



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor(a):* René Adrián Dávila Pérez

*Asignatura:* Programación Orientada a Objetos

*Grupo:* 7

*Proyecto:* Proyecto 3

*Integrante(s):* 320065570  
425133840  
423020252  
322229916  
425032224

*No. de lista o brigada:* 3

*Semestre:* 2026-1

*Fecha de entrega:* 1/12/2025

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Marco Teórico</b>	<b>3</b>
2.1. Flutter . . . . .	3
2.2. Dart . . . . .	3
2.3. Programación Orientada a Objetos . . . . .	3
2.4. Estructuras de Datos . . . . .	4
2.5. Excepciones . . . . .	4
2.6. Funciones y Logicas del sistema . . . . .	4
2.7. Conceptos Teóricos de Sistemas de Batalla . . . . .	5
2.8. Conceptos Teóricos de Ítems y Recursos . . . . .	6
2.9. Portabilidad de Software . . . . .	6
<b>3. Desarrollo</b>	<b>7</b>
3.1. Desarrollo de la aplicación . . . . .	7
3.2. Portabilidad a Windows . . . . .	13
<b>4. Resultados</b>	<b>14</b>
<b>5. Conclusiones</b>	<b>22</b>
<b>6. Datos de trabajo</b>	<b>24</b>

# 1. Introducción

En este proyecto del tercer parcial tenemos que hacer un programa en flutter usando lenguaje dart que simule una batalla Pokémon y luego debemos portarlo a Windows, Para realizarlo debemos integrar todos los temas vistos en el semestre, por lo que a lo largo de este reporte explicaremos cuales fueron las series de pasos a seguir para lograr que nuestro programa fuera exitoso y se presentarán los resultados de una batalla Pokémon en nuestro programa.

Creemos que es necesario resolver este proyecto porque debemos demostrarnos a nosotros mismos qué entendimos los conceptos teóricos de los temas vistos en clase, que no solo nos quedamos con lo teórico y que sabemos ponerlo en práctica y por último pero menos importantes, que tenemos habilidades sociales para saber trabajar en equipo.

Nuestro objetivo de darle solución a este problema es haber tenido un acercamiento más profundo profesionalmente en el área de backend y de frontend de la manera en la que se puede ver en el ámbito profesional para resolver problemas reales en la programación a través de estas áreas.

## 2. Marco Teórico

### 2.1. Flutter

Para empezar la explicación de todo el marco teórico, es fundamental explicar la base del proyecto, aunque sea algo conocido, desarrollaremos el proyecto en flutter. Flutter es un framework de código abierto desarrollado por Google para crear aplicaciones nativas, compiladas a partir de una única base de código, para móviles, web y escritorio. Este mismo se basa en el lenguaje de programación Dart. [6]

### 2.2. Dart

Dart es un lenguaje de programación moderno y multipropósito, desarrollado por Google, que permite crear aplicaciones rápidas y eficientes para móviles, web, escritorio y servidores. Es de código abierto y se utiliza principalmente en el desarrollo de aplicaciones multiplataforma a través del framework Flutter, nos sirve bastante en el proyecto para la estructuración de la lógica de nuestra aplicación. [5]

### 2.3. Programación Orientada a Objetos

Seria algo repetitivo explicar las bases de lo que hemos visto en todo el curso, sin embargo, para que no quede nada sin comentar, pondremos lo que obviamente usaremos. [3]

- **Clases:** Una clase es una plantilla que define atributos y comportamientos comunes, mientras que un objeto es una instancia concreta de una clase. [3]
- **Encapsulamiento:** El encapsulamiento que hemos estado usando previamente, nos permite proteger los atributos de una clase mediante el uso de modificadores de acceso y métodos controlados. En el proyecto, los atributos del Pokémon los declaramos como privados y solo pueden ser modificados mediante *getters* y *setters*. [blasco]

- **Herencia:** La herencia permite que una clase derive de otra y reutilice sus atributos y métodos. En este sistema desarrollado, todas las clases que representan a un Pokémon de tipo específico heredan de la clase abstracta Pokémon, lo que nos permite mantener un modelo reutilizable. [3]
- **Clases Abstractas:** Las clases abstractas son estructuras que sirven como plantilla para otras clases. No pueden ser instanciadas directamente; en lugar de eso, definen métodos que las clases hijas deben implementar. [3]
- **Polimorfismo:** El polimorfismo es la capacidad de que distintos objetos respondan de manera diferente a un mismo método. Gracias a esto, el programa puede llamar a un método común, mientras que cada clase concreta lo ejecuta según su propia implementación.

[3]

## 2.4. Estructuras de Datos

En nuestro proyecto ocuparemos ciertas estructuras de datos, justo para poder representar de mejor forma información. Y como en el proyecto hay múltiples interacciones entre objetos, ocuparemos HashMaps. [4]

- **Mapas Anidados:** Un mapa anidado es un Map cuyo valor es otro Map, es decir, una estructura jerárquica de pares clave-valor. [4]

## 2.5. Excepciones

El manejo de excepciones es un mecanismo que permite controlar errores durante la ejecución del programa. La utilizamos en validaciones para evitar comportamientos inesperados. [3]

## 2.6. Funciones y Logicas del sistema

- **Funciones de Validación:** Las funciones de validación verifican que los datos cumplan con criterios específicos antes de ser utilizados.

- **Funciones de Visualización de Información:** Son funciones cuyo propósito es mostrar los atributos relevantes de un objeto de manera organizada.
- **Funciones de Efectividad de Ataques:** Una función que calcula la interacción entre dos categorías se utiliza para resolver multiplicadores o resultados basados en tablas de correspondencia.
- **Funciones Globales:** Las funciones globales son accesibles desde cualquier módulo del proyecto y permiten realizar tareas que no dependen del estado interno de un objeto. En el proyecto nos interesan

## 2.7. Conceptos Teóricos de Sistemas de Batalla

Si alguna vez hemos tenido la oportunidad de jugar algún videojuego, pues es posible que estos conceptos ya los conozcamos, pero no está demás explicar ciertos conceptos teóricos de lo que tendrá el programa.

- **Puntos de Vida:** Los puntos de vida representan la vitalidad de un personaje. Cuando descienden a cero, la entidad queda fuera de combate. Si bien es algo que mucha gente conoce por todos los videojuegos que alguna vez hemos jugado, no está demás explicarlo y que veremos visible los puntos de vida como "HP".
- **Estados Alterados:** Condiciones que afectan el rendimiento de un personaje en un sistema de combate.
- **Mecánica de Daño y Límites:** La operación `clamp()` restringe un valor entre un mínimo y un máximo. Esto como en cualquier juego nos asegura que los puntos de vida no sean negativos y también al contrario, que la curación sobrepase el límite.
- **Ataque y Recepción de Daño:** Cuando un ataque hace efecto, tiene que sufrir cierto daño. Ocuparemos funciones para representar esto mismo.

## 2.8. Conceptos Teóricos de Ítems y Recursos

- **Recursos Consumibles:** Los ítems consumibles siguen un modelo donde cada uso reduce su cantidad disponible, pero como explicamos antes, pueden ayudar al personaje.
- **Ítems de Curación:** Son recursos que restauran parte de los puntos de vida. Suman un valor fijo o porcentual al HP y no pueden superar el máximo permitido.

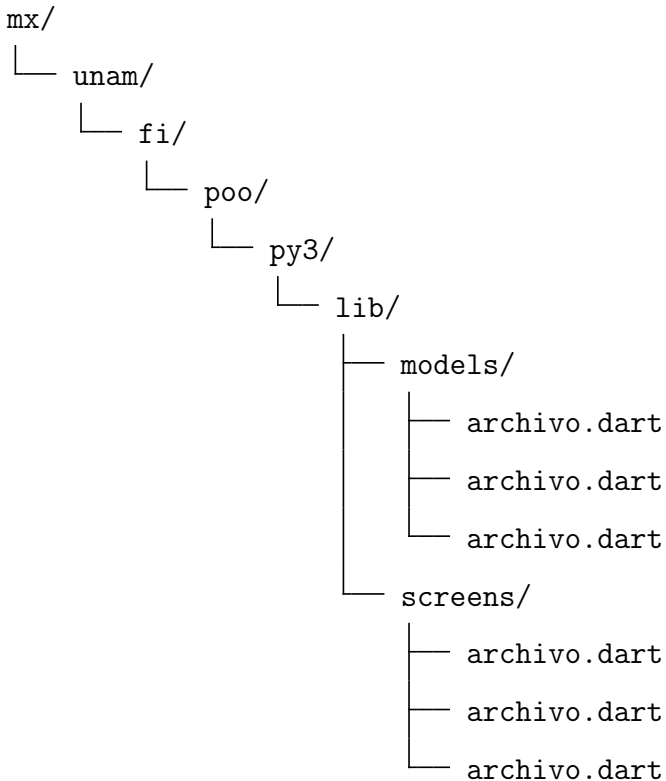
## 2.9. Portabilidad de Software

La portabilidad de software es la capacidad que tiene una aplicación de ejecutarse en diferentes sistemas operativos o entornos con el mínimo esfuerzo de adaptación. Flutter, al ser un framework orientado a la creación de interfaces multiplataforma, proporciona herramientas para generar versiones de una aplicación en sistemas como Android, iOS, Linux, macOS y Windows. Para llevar una aplicación a un sistema operativo específico es necesario realizar un proceso de compilación y configuración del entorno. En el caso de Windows, Flutter produce un ejecutable en formato .exe junto con las bibliotecas necesarias para su ejecución. [2]

## 3. Desarrollo

### 3.1. Desarrollo de la aplicación

Lo primero que hicimos fue decidir cual sería la estructura la estructura general ne nuestro proyecto, y fue la siguiente:



Una vez teniendo la estructura de nuestro proyecto, el siguiente paso fue construir nuestra clase abstracta padre **Pokemon** en la que definimos los atributos privados nombre, ataque, defensa, hp, velocidad, tipo, nombreAtaque y tipoAtaque, posteriormente le dimos valores iniciales a nuestros atributos para poder usar esos valores en nuestro constructor, implementamos métodos de acceso a estos atributos mediante *getters* y *setters*; para los *setters* usamos excepciones para que nuestros atributos no estén vacíos en caso de Strings y en caso de enteros qué tuvieran valores razonables, luego de esto hicimos el método `mostrarInformación()` para visualizar nuestros atributos. A continuación se muestra el diagrama de clases donde hay herencia entre la clase `Pokemon` y los `[TipoPokemon]`:

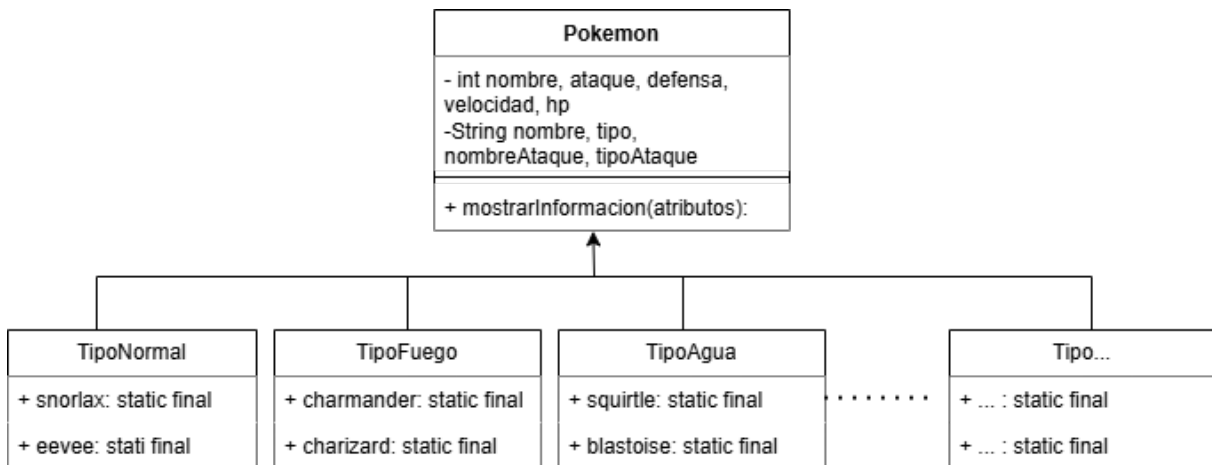


Figura 1: Diagrama de clases UML que muestra la herencia de clases

Dentro de este mismo archivo de **pokemon.dart** creamos una clase de cada tipo de pokémon. Usamos herencia, indicando que la clase **Pokemon** sería nuestra clase padre. En nuestro constructor usamos nuestros atributos de nuestra clase padre y usamos la palabra clave **super** para declarar que queríamos heredar los atributos de nuestra clase **Pokemon**; poder usar estos atributos en nuestras otras clases fue posible gracias a el uso de nuestros métodos de acceso, una vez haciendo esto, nos fue posible crear 2 objetos de cada tipo de Pokémon, por ejemplo, Charmander y Charizard para un Pokémon de tipo fuego. En la creación de nuestros objetos **Pokemon** estáticos finales los creamos dándole los valores a los atributos de nuestro constructor con parámetros. Es importante que en el momento en el que se indicó la herencia, implícitamente nuestras clases derivadas del tipo **Pokemon** ya adquirieron el método `mostrarInformación()`.

El siguiente paso a seguir fue crear nuestra clase **TypeEffectiveness**. Para hacerla nos guiamos de la imagen proporcionada por el profesor en la cual se indicaba cuánto se multiplicaría el ataque a cada pokémon dependiendo del tipo de ataque del defensor. Para ello hicimos un método estático que recibe el tipo de ataque y el tipo del defensor y con `final multipliers = <String, Map<String, double>>` declaramos *multipliers* donde cada clave es un tipo de pokémon y cada valor es otro **Map** que contiene como se relacionan las efectividades con cada tipo. Conforme a la imagen proporcionada nosotros le dimos valores a esas claves, posteriormente hicimos que el multiplicador nos devuelva el valor del **Map** dependiendo del atacante y defensor(2.0, 0.5 ó 0.0) y en los casos donde

no se tenga definido esto se devuelve 1.0

Luego de esto, lo siguiente que se hizo fue definir funciones globales en nuestro archivo llamado **all\_pokemon.dart**; aquí podemos apreciar polimorfismo por qué todos son un distinto Pokémon de la clase `pokemon` pero con distinto comportamiento, en este importamos nuestros archivos de la clase `Pokémon`, de las efectividades y la librería `dart:math`; creamos el método `getAllPokemon()` que nos devuelve una lista de los Pokémon de cada tipo (2 en nuestro caso) y después creamos el método `pickRandomEnemy()` que recibirá la lista de Pokémones y el Pokémon elegido por el usuario y que es el que tomará ciertos criterios para elegir con cual se va a enfrentar el usuario; dentro de este método declaramos una variable para usar `random()`, una variable de tipo `Pokémon` (tipo fuego, roca, etc.) y una variable para contar los intentos para elegir un Pokémon; se elige un Pokémon al azar y mediante un ciclo `if` verificamos si tienen el mismo nombre y si es así, seguimos intentando, una vez elegido uno que no tenga el mismo nombre lo siguiente es consultar con las efectividades, es decir si el rival puede hacer daño al usuario y si el usuario puede hacerle daño al rival para luego verificar que ambos puedan hacerse daño mutuamente; si no es así y se intentan 50 veces, se elegirá como rival a cualquier otro Pokémon diferente al que eligió el usuario.

En la clase `BatallaPokemon` importamos nuestro archivo de la clase `Pokemon` y la librería `dart:math`. Esta clase tiene una variable de tipo `Pokemon`, cuyos atributos `hpActual` (se modifica de acuerdo a los ataques del rival) y `estado` (quemado, paralizado, etc.) inician con el valor “normal”; usamos de nuevo un `Map` con las claves *pocion* y *antidoto* y les guardamos un valor de 2 a ambas. En nuestro constructor con `BatallaPokemon(this.pokemon)` : `hpActual = pokemon.hp` creamos una instancia recibiendo un Pokémon e inicializando su `Hp` actual con su `Hp` máximo y usamos *getters* para tener acceso directo a las propiedades del Pokémon original permitiéndonos acceder a los atributos del Pokémon original como si fueran propias de `BatallaPokemon`. Posteriormente utilizamos un método *getter* que convierte los estados internos en minúsculas a mayúsculas mediante una estructura `switch`; usamos el método `get estaDebilitado()` para determinar cuando un Pokémon está fuera de combate; `get velocidadEfectiva()` para calcular la velocidad dependiendo de los estados – cuando un Pokémon está paralizado reducimos la velocidad a la mitad,

y cuando no lo está, su velocidad es normal. Implementamos el método `puedeAtacar()` de tal manera que si esta congelado no pueda atacar, si esta paralizado tiene 75 % de probabilidad de atacar y si tiene otro estado puede atacar con normalidad; implementamos el método `recibirDanio()` utilizando `clamp()` para definir como límites de Hp 0 como mínimo y Hp máximo como su límite; implementamos el método `curar()` sumando el hp actual más la cantidad usando de igual manera `clamp()` para no excedernos de los límites; el método `curar estado()` para volver a un estado normal cuando se tenga cualquier otro; el método `aplicarEfectoEstado()` usando `floor()` para redondear a un entero para que cuando este quemado o envenenado se le aplique cierto daño al pokémon; el método `puedeUsarPocion()` para poder usarla solo si tenemos al menos una y si nuestro Hp no está completo y el método `puedeUsarAntidoto()` con la condición de tener al menos uno y no estar en estado normal y finalmente los métodos `usarAntidoto()` y `usarPocion()` haciendo que cuando se use un antídoto se vuelva a un estado normal y que cuando se use la poción se sume 40 al Hp y que al usar cualquiera de esos se le reste la cantidad de antídotos o pociones que se tenían inicialmente(2).

Por último, el archivo `music_controller.dart` contiene la clase `MusicController`, la cual define un atributo privado, estático y final, `player`, de tipo `AudioPlayer`. Mediante el anterior, se reproduce la música correspondiente a la pantalla que se muestra – selección ó combate. En el método `changeMusic()` se utiliza tanto para detener la reproducción de una pista y sustituir la por otra, como para iniciar la reproducción de la primer pista cuando comienza la ejecución del programa.

Con todos estos archivos anteriores, ya podíamos hacer la clase que mostraría la interfaz a la hora de escoger un pokémon. Importamos lo necesario para construir nuestra interfaz. Construimos la clase `PokemonSelectionScreen` la cual sería nuestro *widget* y usamos `extends StatefulWidget` para declararlo; se presentó sobrescritura de métodos redefiniendo como queríamos que se comportará el método existente en `StatefulWidget` indicando que el estado pertenece a `PokemonSelectionScreen`. Posteriormente creamos la clase `_PokemonSelectionScreenState extends State<PokemonSelectionScreen>` que es la que controlará la música, la selección y la interfaz en la que `Pokemon? pokemonSeleccionado` será nuestra variable de estado más importante, se redefinieron métodos en los que se ini-

cializó la música, la liberamos y cargamos la lista de pokémones. Creamos la estructura básica de nuestra pantalla y le decimos al usuario qué escoja un Pokémon.

Dividimos la pantalla en 2 paneles, el superior para mostrar la información del pokémon seleccionado, mientras que en el panel inferior se muestran todos los pokémones. En el panel para mostrar la información el borde tendrá un color específico, que depende del pokémon seleccionado y tendremos botones para seleccionar al Pokémon y para quitar este panel, aparecerá el nombre del Pokémon elegido y con **pokemonSeleccionado!.mostrarInformacion()** se ejecutará el método mostrar información de nuestra clase `Pokemon`; aquí también se presenta herencia por qué todas las clases de tipo `Pokemon` responden igual a este llamado. Por otro lado, en el panel inferior dividimos la pantalla en 3 columnas para que en cada una de ellas haya 3 cartas Pokémon, en estas cartas veremos el nombre del Pokémon, la imagen de nuestro Pokémon y abajo las estadísticas de este (tipo, hp, ataque y velocidad). Por último definimos qué color correspondería a cada tipo de Pokémon.

La integración final se realizó en **battle\_screen.dart**, donde implementamos la lógica principal del combate dentro de un `StatefulWidget`. Lo primero que hicimos fue usar el método `initState()` para decidir quién inicia el combate, comparando las velocidades de ambos objetos `BatallaPokemon` y gracias al *getter* de velocidad efectiva que definimos anteriormente, si un pokémon está paralizado su velocidad se reduce y el rival ataca primero. En este mismo método también se realiza el cambio de la música de selección a la música de combate.

Para calcular el daño de los ataques creamos el método `_calculateDamage()`, en este método aplicamos la fórmula matemática usando un nivel fijo de 50, tomando el ataque del agresor y la defensa del defensor; a este resultado lo multiplicamos por el valor de efectividad que obtenemos de nuestra clase `TypeEffectiveness` y finalmente agregamos un factor aleatorio para que el daño varíe ligeramente entre turnos. También implementamos la lógica para aplicar estados alterados dentro de `atacarJugador()`, estableciendo una probabilidad del 15 % para que los ataques de tipo fuego quemen, los de hielo congelen, entre otros.

Para el comportamiento del oponente desarrollamos el método `turnoRival()` junto

con la función de decisión `decidirAcciónRival()`, en esta parte definimos las prioridades claras para la *IA rival*: primero verifica si tiene un estado alterado para usar un antídoto, en segundo lugar revisa si su `hpActual` es bajo para usar una poción de su mochila, y si no necesita curarse, procede a atacar. Todo el flujo de la batalla se controla actualizando la interfaz con `setState()` y verificando en cada turno si algún pokémon se debilitó mediante el método `verificarFinBatalla()`.

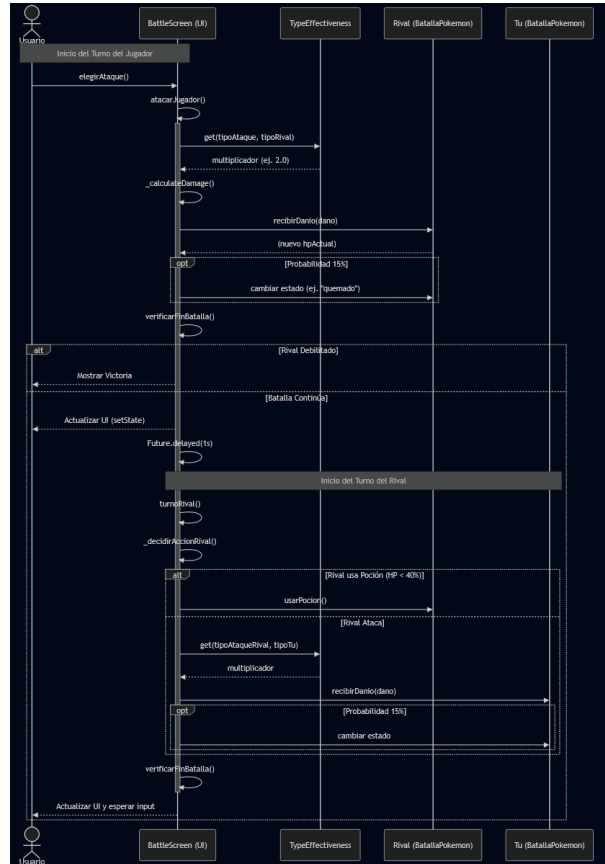


Figura 2: Diagrama de secuencia que modela un turno completo de combate

Dentro del archivo `battle_screen.dart` se encuentra también el método `getImagenPorNombre()`, el cual –de manera similar al método `getColorPorTipo()`– se encarga de proporcionar el nombre de la imagen del pokémon cuyo nombre se ha ingresado a este. El método `getImagenPorNombre()` se utiliza, en conjunto con el elemento `Image.asset`, tanto en el archivo `battle_screen.dart` como en `selection_screen.dart` para mostrar los pokémones involucrados en la interacción con el usuario.

Por último, en nuestro archivo `main.dart` importamos lo necesario para su funciona-

miento, en el título agregamos “BATALLA POKÉMON” e inicializamos nuestro programa.

### 3.2. Portabilidad a Windows

Para concluir con el desarrollo de la aplicación, se buscó la manera de lograr su ejecución en el sistema operativo Windows para así poder distribuirla en equipos que cuenten con este sistema.

Para lograr esto, después de concluir el código y comprobar su correcto funcionamiento ejecutandolo de manera Web a través del editor Visual Studio Code, se realizó una prueba de ejecución en el servicio Windows, a partir del mismo editor, una vez comprobado su correcto funcionamiento se ejecutaron las instrucciones de la documentación de Flutter. [1]

El primer paso realizado, desde el editor Visual Studio Code, consistió en posicionarnos en la carpeta de archivos donde se encontraba nuestro proyecto para ejecutar desde una terminal el comando **flutter upgrade** para actualizar Flutter a la ultima versión disponible, posteriormente, en la misma terminal se ejecutó el comando **flutter config --enable-windos-desktop** para habilitar el desarrollo de las apps de escritorio para Windows en el entorno Flutter; una vez realizados estos dos pasos se procedió a la verificación de la instalación y configuración de todo el entorno necesario para realizar el porteo de la aplicación a través del comando **flutter doctor**. Una vez verificado que todo el entorno se encuentre completamente configurado, se ejecutó el comando **flutter build windows**, este comando hace que Flutter gestione completamente la compilación del proyecto para el sistema Windows y genera una serie de archivos que permiten la ejecución de la aplicación, donde se incluye un archivo ejecutable .exe; estos archivos los genera en una subcarpeta dentro de nuestro proyecto, por lo cual se tuvo que recurrir a ella para ejecutar el archivo .exe y verificar su correcto funcionamiento.

Una vez verificado el correcto funcionamiento del ejecutable, conseguimos el porteo de nuestra aplicación al sistema Windows, lo unico que faltaba era una forma de compartirlo, lo cual se hizo a partir del software Inno Setup Compiler, mencionado en las instrucciones de la documentación de Flutter [7]. Este software fué utilizado siguiendo los pasos presentados en la guía de empaquetado para Windows; a partir de la serie de pasos ahí descrita

se llegó a generar un ejecutable `pokemon_sim_x86.exe`, el cual permite descargar la aplicación desarrollada.

## 4. Resultados

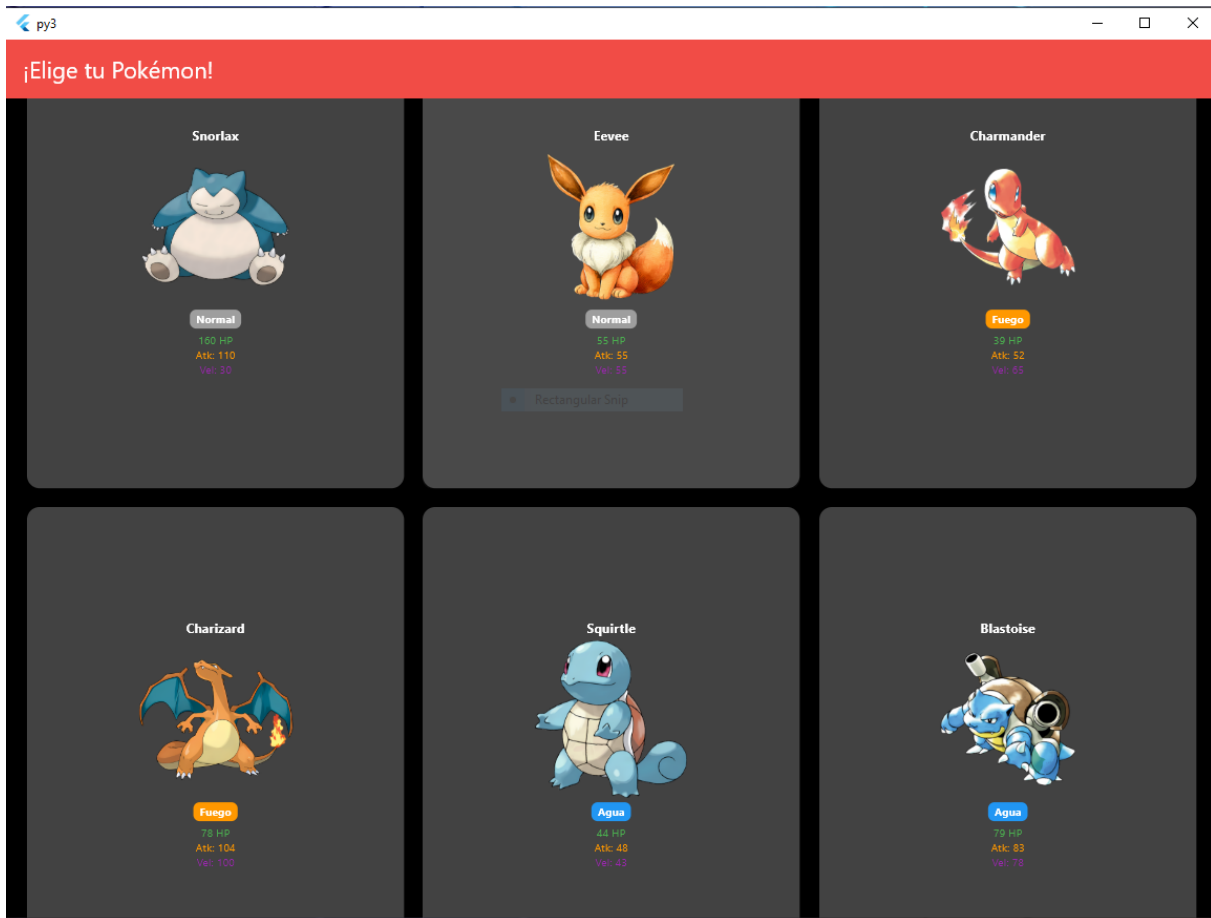


Figura 3: Pantalla de elección de pokémon

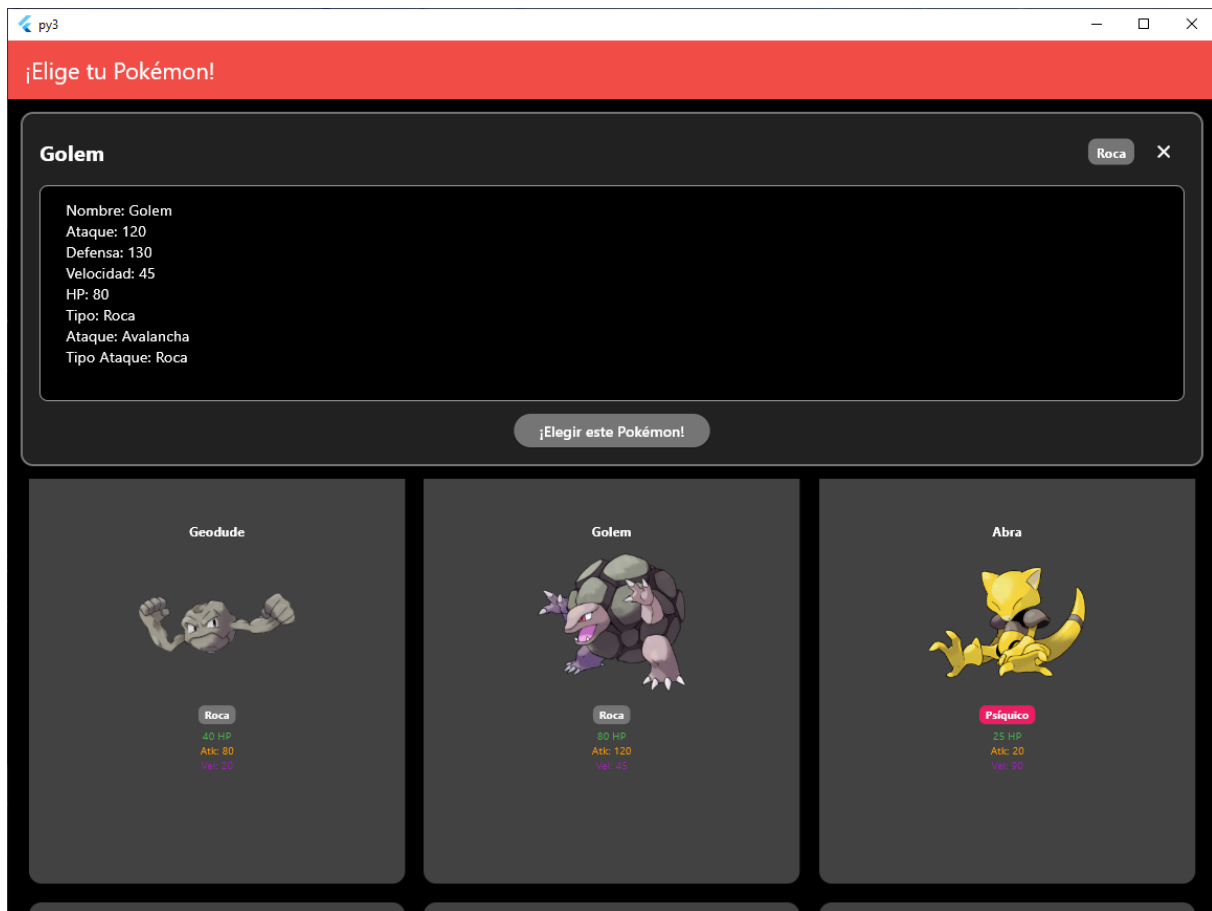


Figura 4: Pantalla de elección de pokémon – pokémon seleccionado

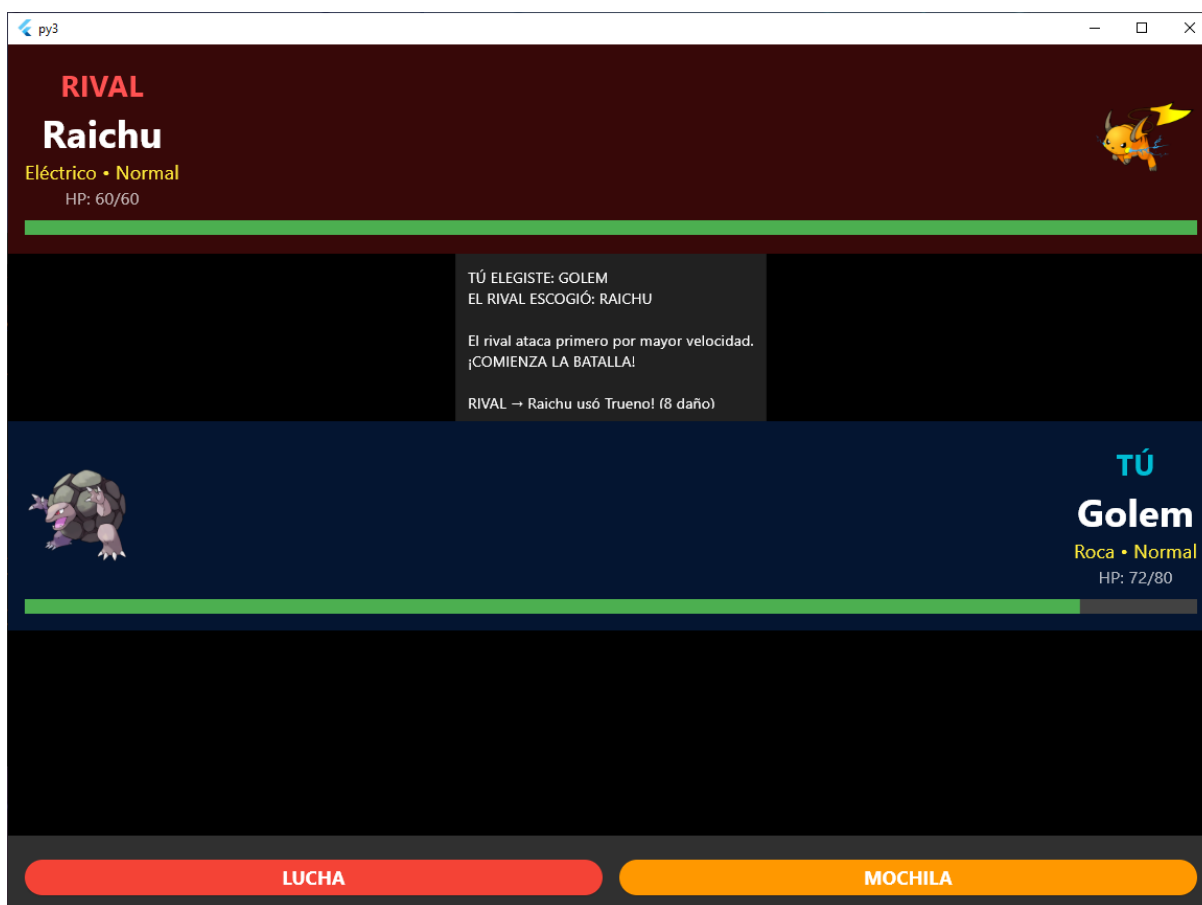


Figura 5: Pantalla de combate

# MOCHILA



Poción (+40 HP) 2 restantes



Antídoto 2 restantes

Figura 6: Funcionamiento de la Mochila

EL RIVAL ESCOGIÓ: DRATINI

Atacas primero por mayor velocidad.

¡COMIENZA LA BATALLA!

TÚ → Swinub usó Rayo Hielo! (13 daño)

RIVAL → Dratini usó Garra Dragón! ¡Es muy efectivo! (34 daño)

Figura 7: Ataque efectivo

EL RIVAL ESCOGIÓ: BULBASAUR

Atacas primero por mayor velocidad.

¡COMIENZA LA BATALLA!

TÚ → Raichu usó Trueno! (19 daño)

RIVAL → Bulbasaur usó Látigo Cepa! No es muy efectivo... (5 daño)

Figura 8: Ataque inefectivo

Atacas primero por mayor velocidad.

¡COMIENZA LA BATALLA!

TÚ → Arbok usó Ácido! No es muy efectivo... (6 daño)

¡Machamp está envenenado!

RIVAL → Machamp usó Antídoto! Estado curado.

Figura 9: Efecto de veneno

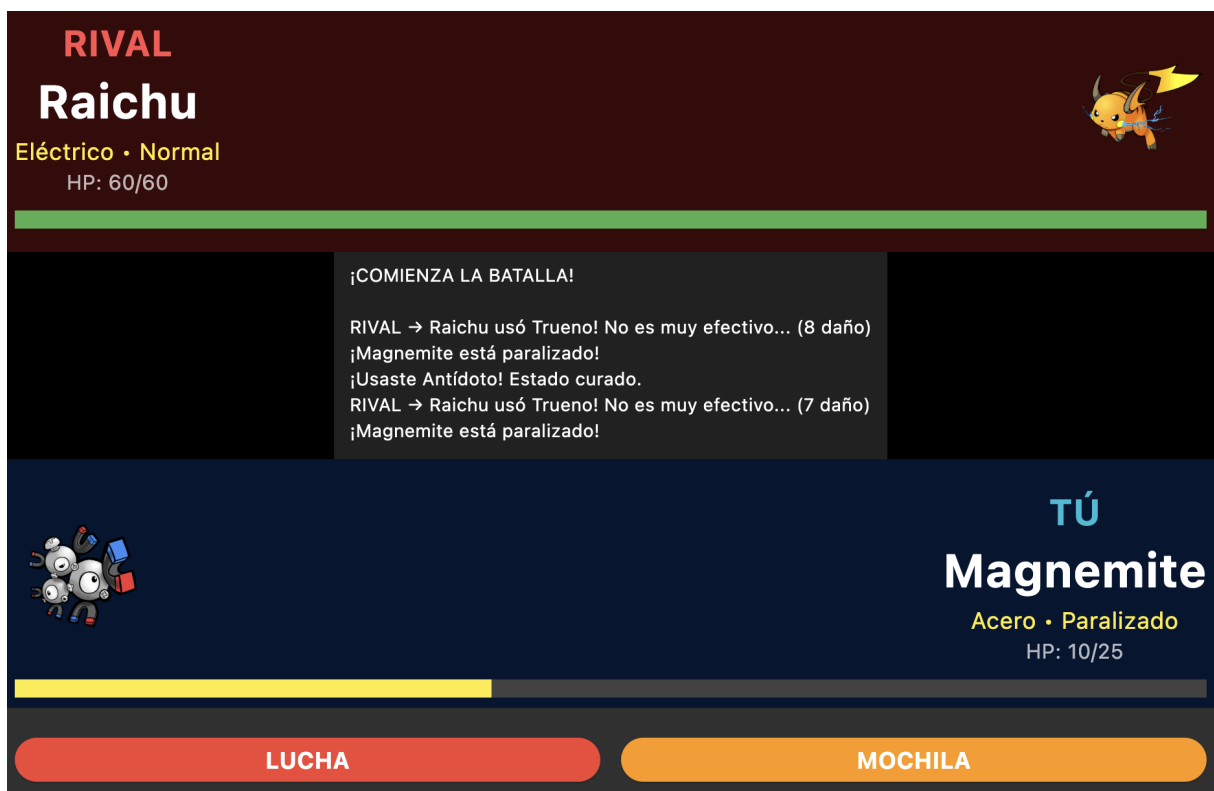


Figura 10: Efecto de parálisis

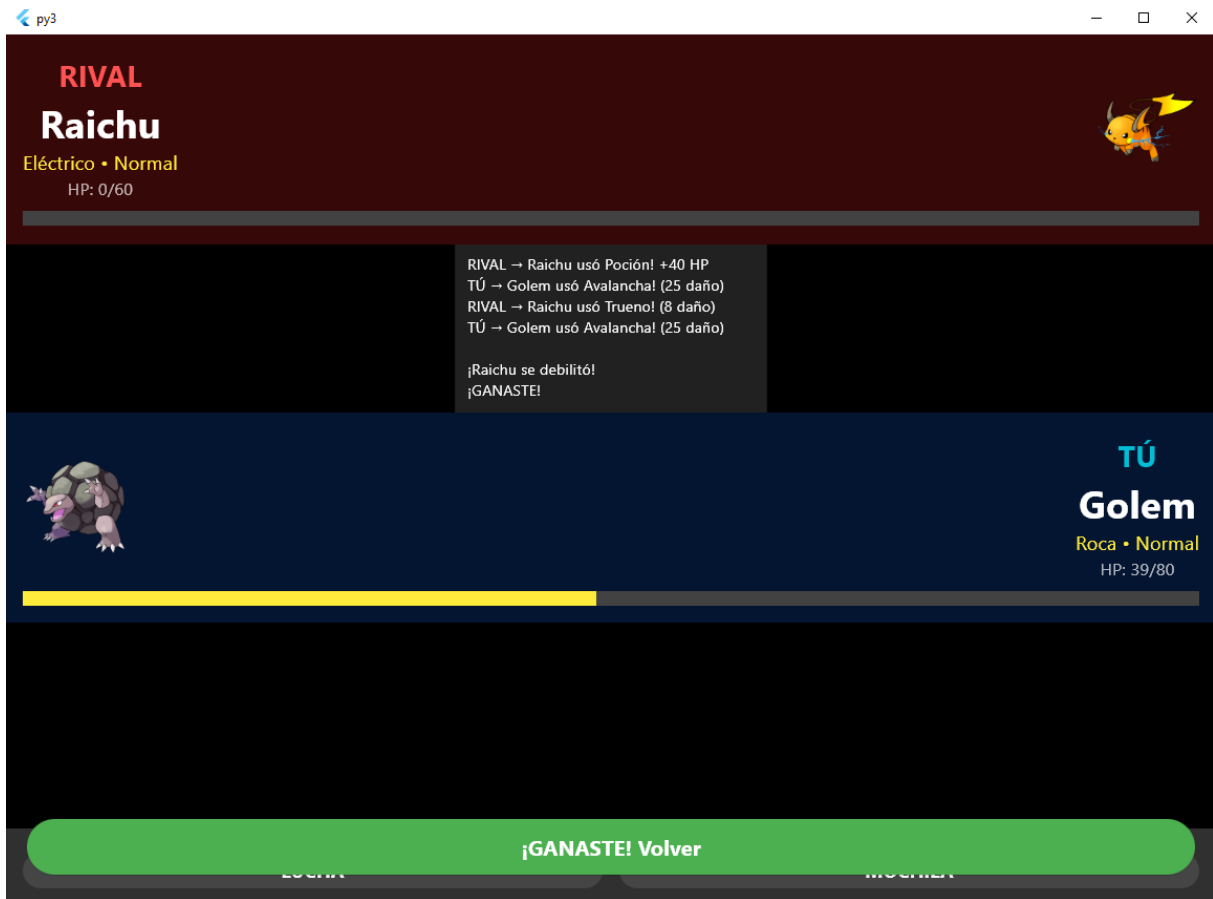


Figura 11: Victoria

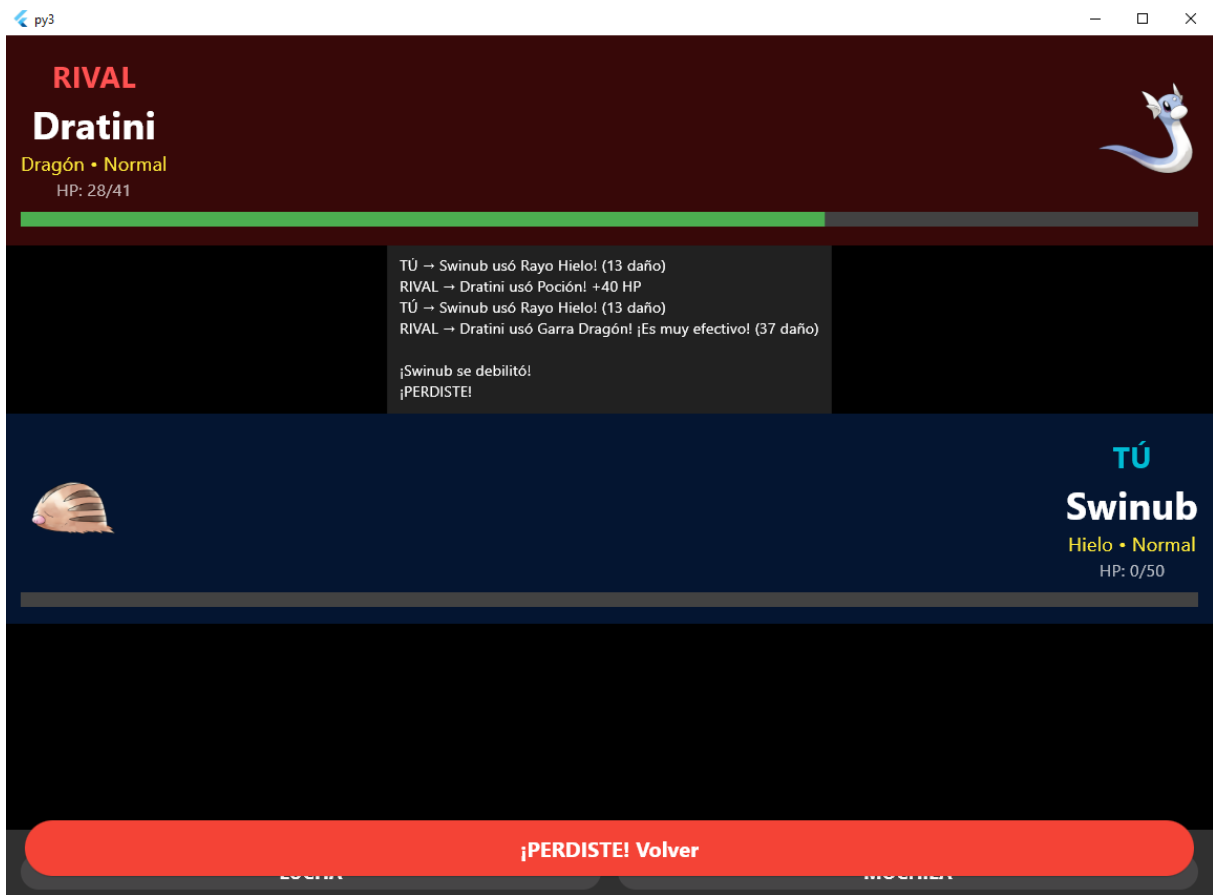


Figura 12: Pérdida

## 5. Conclusiones

La elaboración de este proyecto tan amplio, donde buscamos simular las batallas de un enfrentamiento pokémon, fue algo satisfactorio; observar todos los conocimientos previos en un mismo proyecto no solo permite ver la aplicación de cada tema, si no que demuestra que hemos podido retener los principios de Programación Orientada a Objetos. La implementación se basó en un diseño de clases (clase abstracta Pokemon) y un modelo detallado de las reglas del juego, especialmente en el cálculo de las efectividades de tipos y la lógica avanzada de la clase BatallaPokemon. Esta última encapsuló las condiciones de combate, incluyendo el manejo de estados alterados, la aplicación de límites de HP, y el uso de ítems.

El equipo no solo logró construir la estructura de datos básica, sino que también implementó con éxito reglas de negocio complejas y detalladas necesarias para simular batallas realistas, culminando en una presentación funcional para un combate Pokémon.

## Referencias

- [1] Flutter. *Building Windows apps with Flutter*. URL: [https://docs-flutter-dev.translate.google/platform-integration/windows/building?\\_x\\_tr\\_sl=en&\\_x\\_tr\\_tl=es&\\_x\\_tr\\_hl=es&\\_x\\_tr\\_pto=tc](https://docs-flutter-dev.translate.google/platform-integration/windows/building?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc).
- [2] Flutter. *Flutter*. URL: <https://esflutter.dev/>.
- [3] IBM. *Programación Orientada a Objetos*. 2021. URL: <https://www.ibm.com/docs/es/spss-modeler/saas?topic=language-object-oriented-programming>.
- [4] IONOS. *Java Hashmap: almacenar datos en pares clave-valor*. 2024. URL: <https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/java-hashmap/>.
- [5] M. Martinez. *¿Qué es el lenguaje de programación Dart?* 2021. URL: <https://www.hiberus.com/crecemos-contigo/que-es-el-lenguaje-de-programacion-dart/>.
- [6] D Morales. *Flutter: qué es y por qué facilita el desarrollo multiplataforma*. 2024. URL: <https://www.pragma.co/es/blog/flutter-que-es-y-por-que-facilita-el-desarrollo-multiplataforma>.
- [7] A. Prasad. *Packaging and Distributing Flutter Desktop Apps : The Missing Guide for Open Source Indie Developers — Creating Windows .exe installer [Part 2 of 3]*. 2015. URL: <https://medium.com/@fluttergems/packaging-and-distributing-flutter-desktop-apps-the-missing-guide-part-2-windows-0b468d5e9e70>.

## 6. Datos de trabajo

### **Trabajo en el reporte:**

- Introducción: Colaborada por 425133840
- Marco teórico: Colaborado por 322229916
- Desarrollo y resultados: Colaborado por 425133840, 425032224 , 320065570
- Conclusiones: Colaborado por 423020252, 322229916
- Diagramas: Colaborado por 423020252

### **Usuarios en github y número de cuenta:**

thborthy: 425133840

SantiPichardo: 322229916

GT-8: 425032224

CasvelGit: 423020252

eduarAG:320065570