	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 práctica(s): 11, 12 y 13

Integrante(s):
322276824
322258516
425037384
320108116
322221415

No. de brigada: 01

Semestre: 2026-1

Fecha de entrega: 28 de noviembre de 2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

- **Planteamiento del problema:**
  Analizar el código proporcionado para identificar y explicar la aplicación práctica de los temas de Archivos, Hilos y Patrones de Diseño en los diferentes ejemplos. Además de documentar la arquitectura estática y dinámica de los programas mediante Diagramas UML y dar la interpretación de los conceptos teóricos en el contexto del código.

- **Motivación:**
  La combinación de la persistencia de datos (Archivos), la gestión de tareas concurrentes (Hilos) y las soluciones arquitectónicas comprobadas (Patrones de Diseño) es fundamental en el desarrollo de software complejo. Esta práctica permite integrar conocimientos avanzados de POO, fortaleciendo la capacidad de construir aplicaciones robustas, eficientes y sostenibles, que pueden gestionar recursos y ejecutar tareas de manera óptima.

- **Objetivos:**
  Aplicar los conceptos de manejo de archivos para la persistencia de información y ejecutar procesos concurrentes mediante hilos para mejorar la eficiencia del sistema, así como identificar y modelar los patrones de diseño implementados en el código proporcionado. Además, reforzar la habilidad de documentar la arquitectura del software mediante la creación de Diagramas UML estáticos y dinámicos.

2. Marco Teórico

2.1. Archivos

En Dart, la manipulación del sistema de archivos se gestiona principalmente a través de la librería `dart:io`. Esta librería provee clases como `File` y `Directory` para interactuar con el sistema operativo.

2.2. Asincronía en I/O

Un concepto crítico en Dart es que las operaciones de Entrada/Salida (I/O) deben ser, preferentemente, **no bloqueantes**. Dado que Dart es *single-threaded* en su ejecución principal, leer un archivo grande de manera síncrona congelaría la interfaz de usuario.

- **Síncrono:** `file.readAsStringSync()` (Bloquea el hilo).
- **Asíncrono:** `file.readAsString()` (Retorna un `Future`).

El uso de las palabras reservadas `async` y `await` permite escribir código asíncrono que se lee como si fuera síncrono, facilitando el manejo de excepciones y el flujo lógico.

3. Hilos y Concurrencia (Isolates)

A diferencia de Java, donde se crean múltiples *Threads* que comparten el mismo espacio de memoria, Dart utiliza un modelo de aislamiento basado en el **Event Loop** y los **Isolates**.

3.1. El Event Loop vs. Threads

Dart ejecuta el código en un único hilo principal impulsado por un bucle de eventos (*Event Loop*). Para tareas concurrentes ligeras (como esperar una respuesta HTTP o leer un archivo), no se crean nuevos hilos del sistema operativo; simplemente se programan en el Event Loop.

3.2. Isolates

Cuando se requiere procesamiento pesado (CPU-bound) que podría bloquear el Event Loop, Dart utiliza **Isolates**.

- Cada Isolate tiene su propio espacio de memoria (Heap).
- **No hay memoria compartida:** Esto elimina la necesidad de *locks* complejos y evita condiciones de carrera (*race conditions*).
- La comunicación entre el hilo principal y un Isolate se realiza mediante paso de mensajes (*message passing*) a través de puertos (*Ports*).

4. Patrones de Diseño en Dart

La sintaxis moderna de Dart permite implementar los patrones del *Gang of Four* (GoF) de manera más concisa, e incluso algunos están integrados en el lenguaje.

4.1. Patrón Singleton

El patrón Singleton asegura que una clase tenga una única instancia. En Dart, esto se logra elegantemente utilizando un constructor privado y un *factory constructor*.

- Instancia estática privada
- Constructor privado
- Factory constructor retorna la instancia existente

4.2. Patrón Observer (Streams)

Mientras que en otros lenguajes se implementan interfaces `Observer` y `Subject` manualmente, Dart incorpora este patrón de forma nativa a través de los **Streams**.

- **StreamController:** Actúa como el sujeto que emite eventos.
- **StreamSubscription:** Actúa como el observador que escucha los cambios.

Este mecanismo es la base de la arquitectura reactiva (BLoC, Provider) utilizada comúnmente en Flutter.

5. Desarrollo

5.1. Código del tema Archivos

La aplicación desarrollada permite la creación, lectura y sobreescritura de archivos tipo `.txt` implementando los conceptos de manejo de errores y archivos, pues el programa implementa **dart.io** para la entrada/ salida de datos y basa su funcionamiento en el flujo de datos que se produce entre el porograma y la fuente. Explicando más a fondo la aplicación:

5.1.1. main()

Es la clase principal en la que se imprime un menú con las e opciones disponibles para trabajar con archivos de texto. Cada opción a excepción de la de salir en la que solo se muestra un mensaje, llama un función de tipo privada para realizar la operación en cuestión, ofreciendo así un código más organizado.

5.1.2. Funciones:

crearYEscribirArchivo()

La función tiene el propósito de crear un archivo de tipo `.txt` y permitirle al usuario ingresar el contenido, donde el contenido admitido solo son Strings. Una vez que se confirma que el nombre del archvio no es `NULL` o vacío, utiliza una lista de Strings para agregar las lineas escritas y termina una vez que detecta que se escribión `FIN`.

Las líneas escritas en la lista son agregadas a un archivo **FILE** donde se implementa manejo de errores al poner el bloque de código dentro de un try/catch para cubrir el caso en el que ocurra un error al guardar el archivo.

leerArchivoExistente()

La función tiene el propósito de leer el contenido de un archivo que se busca a través de una ruta proporcionada por el usuario, una vez que se comprueba que la

ruta es válida se implementa un `try` para comprobar la existencia del archivo con `existsSync()`. Si el archivo existe, el contenido se lee como cadena y se imrpime en consola.

sobrescribirArchvio()

La función tiene el propósito de buscar un archivo a través de una ruta proporcionada por el usuario, donde se verifica la validez de la ruta y la existencia del archivo que se está buscando. Se lanza un mensajde de confirmación para la sobreescritura del archivo, en caso de ingresar `SI` se continua con la sobreescritura, guardando las nuevas líneas de texto en la lista `nuevasLineas[]`, cuando se detecte la palabra `FIN` se implementa un `try/catch` para ingresar el nuevo texto al archivo, mandando un mensaje de error en caso de que falle la operación,

5.2. Ejemplos del tema Hilos

5.2.1. Ejemplo 1:

El primer ejemplo es de concurrencua básica donde se utilizan **Future** para marcar una tarea que se completará a futuro sin interfenir con el hilo principal y **async** para marcar que la tarea es asíncrona. Así la tarea Inicio se completa y espera dos segundos a que la tarea `Future.deloyed` terminé su ejecución, imprimiendo los mensajes en consola como si hubiera sido una sola tarea que se ejecutó en secuencia.

5.2.2. EJemplo 2:

Se usa **Future** y **async** para señalar que la función es asíncrona y que puede ejecutar tareas que requieran tiempo de espera además de la opción de utilizar **await** trabajando con la concurrencia. Así funciona imprimiendo un mensaje inicial, pero el `await` señala que el programa podrá continuar una vez que espere el resultado de `Future.deloyed`, finalizando en un tiempo de 2 segundos.

5.2.3. Ejemplo 3:

El tercer ejemplo muestra un caso básico de uso de **Isolate** usando la comunicación por mensajes. Así se crea una función `tarea` donde se define el método `SendPort` utilizado para mandar mensajes al isolate principal. Por otro lado el isolate principal define el método escucha `receivePort` para recibir el mensaje y usa `async` para usar `await` haciendo que el programa espere a que se cree un nuevo isolate donde se ejecute el método tarea indicando con `receivePort.sendPort` lo que va a recibir, extrayendo el mensaje con el método escucha `receivePort.listen()` y por último imprimiendo el mensaje en pantalla y cerrando el **receivePort** ya que no hay necesidad de recibir mensajes.

5.2.4. Ejemplo 4:

El cuarto ejemplo consiste en realizar la suma de cero a cinco millones sin bloquear el hilo principal, implementando los isolates para hacer la operación pesada de manera independiente sin interrumpir el flujo del isolate principal. Así se crea la función `sumaGrande()` que define su método **sendPort**. Por otra parte el isolate principal define su método escucha **receive** y usa **await** para seguir el flujo una vez que se cree el isolate paralelo y ejecute la operación mientras el flujo del isolate principal se prepara para recibir el resultado, imprimiendolo una vez que el hilo paralelo acaba la sumatoria.

5.2.5. Ejemplo 5:

En este ejemplo se implementa una comunicación bidireccional entre dos isolates, uno se encarga de mandar y otro de recibir, para lo cual ambos deben definir sus métodos **receivePort** y **sendPort**. Así la función worker define sus métodos y a través del método escucha `receive.listen()` recibe los mensajes del main y manda una respuesta con `mainPort.send()`. Por otra parte el `main()` crea un isolate donde se ejecuta worker y así recibe su `sendPort()` creando su método escucha y definiendo un if/else para confirmar cuando se establezca la comunicación obteniendo como primer mensaje el `sendPort` y como segundo mensaje la respuesta de `worker`, matando el isolate al finalizar para devolver el control al sistema.

5.2.6. Ejemplo 6:

Este último ejemplo muestra el caso de ejecución del ejemplo 4 donde se realiza la sumatoria, pero ahora realizando todo en un solo hilo mostrando que al final el tiempo de ejecución es despreciable, pero recordando que la utildiad de los hilos está en la eficiencia de la ejecución y la optimización de recursos.

5.3. Patrones de diseño

La implementación se estructuró siguiendo el patrón de arquitectura Modelo-Vista-Controlador, organizando el código en tres capas principales.

5.3.1. Capa de modelo (Clases Pokemon y Ataque)

La clase Pokemon modela las características base de los pokemonen, posee atributos finales como nombre, nivel y tipo, además de atributos calculados dinámicamente como vida y velocidad. En el constructor, se utiliza la clase Random para asignar valores aleatorios a estas estadísticas basándose en el nivel del pokemon. Se implementaron las subclases PokemonFuego y PokemonHierba, las cuales heredan de Pokemon e inicializan el tipo específico mediante la llamada al constructor de la superclase con el método super.

De manera análoga, la clase Ataque define la estructura de los movimientos con atributos para nombre, tipo y potencia. Se extendió esta funcionalidad a través de las clases AtaqueFuego, AtaqueHierba y AtaqueNormal, permitiendo clasificar las acciones para el combate.

5.3.2. Capa de vista (Clases CombateView)

Se definió una clase abstracta CombateView, que establece los métodos necesarios para la interacción con el usuario, tales como mostrarInformacionPokemon, mostrarDanio o mostrarGanador. La implementación concreta se realizó en la clase ConsoleCombateView, la cual define el comportamiento de estos métodos utilizando la función print para desplegar los eventos y resultados del combate directamente en la consola.

5.3.3. Capa de controlador (Clase CombateController)

La clase CombateController es quien gestiona el flujo del programa. Cuenta con un atributo de tipo CombateView para comunicarse con la interfaz. Su lógica central reside en el método iniciarCombate, el cual recibe a los pokemones y gestiona el combate dentro de un ciclo while que se mantiene activo mientras la vida de ambos pokemones sea mayor a cero.

Dentro de este ciclo, se determina el orden de los turnos comparando el atributo velocidad de cada pokemon, en caso de empate, se utiliza Random para decidir aleatoriamente quién ataca primero. El cálculo del daño se realiza con el método privado _atacar, donde se evalúa la efectividad de tipos verificando si el tipo del ataque tiene ventaja contra el tipo del defensor, asignando un multiplicador de 2.0 para ataques "súper efectivos" o de 0.5 para "poco efectivos". Finalmente, se actualiza el estado del defensor y se notifica a la vista.

5.3.4. Método principal (Main)

En la función main se realiza la instanciación e inicialización de los componentes. Se crea primero la vista ConsoleCombateView y se usa en el constructor del CombateController. Posteriormente, se instancian los objetos de tipo PokemonFuego y PokemonHierba con sus nombres y niveles, así como un objeto de tipo Ataque. Finalmente, se invoca el método iniciarCombate del controlador, pasando los objetos creados como argumentos para dar inicio a la ejecución del programa.

Diagrama de clases (UML estático)

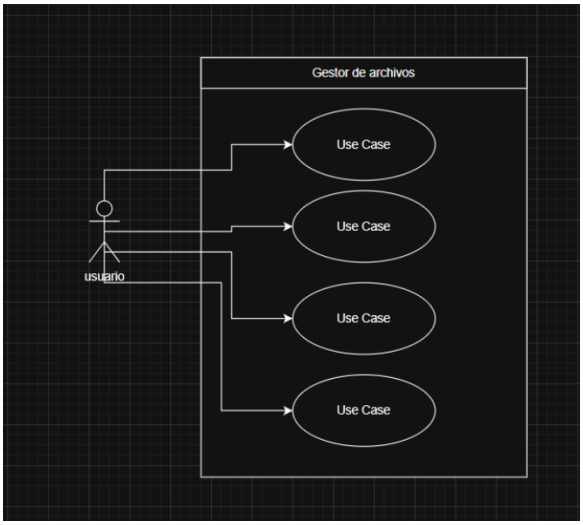


Figura 1: UML:Diagrama de estado del tema 11.

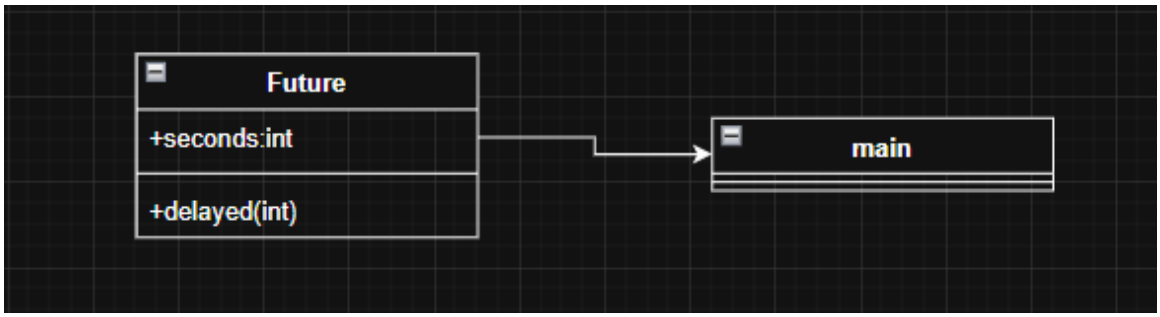


Figura 2: UML:Diagrama de estado del ejemplo 1 y 2 del tema 11.

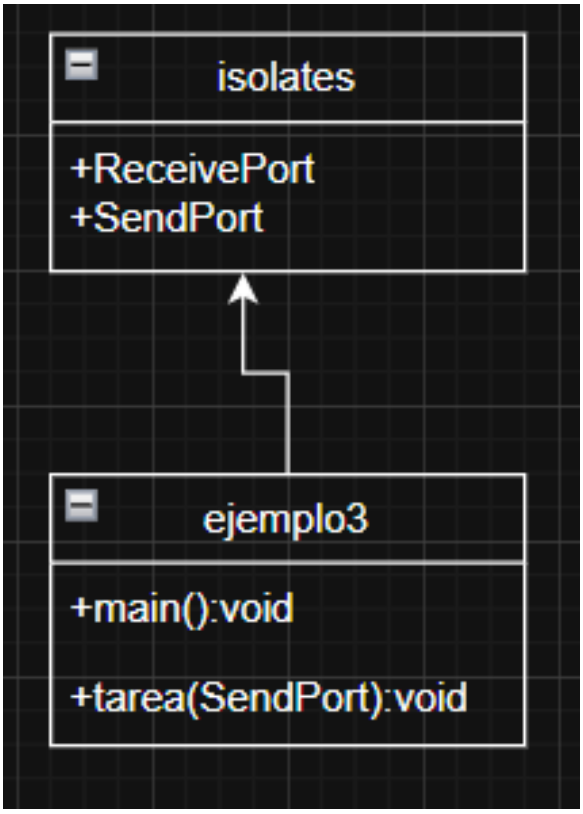


Figura 3: UML:Diagrama de estado del ejemplo 3 del tema 12.

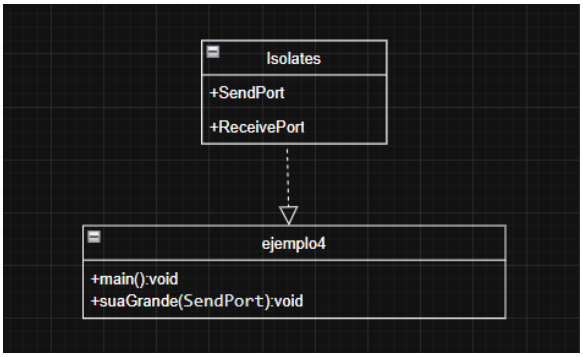


Figura 4: UML:Diagrama de estado del ejemplo 4 del tema 12.

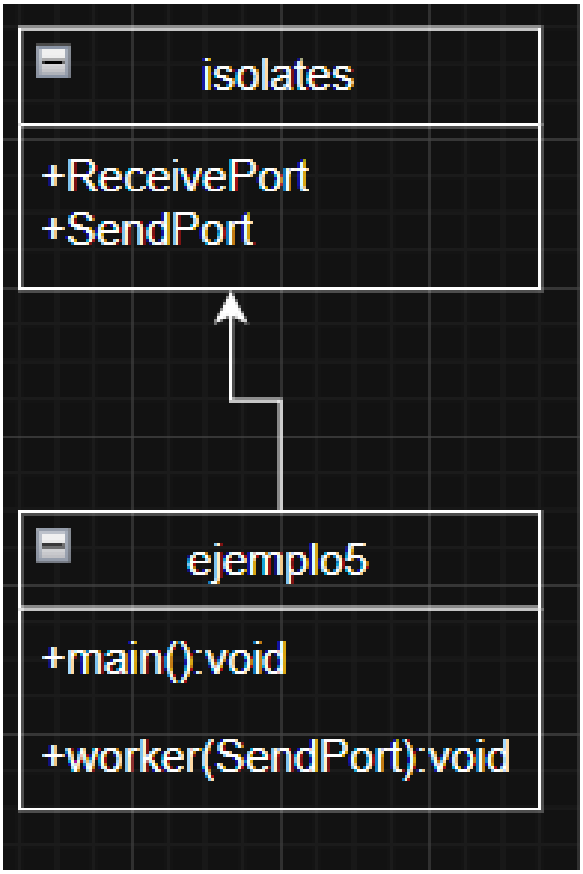


Figura 5: UML:Diagrama de estado del ejemplo 5 del tema 12.

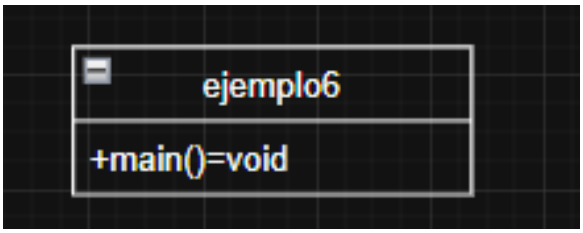


Figura 6: UML:Diagrama de estado del ejemplo 6 del tema 12.

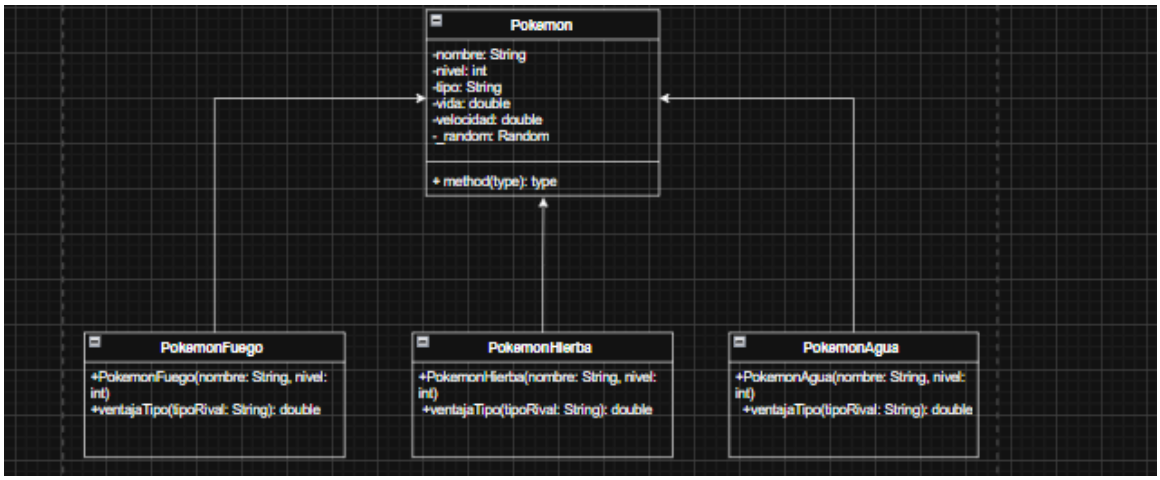


Figura 7: UML:Diagrama de estado de pokemon tema 13.

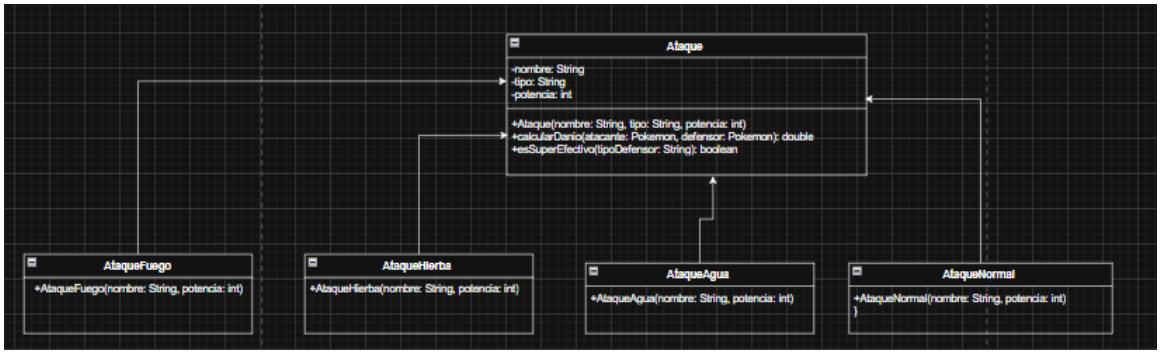


Figura 8: UML:Diagrama de estado de ataques tema 13.

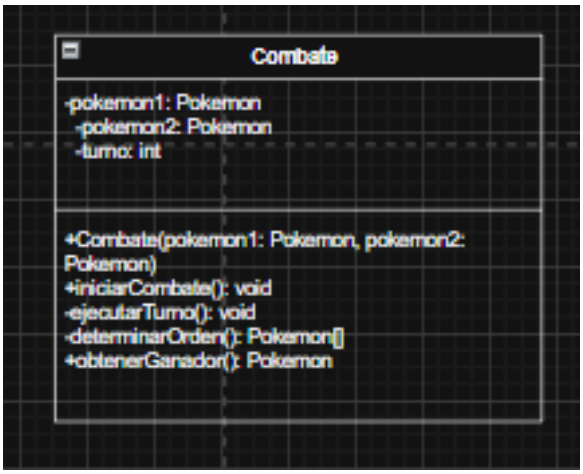


Figura 9: UML:Diagrama de estado de comabte tema 13.

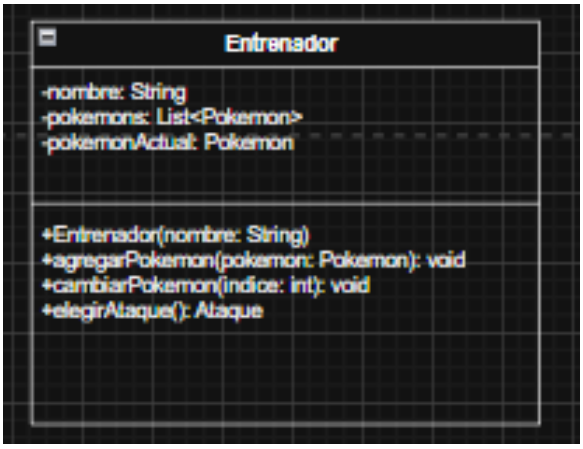


Figura 10: UML:Diagrama de estado entrenador tema 13.

Diagrama de secuencia (UML dinámico)

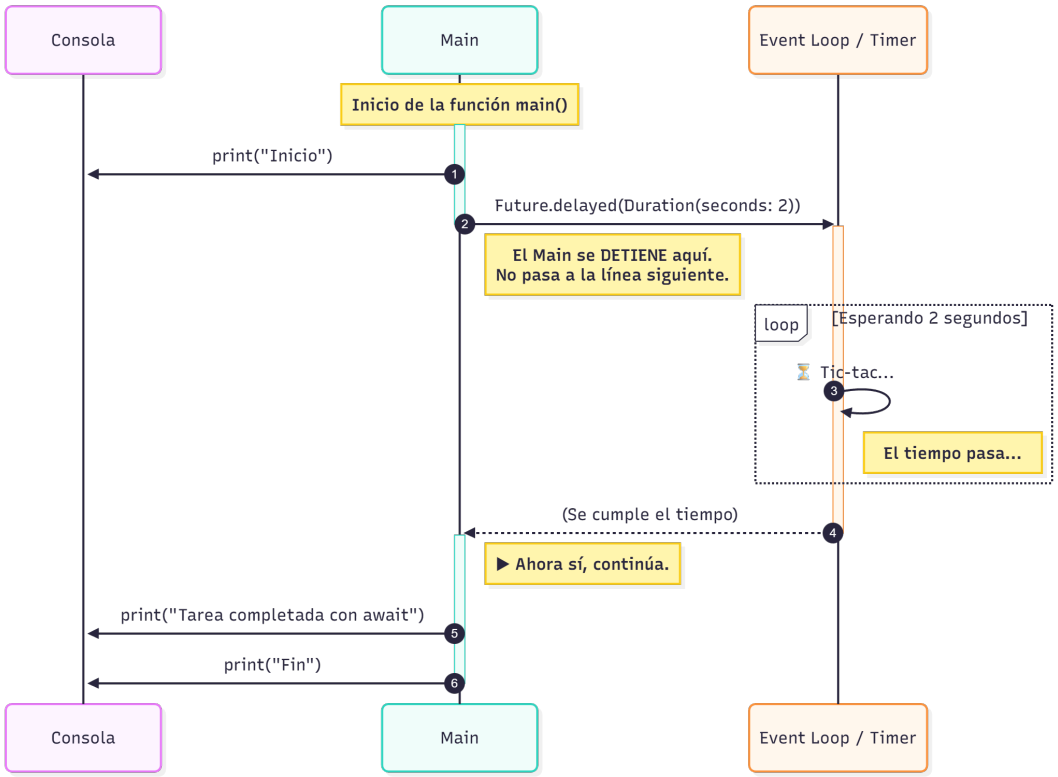


Figura 11: Diagrama de Secuencia: Ejemplo 2

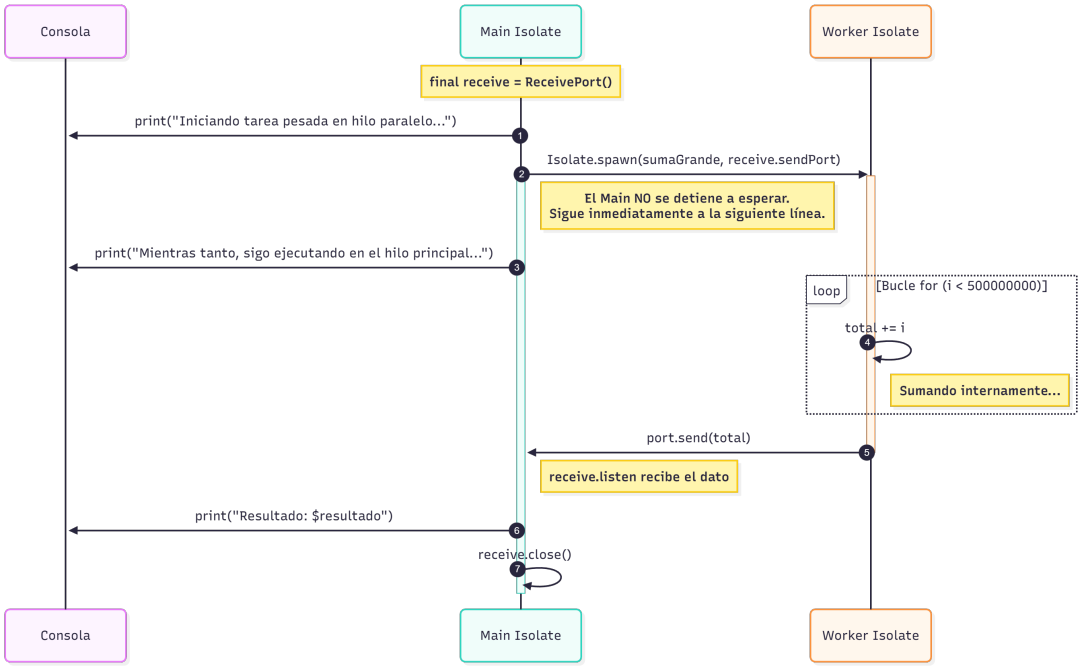


Figura 12: Diagrama de Secuencia: Ejemplo 4

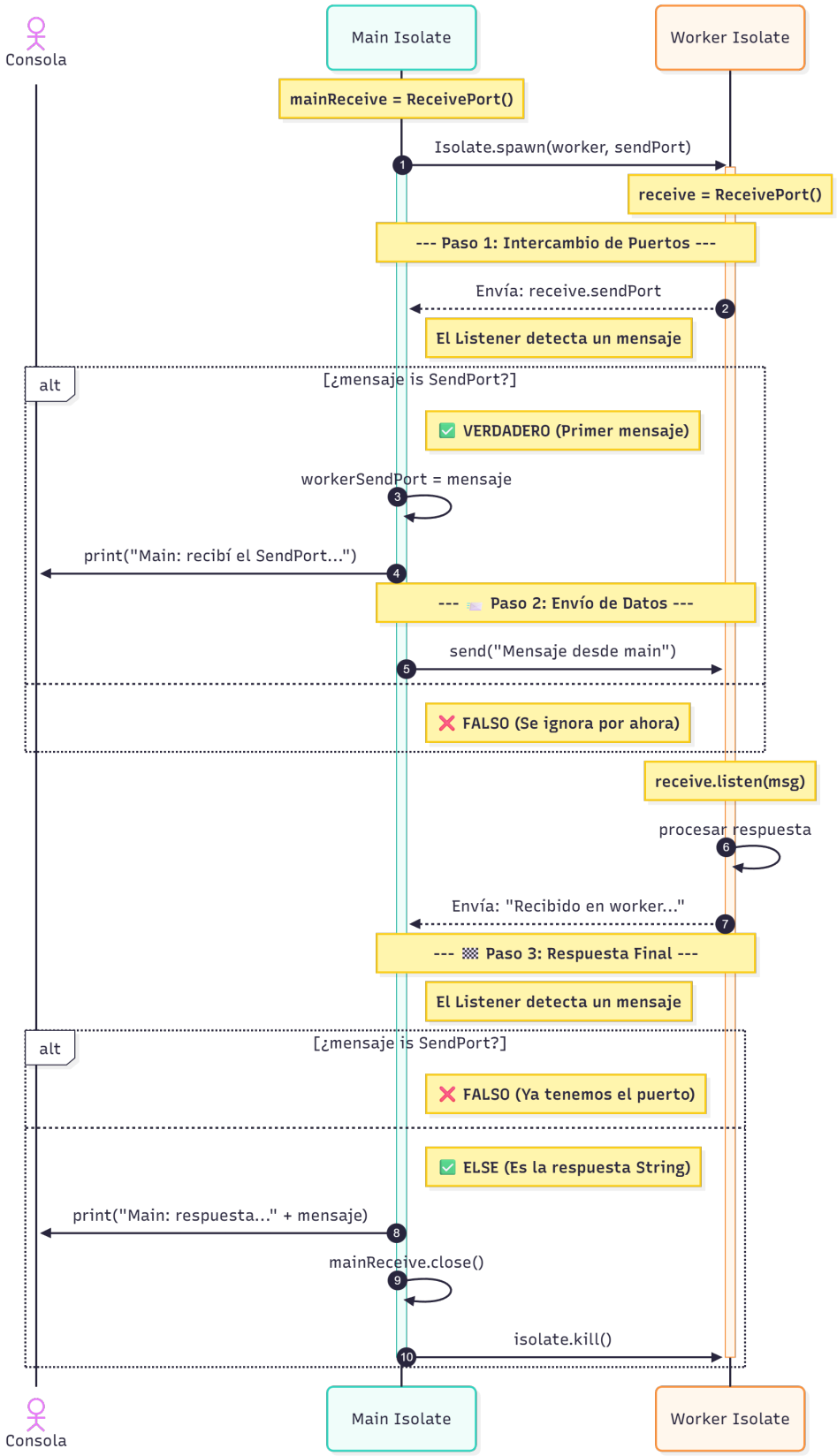


Figura 13: Diagrama de Secuencia: Ejemplo 5

6. Resultados

Archivos

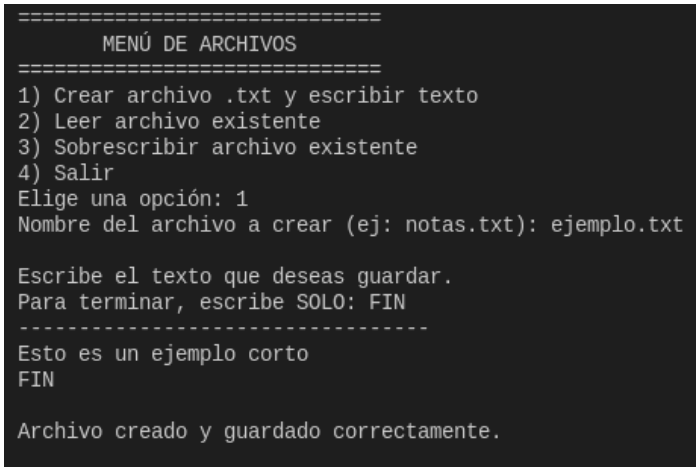


Figura 14: Creando un archivo de ejemplo y escribiendo texto en él.

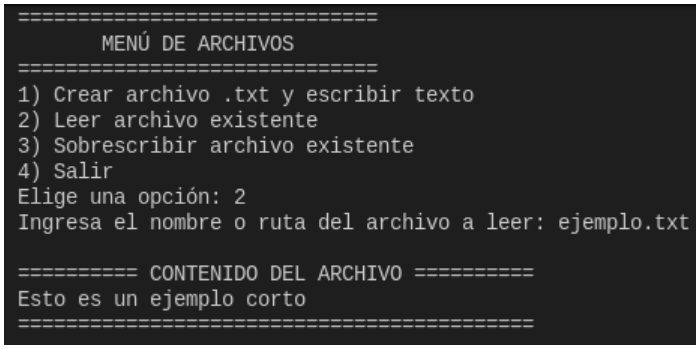


Figura 15: Leyendo el archivo creado.

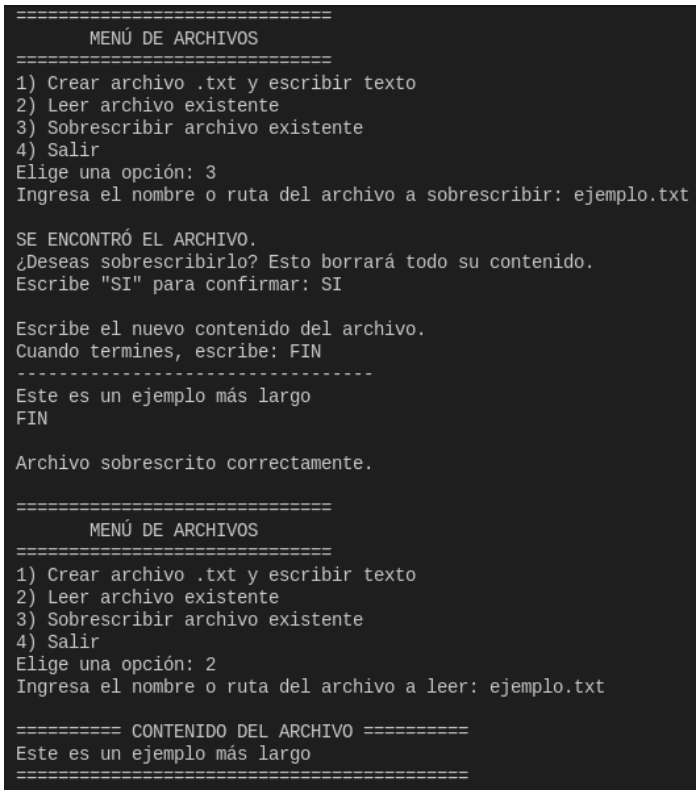


Figura 16: Sobrescribiendo el archivo creado y leyéndolo nuevamente para corroborar el resultado.

Hilos

Implementación donde se busca demostrar los procesos independientes de los hilos al ejecutar el programa.

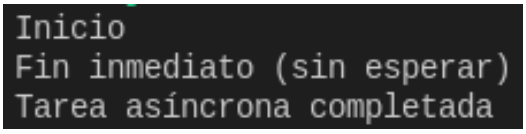


Figura 17: Salida del ejemplo 1.

Implementación donde se ejemplifica la ejecución de hilos de manera secuencial (uno espera al otro).

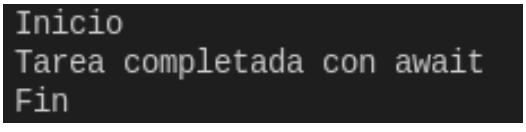


Figura 18: Salida del ejemplo 2.

Ejemplo básico de comunicación entre hilos.

Figura 19: Salida del ejemplo 3.

Implementación donde se aprecia la repartición de tareas hacia cada hilo.

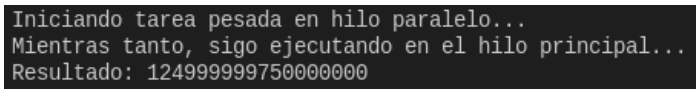


Figura 20: Salida del ejemplo 4.

Ejemplo de comunicación y paso de mensajes entre los hilos y su interacción con el método principal.

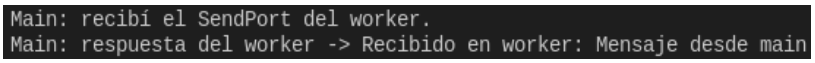


Figura 21: Salida del ejemplo 5.

Implementación de resolución del problema del ejemplo 4 de manera secuencial, con el objetivo de comparar tiempos de ejecución y consumo de recursos.

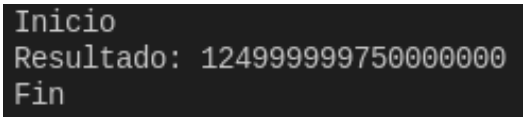
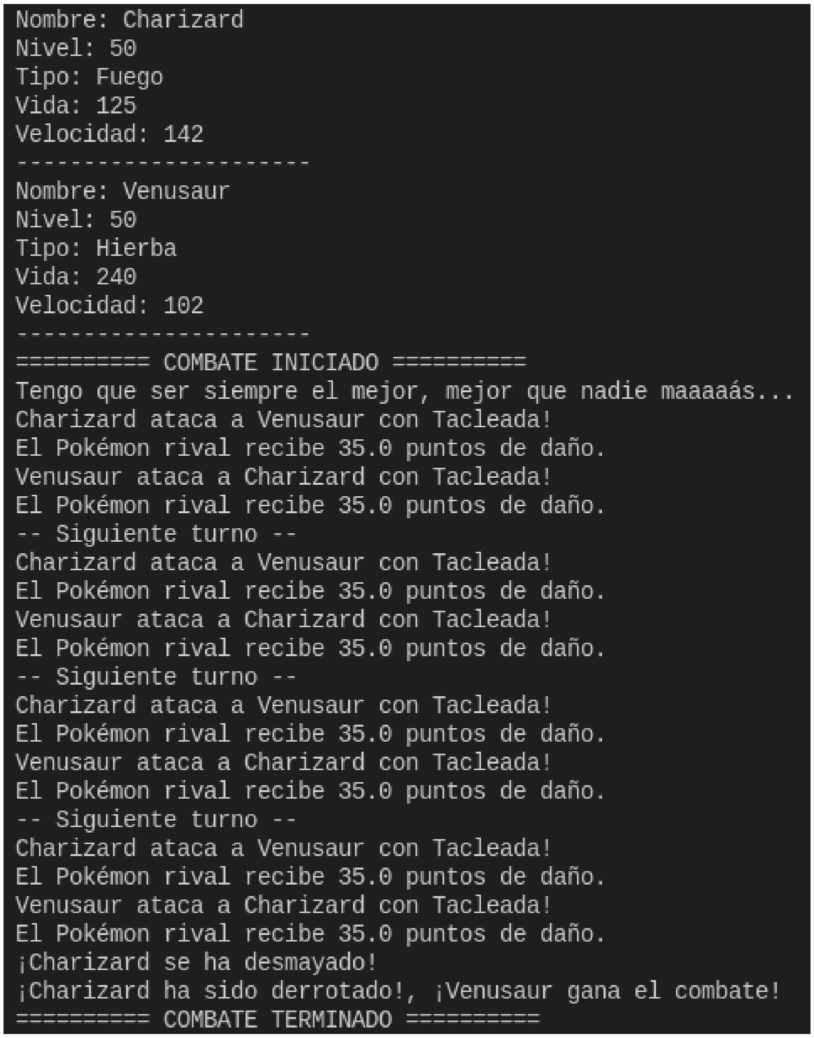


Figura 22: Salida del ejemplo 6.

Patrones de diseño



```
Nombre: Charizard
Nivel: 50
Tipo: Fuego
Vida: 125
Velocidad: 142
----------------------
Nombre: Venusaur
Nivel: 50
Tipo: Hierba
Vida: 240
Velocidad: 102
----------------------
========== COMBATE INICIADO ==========
Tengo que ser siempre el mejor, mejor que nadie maaaaás...
Charizard ataca a Venusaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Venusaur ataca a Charizard con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charizard ataca a Venusaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Venusaur ataca a Charizard con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charizard ataca a Venusaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Venusaur ataca a Charizard con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Charizard ataca a Venusaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Venusaur ataca a Charizard con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
¡Charizard se ha desmayado!
¡Charizard ha sido derrotado!, ¡Venusaur gana el combate!
========== COMBATE TERMINADO ==========
```

Figura 23: Ejecución del programa con pokemones de prueba.

7. Conclusiones

La identificación de las clases y métodos de entrada/salida de datos evidenció la correcta implementación del manejo de archivos, permitiendo al sistema la persistencia de la información.
El análisis de los ejemplos de concurrencia facilitó la comprensión de la gestión de hilos, mostrando cómo se estructuran las tareas para un procesamiento eficiente y simultáneo.
Además, la interpretación de los patrones de diseño presentes en la arquitectura del código mostró la importancia de utilizar soluciones ya establecidas para garantizar un diseño extensible y sostenible. La elaboración de los Diagramas UML complementó el proceso, reforzando la habilidad de modelar y documentar sistemas de software complejos y confirmando la comprensión de los conceptos requeridos en esta práctica.

Referencias

[1] Google Developers, *Dart Language Tour*. Disponible en: `https://dart.dev/language` [Consultado: 26-nov-2025].

[2] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed., Addison-Wesley, 2003.

[3] D. van der Linden, *Effective Dart: Usage*. Disponible en: `https://dart.dev/guides/language/effective-dart/usage` [Consultado: 26-nov-2025].