contradice el concepto de hilos de ejecución ya que la arquitectura sigue organizada en objetos y la ejecución asíncrona o basada en eventos no afecta la modularidad ni encapsulación.

Por ultimo el diseño del sistema refleja la aplicación de patrones de diseño, la lista global de Pokémon y ataques funciona como un repositorio/fábrica de instancias (una fuente centralizada de objetos reutilizables), la jerarquía de clases (ataques, consumibles) aprovecha la herencia y polimorfismo para permitir extensibilidad sin necesidad de modificar código base, además el encapsulamiento de atributos sensibles (vida, velocidad) resguarda la consistencia interna y la lógica de combate que selecciona el ataque según tipo y acelera según velocidad podría verse como una estrategia dinámica en la que el algoritmo de resolución del turno depende del estado de los objetos implicados, asi es como estas técnicas de diseño ayudan a mantener el código modular, limpio, mantenible y escalable, principios centrales de la ingeniería de software orientada a objetos. [9] [10]

Diagrama UML estático:
Trabajado por Alfaro Carreto Irving Alberto

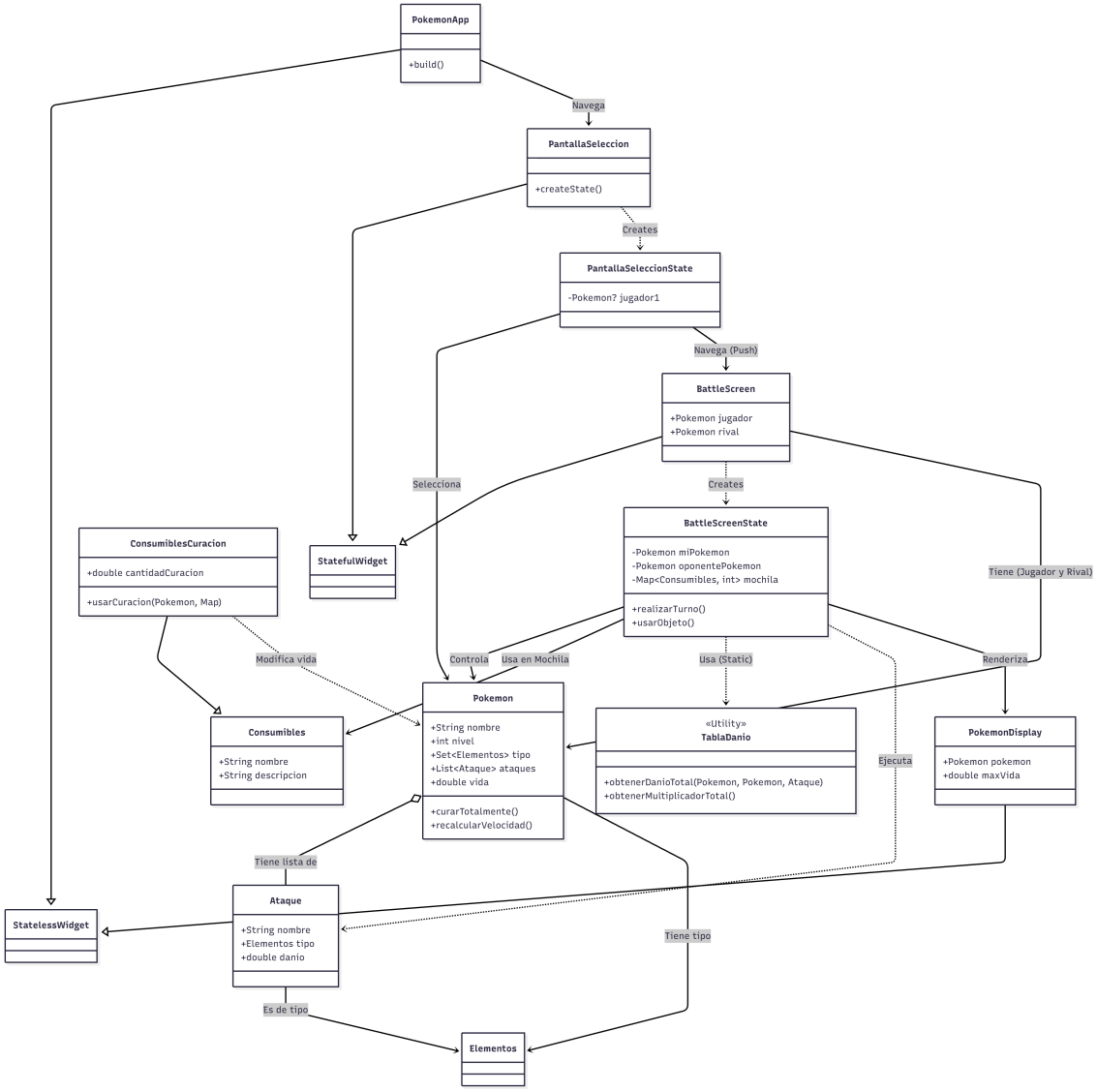
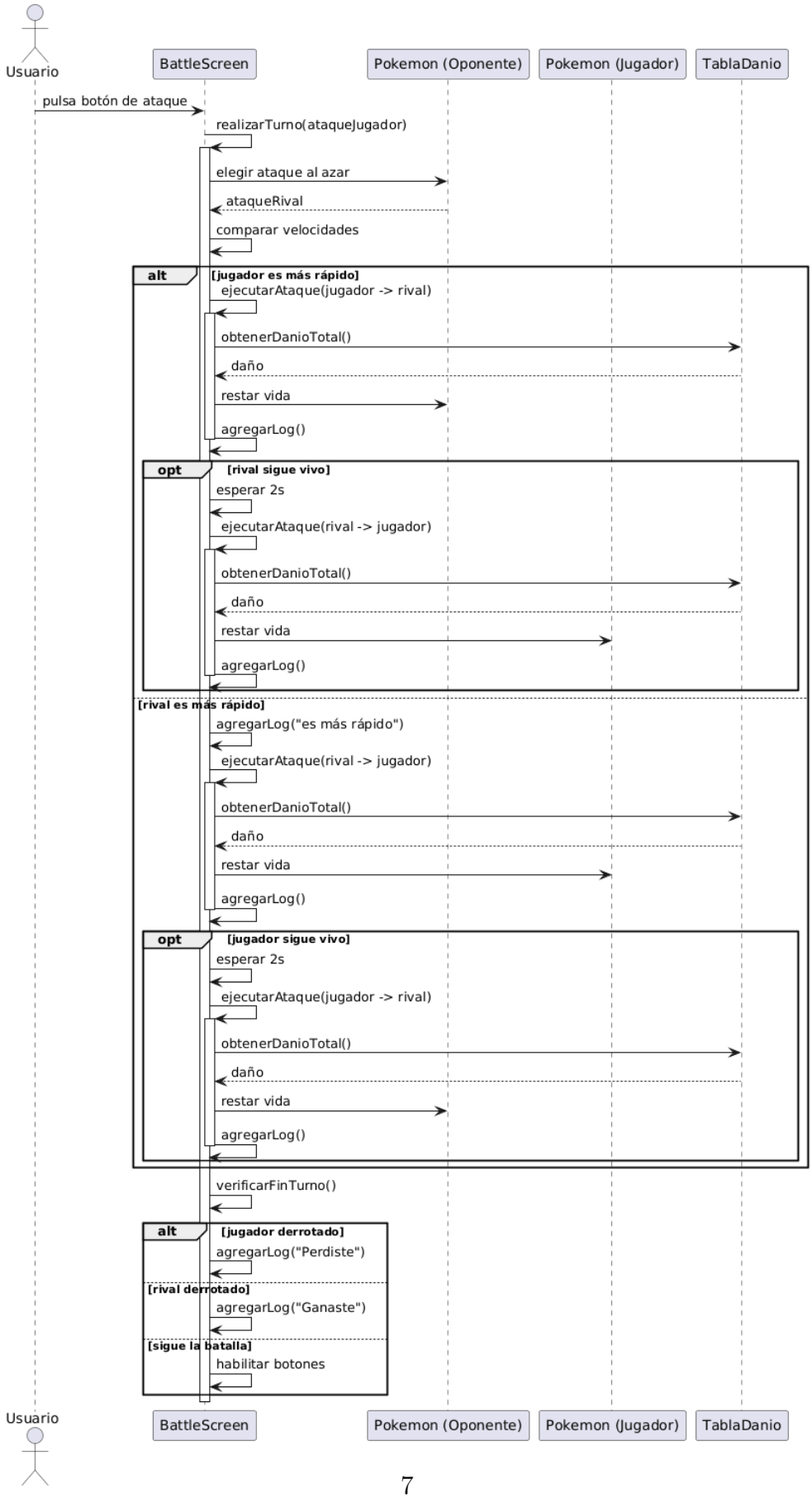


Diagrama UML Dinámico:
Trabajado por Urzúa Sánchez Enrique


Usuario
BattleScreen
Pokemon (Oponente)
Pokemon (Jugador)
TablaDanio
pulsa botón de ataque
realizarTurno(ataqueJugador)
elegir ataque al azar
ataqueRival
comparar velocidades
alt
[jugador es más rápido]
ejecutarAtaque(jugador -> rival)
obtenerDanioTotal()
daño
restar vida
agregarLog()
opt
[rival sigue vivo]
esperar 2s
ejecutarAtaque(rival -> jugador)
obtenerDanioTotal()
daño
restar vida
agregarLog()
[rival es más rápido]
agregarLog("es más rápido")
ejecutarAtaque(rival -> jugador)
obtenerDanioTotal()
daño
restar vida
agregarLog()
opt
[jugador sigue vivo]
esperar 2s
ejecutarAtaque(jugador -> rival)
obtenerDanioTotal()
daño
restar vida
agregarLog()
verificarFinTurno()
alt
[jugador derrotado]
agregarLog("Perdiste")
[rival derrotado]
agregarLog("Ganaste")
[sigue la batalla]
habilitar botones
Usuario
BattleScreen
Pokemon (Oponente)
Pokemon (Jugador)
TablaDanio

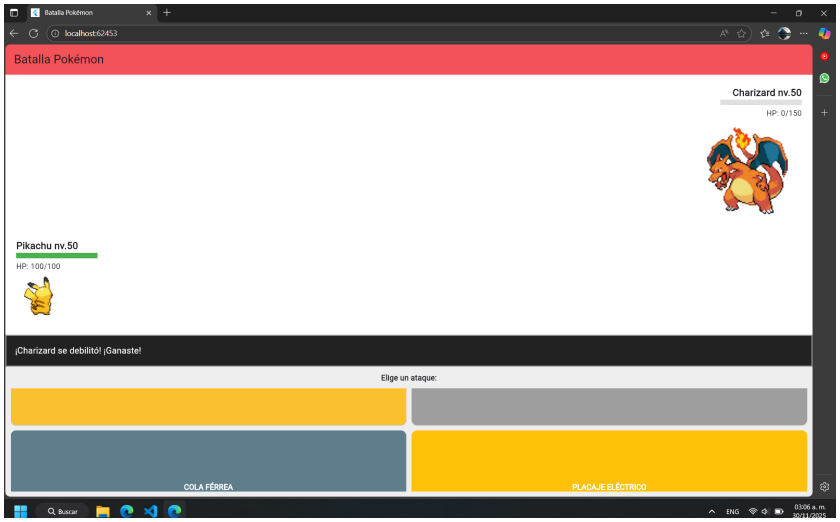
4. Desarrollo paso a paso

1. Creamos un archivo pokedex inspirandonos en la logica del codigo proporcionado.
   Primero creamos el archivo donde definimos a la clase pokemon junto con todos los atributos necesarios con sus gets y sets esto lo hicimos inspirandonos del archivo que se nos proporcionó para crear la clase y además añadimos una lista que será la encargada de guardar a todos los pokemon creados con sus diferentes ataques y tipos.

2. Creamos un archivo de ataques igualmente inspirados en la logica del archivo proporcionado.
   Aquí estructuramos en una clase como va a ser un ataque y creamos varios ataques del fuego original aquí es donde se guarda la tabla de tipos con las fortalezas y debilidades de cada ataque esto lo implementamos mediante un map de Elementos que contenian otro map de Elementos para que en cada uno se vea reflejado los elementos contra los que es fuerte o debil.

3. Se modifico la logica de ataque del profe y se añadio a el archivo ataques.
   Después se colocó basándonos en el código que se nos mostró en clase un sistema de daño para calcular el daño en cada pokemon en combate, el sistema se basa en el calculo de el nivel del pokemon multiplicado por el daño del ataque mandado y eso multiplicado nuevamente por la efectividad del ataque.

4. Se creo una interfaz grafica independiente con pokemon simulados.
   Antes de utilizar la lógica que tenemos echamos un vistazo a nuestras opciones al momento de representar la batalla de los pokemon despues de investigar optamos por utilizar un repositorio en github llamado PokeAPI con todos los sprites de los pokemon o objetos que podamos llegar a añadir. Después de experimentar un poco con los widget la primera versión del combate quedó de la siguiente manera:



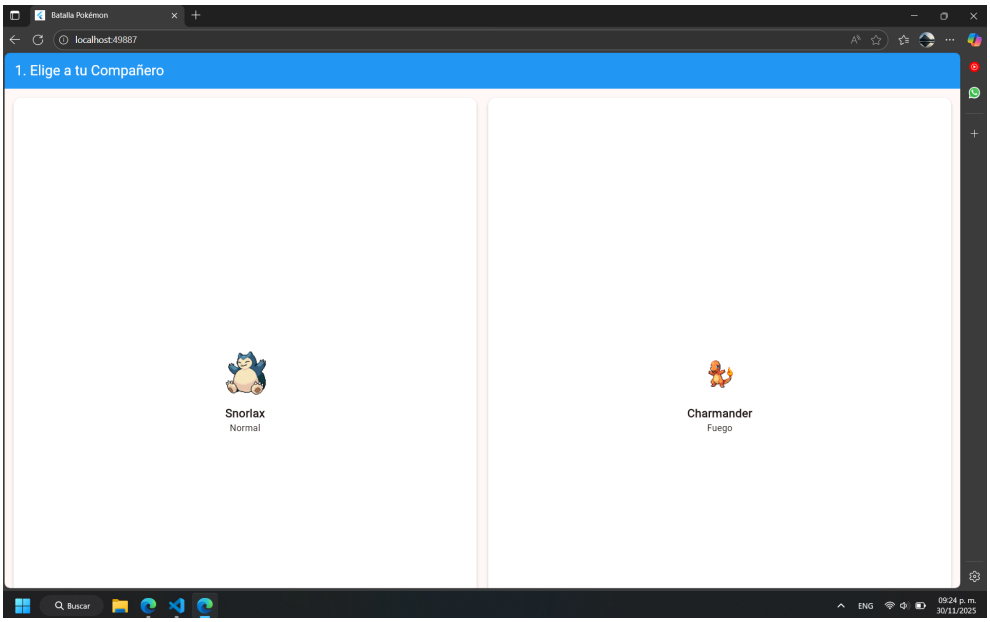
5. Se mejoraron los botones de la interfaz y se añadieron sprites de PokeAPI en el archivo pokedex.
   A partir de la interfaz anterior corregimos varios problemas con el tamaño del que sería el campo de batalla que para solucionar le colocamos un límite para que los botones de los ataques sean visibles y la distancia entre un pokemon y otro no sea demasiada ademas de añadir los pokemon restantes de cada uno de los tipos todos con los sprites de PokeAPI mediante 2 atributos que serían las ligas de los sprites de atrás y de frente.

6. Se unificaron los archivos pokedex, ataques, main (interfaz grafica).
   En este punto unimos todos los archivos que realizamos anteriormente con el objetivo de ver cómo la lógica trabaja junto con la interfaz realizada sustituimos varias funciones donde se encontraban los pokemon y ataques simulados de la interfaz el resultado final de este punto fue la interfaz con la lógica de ataque y daño funcional pero sin el atributo de la velocidad todavía.

7. Se añadio que el texto se quede reflejado a lo largo de la pelea.
   Colocamos un widget de una barra deslizante para que los ataques que se vayan lanzando se queden reflejados en la pantalla pero que simplemente se vayan desplazando hacia abajo.

8. Se añadio los assets para el fondo de la pelea. Se modificó el pubspec.yaml del flutter para poder añadir imágenes que en este caso sería el fondo que se usaría en el fondo de la pelea y dentro del Build de main metimos la imagen para que se mostrará durante toda la pelea.

9. Se añadio el menu de seleccion de pokemon antes de la pelea.
   Una vez que verificamos que las peleas de pokemon funcionaba comenzamos a probar la opción de desplegar con un widget los pokemon disponibles para que antes de la pelea se seleccionen con los que quieres pelear y realizamos el mismo proceso de limitar el espacio de pantalla de este menú para que sea el mismo que el del campo de batalla ahora los pokemon seleccionados en ese menu seran el pokemon propio y el pokemon oponente en la pelea.



10. Se arreglo el sistema de velocidades para que se respeten los turnos Ya que aunque los pokemones si tenían un atributo de pelea todavía no se veía reflejado del todo en la batalla por lo que tuvimos que realizar modificaciones a la función de realizarTurno para que se comparan las velocidades del pokemon propio y el pokemon oponente y que dependiendo de la velocidad se vea reflejado antes el ataque por ello creamos la función de ejecutar ataque para poder realizar correctamente los turnos.

11. Se crea el archivo de consumibles y se añaden assets de pociones Se creó un archivo nuevo destinado a los objetos en batalla primero declaramos la clase de objetos curativos que estarían encargados de curar a nuestros pokemon en pelea como en el juego original esto mediante el método dentro de la clase de ConsumiblesCuracion estos objetos se realizaron igual que los pokemon en un Map para que fueran fácil de desplegarlos en la pantalla gráfica de la misma forma en la que desplegamos los ataques además creamos una clase de ConsumiblesEstado para ataques que pudiera generar efectos en los pokemon como dormir o parálisis pero finalmente no nos dio tiempo de implementar por lo que solamente añadimos las pociones principales del juego.

12. Se modifica la interfaz grafica para poder acceder a los consumibles Finalmente añadimos dos botones principales que dependiendo de cual presiones se desplegaran mediante el Map correspondiente a cada uno los ataques o los objetos en la mochila que en este caso colocamos que se den ciertas pociones de manera aleatoria para poder usarlas en cada pelea.

5. Resultados

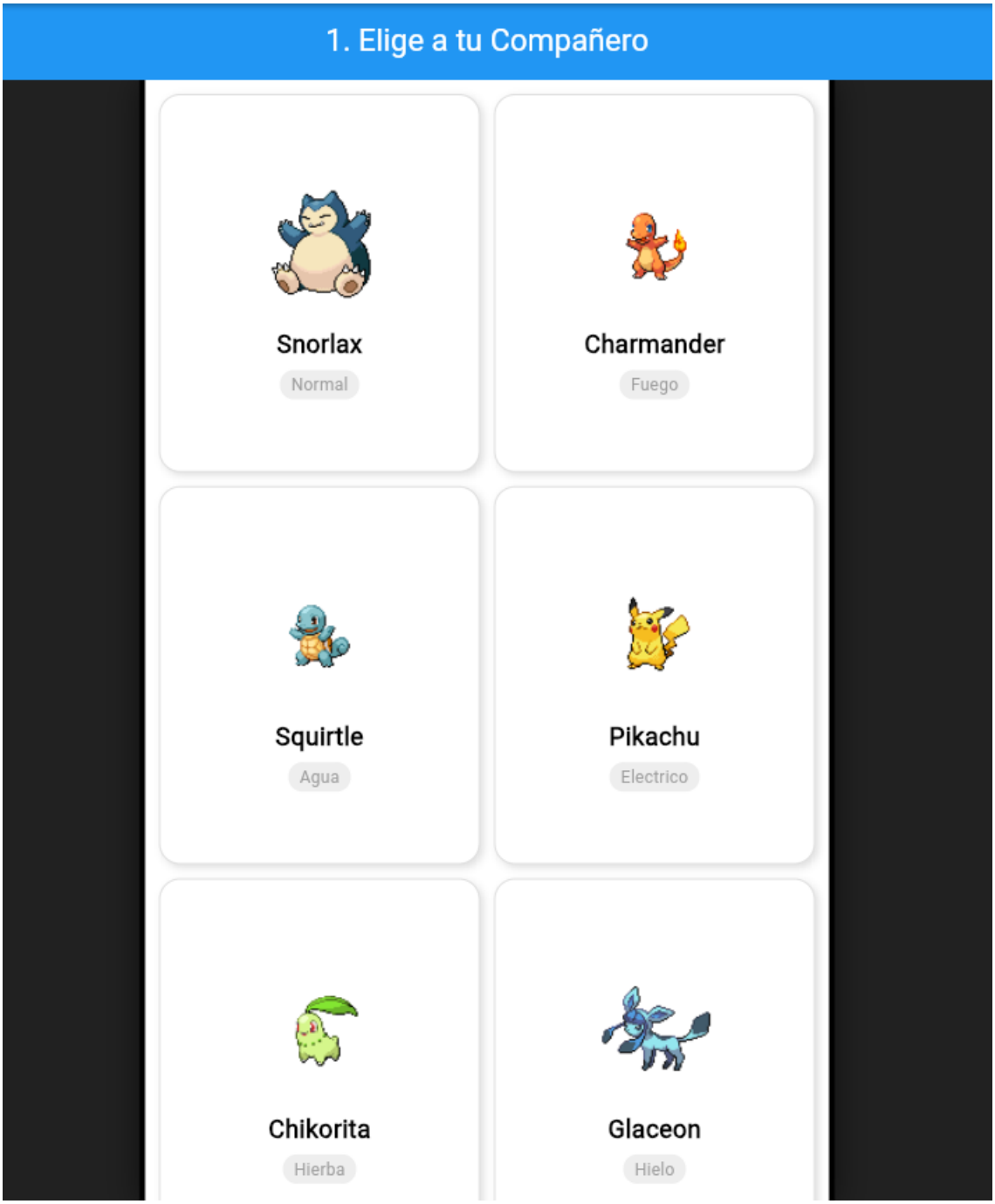
Nuestra aplicación Pokémon se ejecuta sin problemas y es funcional, de la forma es esperada.

Cumple las cararcteristicas requeridas para la entrega, solo se realizó un punto de la caracteristicas de punto extra (objeto para recuperar vida). Existe la base para la implementación de estados de pokemon pero se deja para posibles futuras actualizaciones.

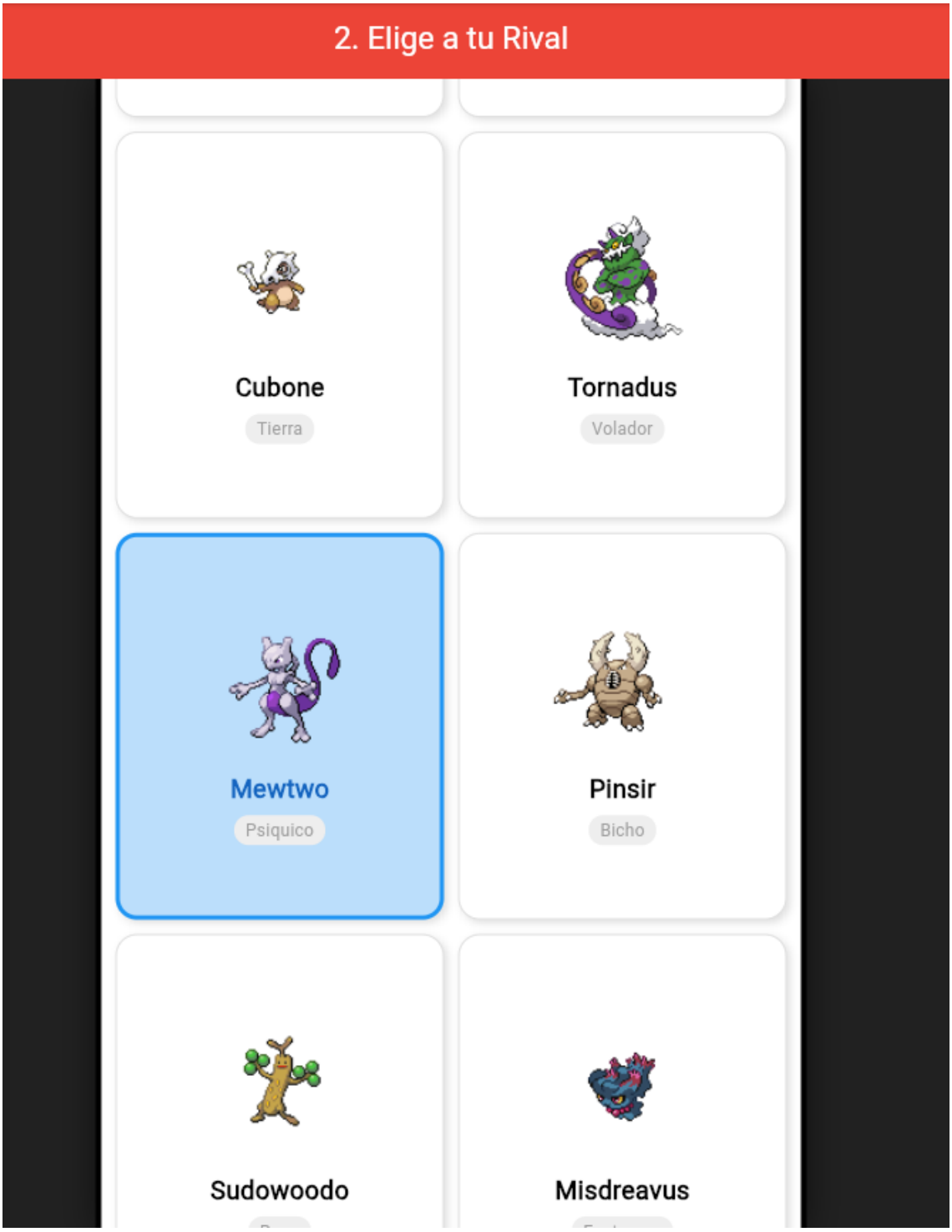
Algunos de los objetivos tecnicos logrados son:
- Ciclo de Combate y Velocidad: Se implementó exitosamente el sistema de turnos dinámico.
- Lógica de Daño y Tipos: Se integró la matriz de efectividad completa usando Map para calcular multiplicadores
- Sistema de Inventario (Puntos Extra): Se cumplió con el requerimiento de objetos recuperadores con posibilidad de expansion
- Interfaz Gráfica (UI): Se consiguió una representación visual fiel a los juegos clásicos.

Presentamos imagenes de la ejecución del programa:

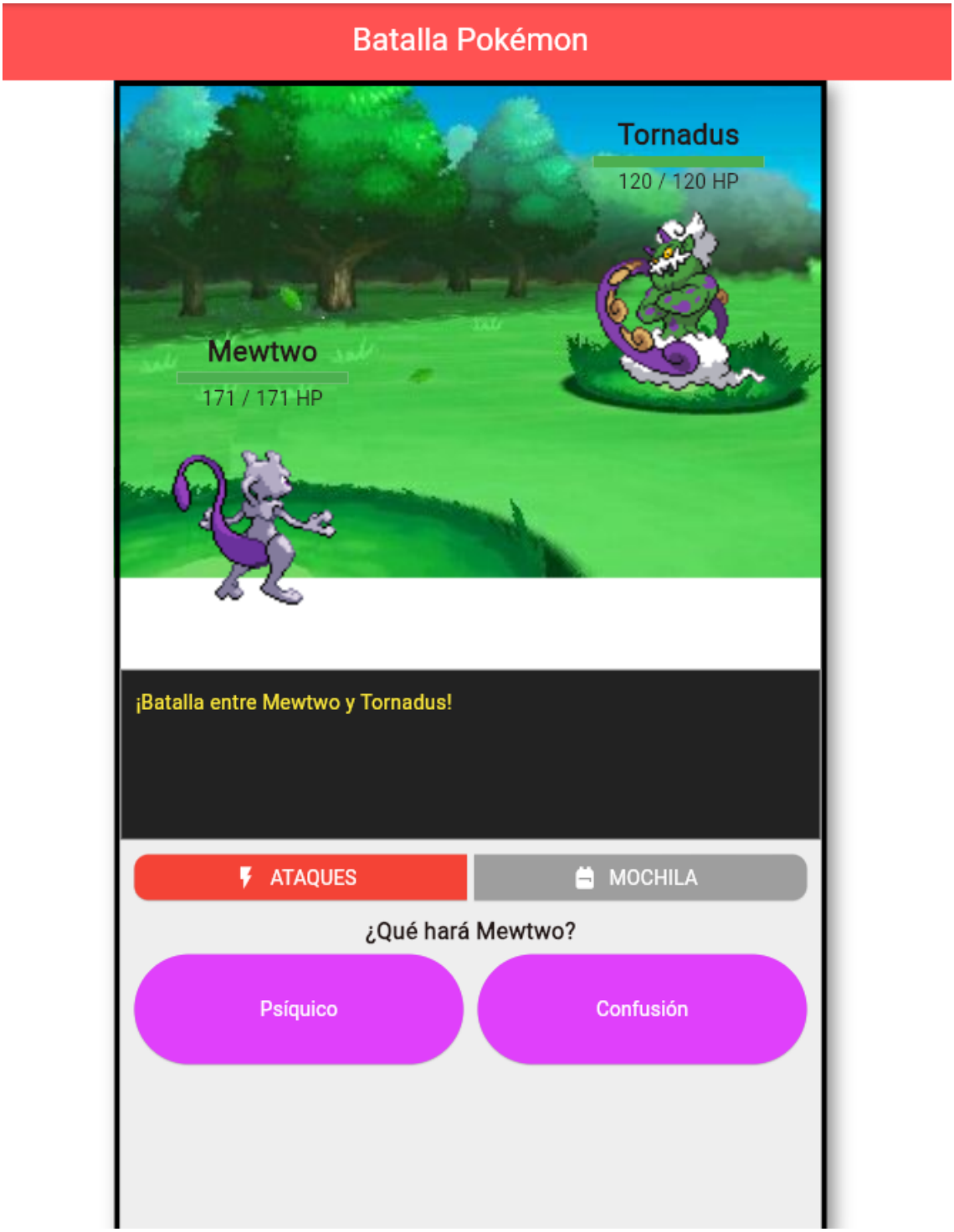
Elegir tu pokemon:



Elegir el pokemon rival:



Menu de pelea:



Adjuntamos link del video de la ejecución del programa:
https://www.youtube.com/watch?v=LYLselLbb1E

6. Conclusiones

Con el proyecto demostramos la aplicación práctica de la abstracción y la herencia para construir un sistema modular. Al definir la clase abstracta Empleado fue fundamental, ya que estableció el método ingresos()) que las clases concretas, EmpleadoAsalariado y EmpleadoPorComision, debían implementar. Esta estructura nos permitio usar polimorfismo, dejandonos que en la clase MainApp tratáramos los distintos tipos de empleados de forma genérica, pero invocando la lógica de cálculo específica de cada uno, como se comprueba en los resultados.

Adicionalmente, el encapsulamiento demostró ser crucial para garantizar la integridad de los datos. Al implementar validaciones directamente en los métodos set —como asegurar que el salario no fuera negativo y que la tarifa de comisión se mantuviera en un rango de 0.0 a 1.0 — se protegió la lógica de negocio de la aplicación. La ejecución y la precisión de los cálculos demuestran que la aplicación funciona y aplica estos principios orientados a objetos que hemos visto a lo largo del curso.

7. Referencias bibliográficas

[1] IONOS Digital Guide, "Qué es Flutter," 2020. Disponible en: https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/que-es-flutter/.

[2] [4] Redwerk, "Programación asíncrona en Flutter," 2024. Disponible en: https://redwerk.es/blog/programacion-asincrona-en-flutter/.

[3] Dart.dev, "Exceptions — Dart Language Tour," 2024. Disponible en: https://dart.dev/language/error-handling.

[4] EsDocu, "Async / Await en Dart," 2024. Disponible en: https://esdocu.com/dart/tutoriales-y-codelabs/codelabs/async-await/.

[5] Flutter Documentation, "App Architecture — Design Patterns," 2025. Disponible en: https://docs.flutter.dev/app-architecture/design-patterns.

[6] Appventurez, "Introduction to MVVM Architecture in Flutter," 2025. Disponible en: https://www.appventurez.com/blog/introduction-mvvm-architecture-flutter.

[7] Medium, "Understanding MVC Architecture in Flutter," 2023. Disponible en: https://medium.com/@Faiz_Rhm/understanding-mvc-architecture-in-flutter-a-comprehensive-guide-with-examples-5d1a372c7eaf.

[8] WikiDex. (s.f.). En WikiDex. Recuperado el 1 de diciembre de 2025, de https://www.wikidex.net/wiki/WikiDex

[9] “Introducción al paradigma de la programación orientada a objetos”, FUOC, 2024. Disponible en: https://openaccess.uoc.edu/bitstream/10609/149901/2/IntroduccionAlParadigmaDeLaProgramacionOrientadaAObjetos.pdf

[10] “Programación Orientada a Objetos – Temario y apuntes”, ESIME-IPN / UTM (compendio de conceptos: herencia, polimorfismo, encapsulación, estructuras). Disponible en: https://fic.uat.edu.mx/resources/pdfs/ETI/poo.pdf