	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: Programación Orientada a Objetos

Grupo: 7

No de Práctica(s): Práctica 11, 12 y 13

Integrante(s): 322044339

322094152

322114461

425093384

322150984

No. de brigada: 6

Semestre: 2026-1

Fecha de entrega: 28 de noviembre de 2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción **2**

2. Marco Teórico **2**
2.1. Archivos 2
2.2. Hilos 3
2.3. Patrones de diseño 3

3. Desarrollo **4**
3.1. Archivos 4
3.2. Hilos 5
3.2.1. Metodo Future y Async 5
3.2.2. Metodo Await 5
3.2.3. Uso de ReceivePort() 6
3.2.4. Ejemplo4 6
3.3. Patrones de diseño 6
3.4. Diagrama estático 7
3.5. Diagrama dinámico 8

4. Resultados **9**
4.1. Archivos 9
4.2. Hilos 11

5. Conclusiones **14**

1. Introducción

Interacciones con diferentes componentes del sistema son habituales en la elaboración de software, ya sea para guardar datos, realizar tareas en segundo plano o estructurar programas más extensos y sencillos de mantener. Conforme los proyectos se expanden, la necesidad de trabajar con archivos, ejecutar operaciones que no obstaculicen la ejecución principal y estructurar el código a través de patrones que distingan la lógica de la interfaz también se incrementa.

Por esta razón, en la siguiente práctica se estudian tres asuntos importantes dentro del lenguaje Dart: la gestión básica de archivos, la realización de tareas simultáneas y la aplicación de patrones de diseño. Cada uno de estos componentes soluciona un tipo diferente de problema que surge con regularidad en aplicaciones reales. La gestión de archivos posibilita la conservación y recuperación de datos externos al programa, lo cual es esencial cuando se requiere persistencia. Por otro lado, la concurrencia hace posible que algunas operaciones se ejecuten sin interrumpir el flujo principal, lo cual es fundamental para tareas pesadas o procesos que requieren tiempo. Para terminar, los patrones de diseño contribuyen a que el código tenga una estructura más clara, sea fácil de ampliar y se pueda reutilizar.

Durante la práctica, estos conceptos se trabajan de forma independiente, lo que posibilita apreciar cómo se implementan directamente sin recurrir a una interfaz gráfica o a un entorno complicado. Esto facilita la comprensión de su objetivo y de cómo Dart los aplica. En resumen, estos asuntos ofrecen herramientas que son esenciales para desarrollar programas más completos y profesionales en trabajos futuros.

2. Marco Teórico

2.1. Archivos

Dentro del lenguaje Dart, tenemos una biblioteca que nos proporciona APIs con las cuáles podemos trabajar con archivos en aplicaciones que no corren en el navegador.

Esto permite que un programa Dart realice operaciones de entrada/salida directamente sobre el sistema de archivos, lo que resulta fundamental para tareas como leer o guardar datos persistentes, manejar configuraciones, logs, etc.

Operaciones que permite la biblioteca dart.io:

- Leer un archivo de texto: Al leer un archivo de texto codificado con UTF-8, puede leer todo el archivo Contenido del archivo con "*readAsString()*". Cuando las líneas individuales son importantes, se puede utilizar "*readAsLines()*" en ambos casos se devuelve un "*Future<String>*" que contiene el archivo en forma de una o mas cadenas de texto.
- Leer archivos en binario: Con el método "*readAsBytes()*" puedes leer un archivo como bytes (una lista de <int>) y devuelves un "*Future<String>*" cuando el resultado se encuentra disponible.

- Tratar errores de manipulación: Para capturar errores que no resulten en excepciones no capturadas, pudiendo manejarse con un "*catchError*"
- Flujo de contenido de archivos: Permite leer un archivo poco a poco, como un flujo, en lugar de cargarlo por completo a la memoria.
- Escribir sobre archivos: Puede escribir datos en un archivo. Utilizando el método "*openWrite()*" el método para obtener un "*IOSink*" (una salida que combina texto y bytes) en el que puedes escribir. El modo predeterminado, "*FileMode.write*", completamente sobrescribe los datos existentes en el archivo.
- Listar archivos de un directorio: Útil para encontrar todos los archivos y subdirectorios en un directorio; es una operación asíncrona. El método "*list()*" devuelve un flujo que emite un objeto cuando se encuentra un archivo o directorio.

2.2. Hilos

En Dart, en lugar de usar los clásicos "hilos compartidos" con memoria compartida, cada unidad de concurrencia se ejecuta en un "*isolate*": cada "*isolate*" tiene su propio contexto de memoria, su propia cola de eventos y su propio bucle de ejecución, de modo que no hay memoria compartida entre "*isolates*". Al crearse un nuevo isolate mediante Isolate (usando por ejemplo Isolate.run() o Isolate.spawn()), una función o tarea se ejecuta en paralelo al isolate principal, lo que permite hacer cómputos costosos, procesamiento de datos pesados u operaciones de entrada/salida intensivas, sin bloquear la ejecución principal. La comunicación entre isolates se realiza exclusivamente por paso de mensajes: se usan objetos como "*ReceivePort*" y "*SendPort*", de forma que un isolate puede enviar datos a otro, pero no existe acceso directo a la memoria del otro. Este modelo, promueve un estilo de programación más seguro y confiable, aunque requiere diseñar la aplicación pensando en comunicaciones por mensaje.[**2**]

2.3. Patrones de diseño

[4]

Un patrón de diseño es una solución repetible y general para problemas de ocurrencia cotidiana en el diseño de software. Es un modelo de como resolver un problema que puede utilizarse en diferentes situaciones. Los patrones de diseño permiten agilizar el proceso de desarrollo de solución debido a que proveen un paradigma desarrollado y aprobado.

En el diseño de software efectivo considera detalles que no son fáciles de ver hasta que se implementa una solución, por lo tanto, uno debe anticiparse a los problemas. La reutilización de patrones de diseño ayuda a prevenir los detalles que provocarían problemas más complejos, además de ayudar a realizar un código mas legible para los programadores o analistas que se encuentran familiarizados con patrones de diseño.

Los patrones de diseño describen un problema que ocurre varias veces de forma cotidiana y describe el núcleo de la solución, de tal forma que esta solución pueda ser implementada muchas veces.

Constan de 4 partes:

1. El nombre del patrón, el cual describe el diseño del problema, su solución y consecuencias.

2. El problema, describe cuando debe implementarse el patrón, explicando el problema y su contexto.

3. La solución, describe los elementos que conforman al diseño, su relación, responsabilidades y colaboraciones, provee una visión abstracta del diseño de un problema y como ciertos elementos resolverían el problema, de forma general.

4. Las consecuencias, son las recompensas y el resultado de haber aplicado el patrón.

3. Desarrollo

En esta práctica ahora abordamos tres temas distintos, siendo estos Archivos, Hilos y Patrones de diseño con un poco del tema de Errores, aunque en la páctica solo abordaremos los primeros tres, desglosando su código, funcionalidad e importancia:

3.1. Archivos

[3] Los archivos son un tema fundamental en la computación, por lo que mediante dart podemos realizar operaciones sobre ellos como se mostró en el ejemplo práctico. Primero creamos la función principal que de inmediato nos despliega un menú con 4 opciones, 3 de ellas siendo las operaciones de crear, leer y sobrescribir un archivo, la cuarta siendo la puerta de salida para terminar la ejecución del programa. Se leerá la opción del usuario con stdio.readLineSync() y lo convertirá en número con int.tryParse, claramente si la opción no es valida se mostrará un mensaje de error.

Si es que el usuario elige la opción 1 se llamará a la función crearYEscribirArchivo() la cual pedirá el nombre del archivo stdout.write, si es que no se escribe nada mostrará un error, después capturará las líneas de texto en un lista, el usuario puede escribir libremente, pero cuando escribe FIN el programa deja de pedir líneas pues el texto se habrá concretado, para crear el archivo lo nombra con el nombre establecido y haciendo comunicación con writeAssStringSync(lineas.join()) lo crea y escribe dentro.

Si es que se elige la opción 2 se llama a leerArchivoExistente(), función la cuál lee el contenido de un archivo y lo muestra en la pantalla, para ello primero solicita la ruta o el nombre del archivo, verifica si es que existe haciendo uso de archivo.existSync(), y si es que de verdad existe lee el contenido, que sería imprimirlo mediante una variable llamada contenido que recibe la comunicación entre archivo.readAsStringSync()

Si es que se elige la opción 3 se llama a sobrescribir un archivo(), para ello, de manera similar al anterior se pide la ruta del archivo y si es que existe se pedirá la

confirmación mediante la palabra clave "SI", evitando que se sobrescriba por equivocación, entonces se capturarán las nuevas líneas para concretar la sobre escritura del archivo de manera similar a la opción 1, entonces se podría decir que la operación 3 combina elementos de tanto la primera como la segunda, por que hay que buscar un archivo y en lugar de leerlo, editarlo.

Por último si se selecciona la opción 4, simplemente se sale del programa finalizando la ejecución y habiendo creado, leído o modificado archivos.

3.2. Hilos

[2] Muchas veces lo ideal para que un programa sea optimo es tener varios flujos de ejecución para poder realizar tareas sin necesidad de esperar a que se concrete la anterior, por eso es que usamos hilos de ejecución, en el caso de Flutter trabajamos con Isolate y event loop, el primero es una unidad de ejecución totalmente independiente por lo que no comparte memoria y no existen condiciones de carrera que lleven a errores o corrupción de datos, el segundo es la forma en que dart maneja tareas concurrentes con métodos como Future, Async, await y stream, ejecutando tareas en un solo hilo de forma asíncrona. Se realizaron 4 códigos para la explicación de este tema

3.2.1. Metodo Future y Async

[1] En el primer ejemplo el código inicia como el método Future<void>esto indica que la función retorna un Future u objeto con valor asíncrono sin un valor de retorno, por su parte, Async marca a la función como asíncrona, permitiendo el uso de operaciones como await, aunque en este c+odigo no se usa ese método, después de imprimir el inicio se usa un Future.delayed(Duration(secons:2,()), esto quiere decir decir que se crea un Future despues de un retraso de d2 segundos, los parentesis posteriores son una función callback que se ejecutará después del delayed, terminando con una impresión. Entonces lo que se esperaría de salida es primero recibir un mensaje de inicio, programar un mensaje asíncrono dentro de 2 segundo, escribir un mensaje de fin y después recibir el mensaje con retraso "Tarea asincronica completada"

3.2.2. Metodo Await

Primero declaramos la función principal haciendo uso de Future<void>y Async como en el ejemplo anterior, imprimiendo primero el mensaje de inicio, pero ahora haremos uso del comando await el cual pausa la ejecución hasta que Future se complete, entonces si hacemos casi lo mismo que en el ejemplo anterior, pero con la distinción del await entonces imprimiría la secuanecia Ïnicio", "Tarea completada con awaitz cerraría con un fin, por lo que se podría decir que es un como una barrera hasta que termine Future.

3.2.3. Uso de ReceivePort()

Primero se importa la libreria de isolates y se crea una función que se ejecutará en un isolate secundario usando sentPort port que será el parámetro que permitirá enviar mensajes al isolate principal, después se hace una comunicación usando sendPort para enviar un mensaje de texto al Isolate principal. La función principal será asincrona haciendo uso Async, dentro de ella creamos un puerto de recepción para recibir mensajes desde otros Isolates. Haciendo uso de Isolate.spawn se crea otro isolate al cuál se le asigna una tarea, se pasa al puerto de envío al isolate secundario y con await se espera la creación de ese isolate. Para ello ahora necesitamos configurar un listener que se ejecutará cada vez que detecte que llego un mensaje del isolatee secundario, finalmente imprimiendo el mensaje. Solo así es como obtenemos un mensaje desde otro isolate, siendo "Hola desde otro Isolate"

3.2.4. Ejemplo4

Este código primero importa la libreria de isolates pues tendremos procesos paralelos, después crearemos una función que se ejecutará en el isolate secundario haciendo uso de SendPort y este será sumaGrande, se inicializará un total en 0 y mediante un bucle for de 5 mil millones se incrementará el total más i, como está operación es pesada se ejecutará en el isolate secundario y al final del bucle se envia el resultado al isolate principal. En el main se crea un puerto para recibir mensajes del isolate secundario haciendo uso de ReceivePort(), se crea un isolate secundario con la función a ejecutar sumaGrnade y con ayuda de await se espera a que el isolate se cree, no que termine. Mientras esa garn tarea termina el hilo principal sigue disponible mandando un mensaje de ello, se crea un listener que avisará cuando el isolate secundario termine y se imprime el resultado del calculo

3.3. Patrones de diseño

En este apartado de la práctica se utilizó un ejemplo que une los fundamentos de la Programación Orientada a Objetos con una estructura que distingue nítidamente entre la lógica del programa, la vista y los datos. Esta organización no solo limpia el código, sino que también sigue la idea de los patrones de diseño, específicamente la perspectiva Modelo-Vista-Controlador (MVC), en la cual cada componente tiene un papel diferente dentro del programa.

Primero, el archivo contiene la definición de diversas clases que constituyen el modelo. La clase principal de Pokémon está aquí, y describe las características que cada Pokémon requiere. Además, se incluyen las clases especializadas PokemonFuego y PokemonHierba. Las clases de ataques también están incluidas, las cuales tienen nombre, tipo y potencia. La idea es que estas clases incluyan solamente datos y procedimientos concretos, como determinar la vida, la velocidad y el tipo de ataque en función de la especie.

Se utiliza una interfaz llamada CombateView en la sección de la vista, que define lo que cualquier vista debería poder mostrar: datos sobre los Pokémon, acciones llevadas a cabo, daño y mensajes acerca del estado de la batalla. La implementación específica que muestra los datos en pantalla es la clase ConsoleCombateView. Al dividir esta parte, el mismo enfrentamiento podría representarse posteriormente en una interfaz gráfica, un sitio web o cualquier otra sin que la lógica interna se altere.

La coordinación del combate está a cargo del controlador, que es la clase CombateController. Esta sección no tiene la función de presentar información, sino que se ocupa de determinar qué sucede en cada turno, quién ataca primero y cuántos puntos de vida pierde cada Pokémon. Además, establece cuándo acaba la pelea y transmite el resultado a la vista. Así, la lógica del juego se concentra en una única clase, mientras que la vista solo imprime lo que le señala el controlador.

Por último, el main crea los objetos requeridos y le solicita al controlador que comience la batalla. Esta estructura posibilita que el programa sea organizado, sencillo de expandir y con responsabilidades repartidas de manera adecuada, lo cual representa la finalidad principal de los patrones de diseño en el desarrollo del software.

3.4. Diagrama estático

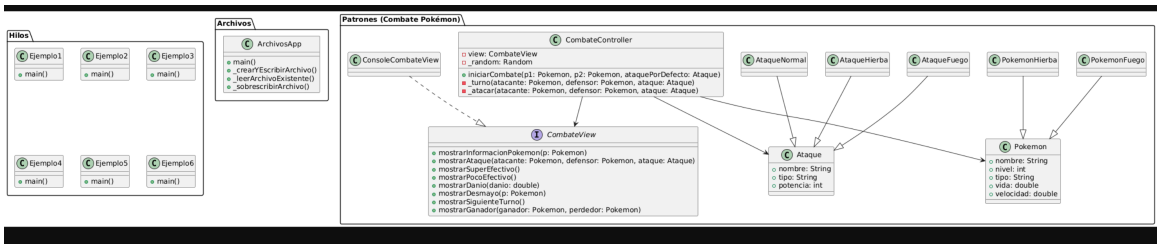

Figura 1: UML Diagrama de clases

3.5. Diagrama dinámico

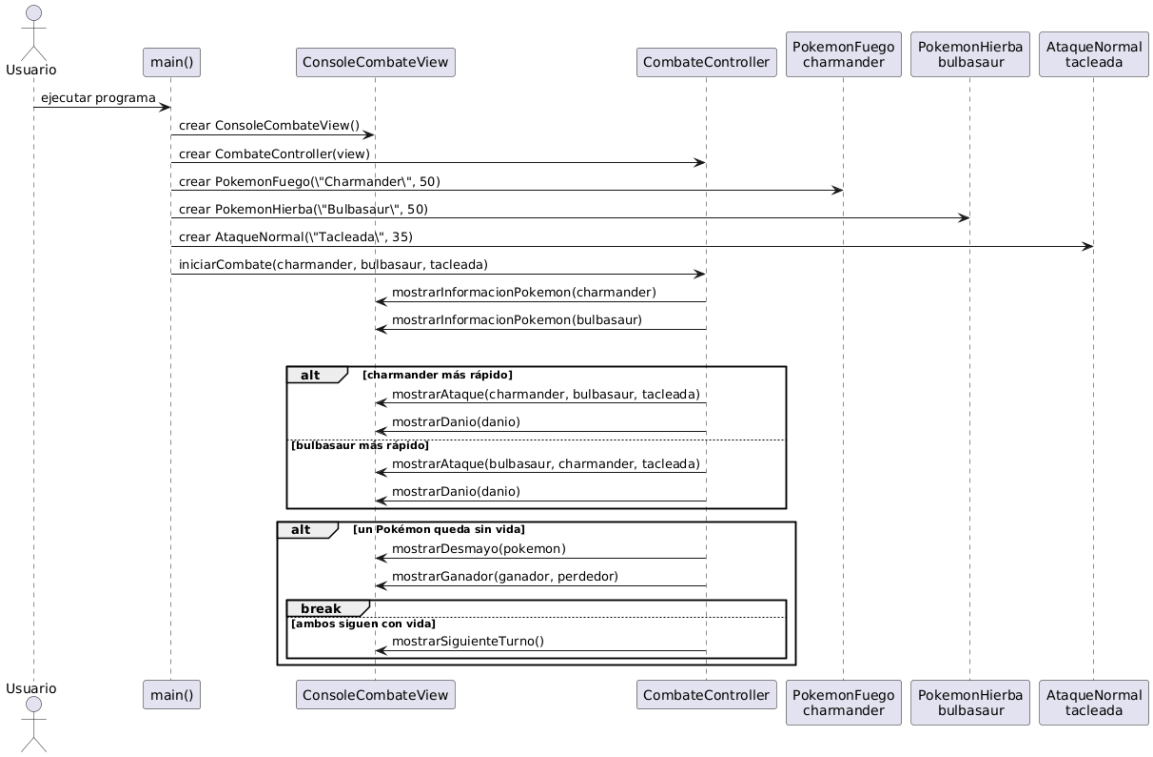

Figura 2: UML Diagrama de secuencia

* Recomendamos hacer zoom para una mejor visualización

4. Resultados

4.1. Archivos

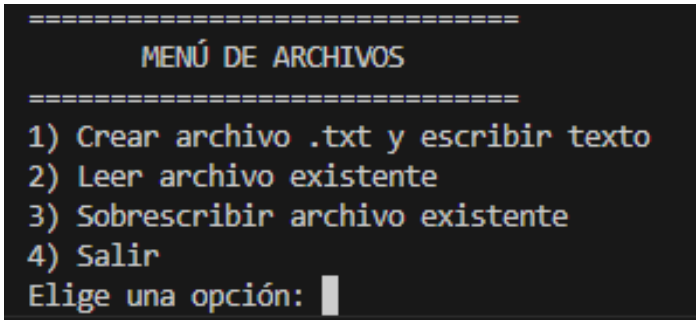

Figura 3: Menú tras ejecución

Al momento de ejecutar el programa se despliega el menú y se solicita al usuario ingresar la opción deseada.

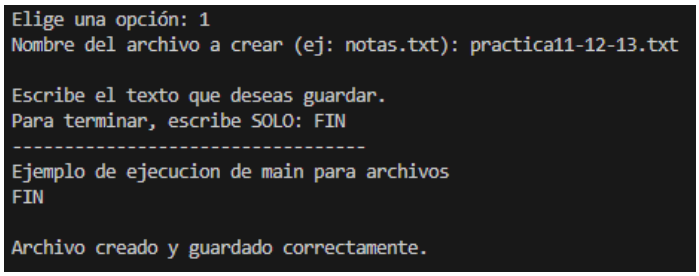

Figura 4: Creación de un archivo de texto

Se ingresa la opción 1, se ingresa el nombre del archivo a crear con la extensión "txt" y se finaliza y guarda con la palabra "FIN".

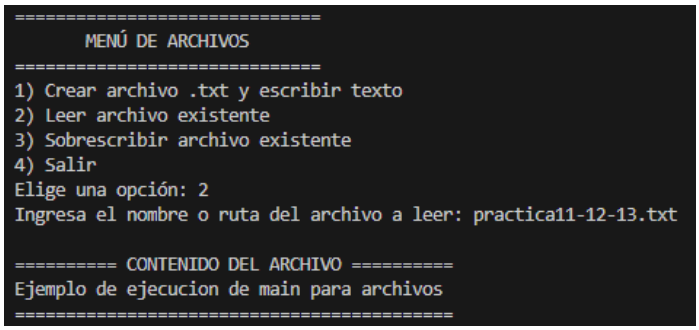

Figura 5: Lectura de archivo

Al ingresar la opción 3 una vez creado un archivo, se pregunta qué archivo se quiere leer, cuando se ingresa el nombre del archivo deseado se muestra su contenido.

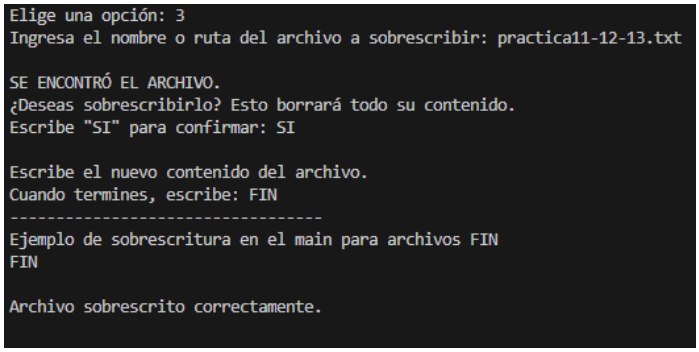

Figura 6: Sobrescritura de un archivo

Al ingresar la opción 3, se pregunta el archivo que se desea sobrescribir, cuando se selecciona el archivo deseado se da la opción de volver a escribir un texto nuevo, terminando de igual manera con la palabra ”FIN”.

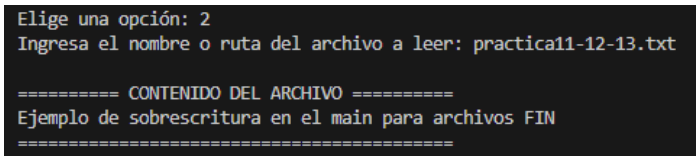

Figura 7: Nueva lectura tras la sobrescritura

Una vez terminada la sobrescritura, si se selecciona la opción 2 y se selecciona el archivo sobrescrito se observa el nuevo texto.

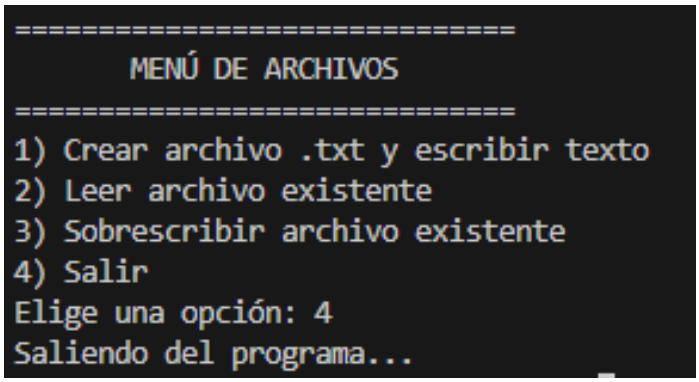

Figura 8: Salida del programa

Al momento de ingresar la opción 4, se muestra un mensaje de salida y se termina la ejecución.

4.2. Hilos

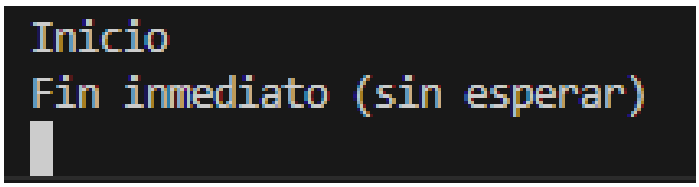

Figura 9: Ejemplo 1. Ejecución inmediata

Al momento de ejecutar el programa se despliega el siguiente mensaje, mientras, ”de fondo” continúa la ejecución del programa.

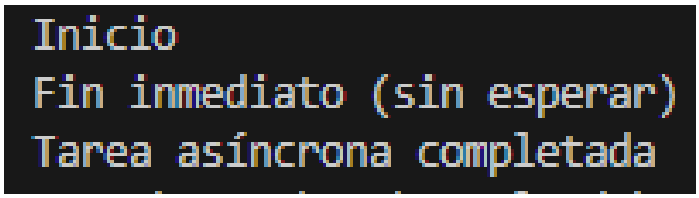

Figura 10: Ejemplo 1. Finalización de la ejecución

Pasado un tiempo de no más de un segundo, se termina la ejecución del programa completo.

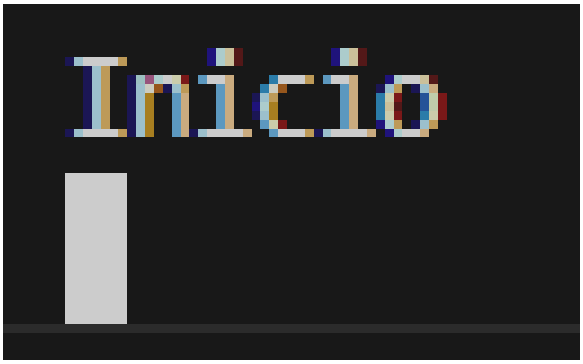

Figura 11: Ejemplo 2. Ejecución inmediata

Al momento de ejecutar el programa se despliega el siguiente mensaje, mientras, ”de fondo” continúa la ejecución del programa.

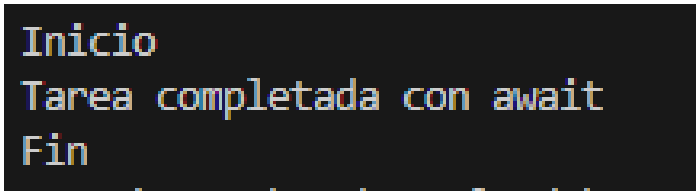

Figura 12: Ejemplo 2. Finalización de la ejecución

Pasado un tiempo de no más de un segundo, se termina la ejecución del programa completo.

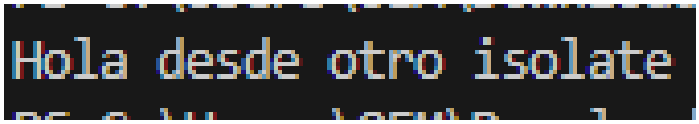

Figura 13: Ejemplo 3

El ejemplo 3 crea un isolate y le envía un mensaje desde el hilo principal. Ese isolate devuelve un texto de respuesta, y el programa lo imprime antes de cerrar la comunicación.

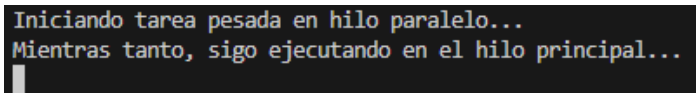

Figura 14: Ejemplo 4. Inicio de la ejecución

Se imprime el mensaje inicial mientras en ”segundo plano” se efectúa una suma muy grande.

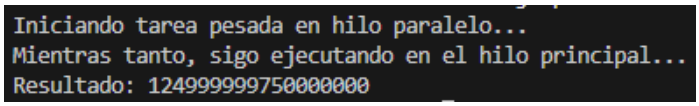

Figura 15: Ejemplo 4. Finalización de la ejecución

Pasado un tiempo se recibe el resultado de la suma cuando esta termina y se imprime.

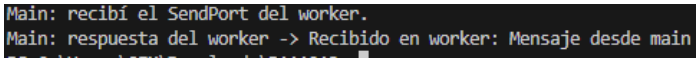

Figura 16: Ejemplo 5

Crea un isolate separado que puede enviar y recibir mensajes del hilo principal. Primero intercambian sus SendPort para establecer conexión y luego se envían

mensajes mutuamente, mostrando cómo funciona la comunicación en ambos sentidos.

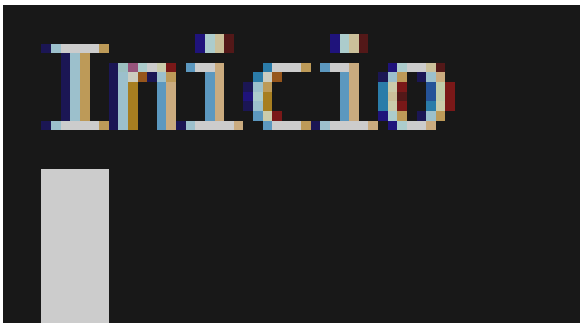

Figura 17: Ejemplo 6. Ejecución

Realiza una suma muy grande dentro del hilo principal sin usar isolates ni tareas asíncronas.

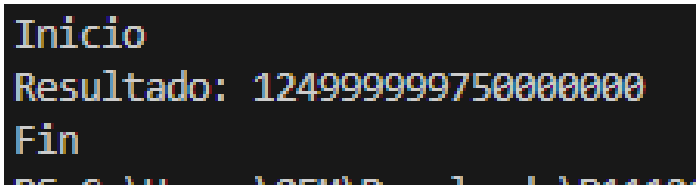

Figura 18: Ejemplo 6. Finalización de la ejecución

La complejidad de la suma hace que el programa se bloquee mientras termina el cálculo, mostrando la diferencia con ejecutar tareas pesadas en paralelo.

5. Conclusiones

En esta práctica se integraron tres elementos que aparecen con mucha frecuencia en el desarrollo de aplicaciones reales: el manejo de archivos, la ejecución concurrente y la organización del código mediante patrones de diseño. A través del uso de dart.io se vio cómo un programa puede comunicarse directamente con el sistema de archivos para crear, leer y sobrescribir información de manera controlada, cuidando aspectos como la validación de rutas y el tratamiento básico de errores. Esto muestra que la persistencia de datos no es solo un detalle adicional, sino una parte central del comportamiento de muchos programas.

En el apartado de hilos, los ejemplos permitieron comparar el uso de tareas asíncronas simples con Future y await frente al trabajo con isolates para realizar cómputos pesados. La diferencia entre ejecutar una operación costosa en el hilo principal y delegarla a un isolate se aprecia claramente en la respuesta del programa: en un caso la aplicación se bloquea mientras termina el cálculo y en el otro sigue activa mientras la tarea pesada se resuelve en paralelo. Esto refuerza la importancia de elegir el modelo de concurrencia adecuado según el tipo de problema que se quiere resolver.

Finalmente, el ejemplo de patrones de diseño, basado en un combate entre Pokémon, mostró cómo separar la lógica del dominio (modelo), la presentación de la información (vista) y la coordinación del flujo (controlador). Esta organización hace que el código sea más sencillo de leer, de probar y de extender, ya que cada parte tiene responsabilidades bien definidas. En conjunto, la práctica pone en evidencia que conceptos como archivos, concurrencia y patrones no son temas aislados, sino piezas que se pueden combinar para construir software más robusto, ordenado y preparado para crecer.

Referencias

[1] J. J. Deshpande et al. “Permissioned blockchain based public procurement system”. En: *Journal of Physics: Conference Series*. Vol. 1706. 2020.

[2] Freya Sheer Hardwick, Raja Naeem Akram y Konstantinos Markantonakis. “Fair and Transparent Blockchain Based Tendering Framework - A Step Towards Open Governance”. En: *Proceedings of the 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications and the 12th IEEE International Conference on Big Data Science and Engineering (Trustcom/BigDataSE)*. 2018.

[3] D. Mali et al. “Blockchain-based e-Tendering System”. En: *Proceedings of the International Conference on Intelligent Computing and Control Systems (ICICCS 2020)*. 2020.

[4] D. Yaga et al. “Blockchain Technology Overview”. En: *arXiv* (2019).