	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): Practica 11 12 y 13

Integrante(s): 322125337

322080869

322059179

323629814

322113536

No. de lista o brigada: 6

Semestre: 2025

Fecha de entrega: 28/11/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

La práctica consiste en analizar y documentar una serie de ejemplos de código proporcionados en la serie de videos del curso. El problema central es desglosar e interpretar la implementación técnica de tres pilares fundamentales en el desarrollo de software: el manejo de Archivos para la persistencia de datos , el uso de Hilos para la ejecución concurrente y la aplicación de Patrones de Diseño para estructurar la arquitectura del sistema. La comprensión de estos temas es crucial para el desarrollo de software a nivel profesional y también más eficiente. El manejo de archivos permite que la información perdure más allá de la ejecución del programa, los hilos optimizan el rendimiento permitiendo realizar múltiples tareas simultáneamente, y los patrones de diseño ofrecen soluciones estandarizadas a problemas recurrentes.

- Explicar detalladamente la aplicación de los temas de Archivos, Hilos y Patrones de Diseño en los ejemplos analizados.
- Elaborar diagramas UML estáticos y dinámicos que representen fielmente la estructura y el comportamiento del código proporcionado.
- Realizar una interpretación profunda de los conceptos aplicados, demostrando el entendimiento de su funcionamiento e importancia dentro del código.

2. Marco Teórico

Trabajado por Juan Prado

Archivos: El manejo de Archivos en Java es el mecanismo que permite la persistencia de datos, posibilitando que la información sobreviva más allá de la ejecución del programa al ser almacenada en una unidad de almacenamiento fisica. La mayoría de los lenguajes utilizan el concepto de "flujos.º streams para gestionar la entrada y salida de datos. Estos flujos actúan como canales secuenciales que transportan bytes o caracteres entre la memoria principal y el dispositivo de almacenamiento, permitiendo operaciones fundamentales como la creación, lectura, escritura y eliminación de ficheros de manera controlada y eficiente. [1]

Hilos: Los Hilos (o threads) son las unidades básicas de utilización de la CPU que permiten la ejecución concurrente o simultánea de múltiples tareas dentro de un mismo programa. Los hilos facilitan la multitarea, permitiendo que procesos complejos se ejecuten en segundo plano sin congelar la interfaz de usuario o el hilo principal. Cada hilo opera de forma independiente con su propia pila de ejecución, pero comparte los recursos de memoria del proceso padre, lo que optimiza el rendimiento del sistema pero requiere una gestión cuidadosa de la sincronización para evitar conflictos en el acceso a datos compartidos. [2]

Patrones de Diseño: Un Patrón de Diseño se define como una solución probada, repetible y general a problemas que ocurren cotidianamente en la arquitectura de

software. Es fundamental comprender que no se trata de código terminado ni de una librería lista para ser insertada, sino de un modelo conceptual o plantilla (template) que guía al desarrollador sobre cómo resolver una situación específica. Estas soluciones fomentan la reutilización de diseños exitosos y estandarizan el vocabulario técnico entre programadores, clasificándose típicamente en tres categorías: creacionales (creación de objetos), estructurales (composición de clases) y de comportamiento (interacción entre objetos). [3]

3. Desarrollo [Analisis]

Trabajado por Edgar Saúl Bautista y Said Correa Sam

Interpretacion de conceptos aplicados en el codigo.:

-Archivos

El tema de Archivos visto previamente en los videos proporcionados se ve reflejado claramente de manera práctica en el código visto main.dart donde se muestra cómo una aplicación puede comunicarse con su entorno para obtener, procesar y almacenar información con el fin de hacerla persistente, en este caso en el código los archivos funcionan como elementos clave para mantener datos disponibles incluso después de que el programa finaliza su ejecución, aquí cada archivo se maneja como un objeto de la clase File. Un archivo no es solo una secuencia de texto sino una entidad con propiedades y métodos que permiten leerlo, escribirlo, verificar su existencia o sobrescribir su contenido.

El concepto de flujo de datos también se ve reflejado ampliamente en el código ya que las entradas de información se manejan a través del flujo de entrada mediante "stdin.readLineSync().ª través del cual el usuario proporciona datos que el programa recibe como un stream desde una fuente externa, además el flujo de salida se manifiesta en dos sentidos: tanto hacia la consola mediante print() y stdout.write() y como hacia los archivos mediante métodos como writeAsStringSync(). Esta doble salida nos muestra cómo el programa dirige datos tanto al usuario como al repositorio donde se almacenan de forma persistente.

También se ve incluido el repositorio el cual se tiene entendido tanto como el dispositivo físico de almacenamiento como el espacio lógico donde se organizan los archivos, el cual es representado en el código por los archivos .txt que se crean, leen o sobrescriben, además el programa verifica la existencia de estos repositorios mediante existsSync() y esto asegura que los flujos de datos se dirijan correctamente hacia un nodo válido. Cuando el programa escribe en un archivo la información se convierte en parte del repositorio lógico quedando almacenada en el dispositivo físico gracias al flujo de salida.

Incluso cada opción del menú refleja la aplicación directa de estos conceptos ya que la creación de un archivo implica recibir texto desde el usuario (flujo de entrada), encapsularlo en un objeto archivo y enviarlo hacia el repositorio mediante escritura (flujo de salida). Aunque también esta la lectura que demuestra el proceso inverso en donde recupera información persistente desde el repositorio hacia el programa, asi reafirmando la importancia del flujo de entrada desde una fuente externa. La sobrescritura combina ambos procesos pues requiere entrada del usuario, validación del repositorio existente y finalmente el envío del nuevo contenido hacia el archivo reemplazando completamente la información previa.

-Hilos

El tema de hilos de ejecución que vimos en el video previamente revisado se ve reflejado de una forma clara en los códigos, por ejemplo el event loop que permite manejar tareas asincrónicas dentro de un mismo hilo sin bloquear la ejecución y los isolates que representan hilos completamente independientes con su propia memoria y ciclo de ejecución, estos ejemplos vistos muestran cómo se evita la ejecución estrictamente secuencial que puede generar cuellos de botella y se optimiza el flujo de trabajo permitiendo que las tareas no dependan entre sí para continuar.

En los primeros ejemplos de código se muestra el funcionamiento del event loop el cual permite que una tarea programada no detenga el avance del programa , por ejemplo en el archivo Ejemplo1.dart la instrucción "Future.delayed"nos muestra cómo una tarea diferida no bloquea el hilo principal permitiendo que las demás instrucciones continúen ejecutándose mientras el futuro queda pendiente en la cola de eventos. Luego en el Ejemplo 2 el uso de async y await muestra una forma más controlada de como gestionar estas operaciones asincrónicas las cuales detenienen temporalmente la ejecución de la función sin bloquear el hilo ya que es el event loop quien reanuda la función cuando la operación ha concluido. En ambos casos podemos observar que reflejan la concurrencia lógica dentro de un solo hilo característica del modelo de event loop.

También vemos los ejemplos con isolates que muestran cómo se maneja la ejecución paralela real mediante unidades de ejecución completamente separadas, por ejemplo en el Ejemplo 3 se observa la creación de un isolate independiente utilizando Ïsolate.spawn", este nuevo hilo cuenta con su propio espacio de memoria y con su propio event loop por lo que no comparte memoria con el hilo principal, además la comunicación se realiza exclusivamente a través de puertos (SendPort y ReceivePort) cumpliendo con el modelo de paso de mensajes descrito en los conceptos. Después también tenemos el Ejemplo 4 donde esta capacidad se aprovecha para ejecutar una tarea bastante complicada computacionalmente sin bloquear el hilo principal, mientras el isolate secundario realiza la operación pesada el hilo principal continúa su ejecución normalmente quedando asi demostrado la eliminación de cuellos de botella

mediante procesamiento paralelo.

Luego en el Ejemplo 5 se profundiza la comunicación bidireccional entre isolates, donde tanto el isolate principal como el secundario intercambian mensajes estableciendo un canal de comunicación que permite coordinar acciones sin compartir memoria, con este ejemplo se muestra claramente las etapas fundamentales del ciclo de vida de un isolate: creación, ejecución, comunicación y finalización mediante la instrucción kill que cierra explícitamente el hilo secundario.

Por ultimo el Ejemplo 6 sirve como contraste ya que muestra una tarea pesada ejecutada completamente dentro del hilo principal, aquí la aplicación permanece bloqueada mientras se ejecuta el ciclo, reproduciendo precisamente el tipo de cuello de botella que los hilos de ejecución buscan evitar.

-Patrones de diseño

El tema Patrones de diseño lo podemos ver reflejado en los códigos revisados anteriormente en donde podemos observar el patrón Modelo–Vista–Controlador (MVC) y el patrón Singleton, ambos ejemplos nos muestran cómo los patrones permiten estructurar soluciones reutilizables, escalables y fáciles de mantener ya que aportan modelos probados para resolver problemas comunes en el desarrollo de software.

Para empezar en el archivo main.dart vemos que se implementa el patrón de diseño MVC al separar de manera explícita los componentes del sistema en Modelo, Vista y Controlador, las clases que representan a los Pokémon y sus ataques conforman el Modelo ya que encapsulan la información del dominio como los son los atributos, características y comportamiento base de los objetos. El Modelo no tiene conocimiento de cómo se mostrará la información ni de la secuencia del combate de esta manera cumpliendo con la idea central de este patrón que es mantener la lógica del dominio independiente de la interfaz y del control de la aplicación.
La Vista está representada por la clase ConsoleCombateView la cual su única responsabilidad es mostrar al usuario información relacionada con el combate como estadísticas, ataques, daño y resultados, esta clase no contiene lógica del proceso ni manipula datos más allá de su presentación lo cual alinea su función con la definición teórica de un objeto de Vista, por otra parte el Controlador (CombateController) actúa como intermediario entre Modelo y Vista siendo asi que coordina turnos, define el flujo del combate y actualiza la Vista con los cambios generados en el Modelo. Entonces este código ejemplifica adecuadamente cómo el patrón MVC divide responsabilidades y favorece la claridad, la extensibilidad y la reutilización del código.

Por ultimo en el archivo Singleton.dart podemos ver como se implementa el patrón de diseño Singleton para asi poder asegurar que la clase Impresora posea una única instancia accesible desde cualquier parte del programa, este patrón se aplica correctamente mediante un constructor privado, el mantenimiento de una instancia estática y un método público (en este caso un factory) que siempre retorna la misma

instancia.
En el código la impresora centraliza la cola de documentos y el historial de impresiones lo cual resuelve un problema en donde un recurso compartido no debe duplicarse para evitar inconsistencias, además el patrón garantiza que todos los usuarios del sistema interactúen con la misma impresora lo que se demuestra al comprobar que diferentes variables que intentan obtener una instancia terminan apuntando al mismo objeto, esto logra evitar la creación accidental de múltiples colas de impresión o historiales independientes de tal forma que muestra las consecuencias positivas esperadas de este patrón las cuales son el control centralizado, coherencia y previsibilidad del comportamiento del recurso compartido.

Enfasis en interpretación de los conceptos:

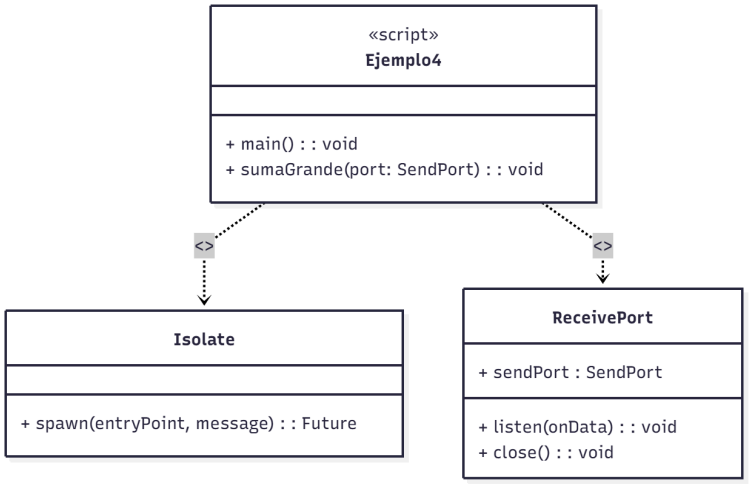
Interpretación de Archivos:
Los datos de un programa solo estan vivos mientras que el programa tambien lo esté, en caso de que el programa finalize los datos se pierden con el. Para hacer esto necesitamos almacenar los archivos de otra forma.
El codigo establece el flujo de salida. Crea una conexión desde el programa hacia donde se va a almacenar (un archivo en este caso). Asi dejando de existir solo en el codigo, pasando a ser una entidad en el sistema (siendo lo más importante, que ahora se guarda)

Interpretación de Hilos:
Los programas se ejecutan de manera secuencial, por lo que si una tarea es lenta puede generar un cuello de botella, haciendo el programa lento Para evitar esto empleamos métodos para tener varios flujos de ejecución al mismo tiempo, y aqui necesitamos los hilos:
Event loop: Simula hacer varias cosas a la vez avanzando las tareas poco a poco, esperando y avanzando, pero en realidad siguen en un solo hilo
Isolate: Es una unidad de ejecución independiente, por lo que si es paralelo, tampoco comparrten memoria por lo que no hay condiciones de carrera. La forma en que se puede comunicar con otros programas es a travez de puertos

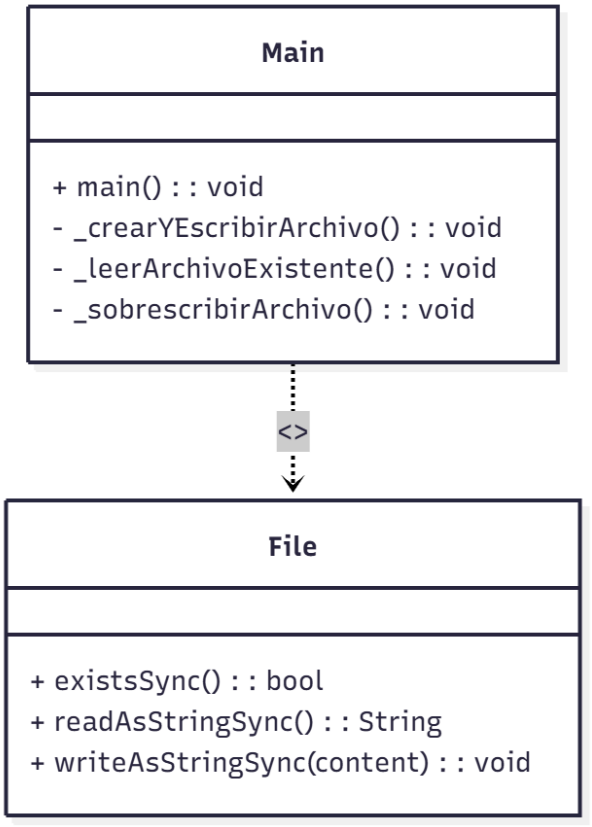
Interpretación de Patrones de Diseño:
Un patron no es un codigo que copiemos en cada uno de los codigos donde se implemente, sino un modelo estandar a seguir para resolver un problema, gracias a esto nos podemos anticipar a problemas y hacer más legible el código
Patrón Singleton: Hay objetos donde solamente debe existir uno, teniendo una estructura de cinstructor privado y con un punto de acceso estatico bien conicido
Modelo-Vista-Controlador (MVC): Este se divide en 3, Modelo (objetos con información encapsulada y los metodos establecidos), la Vista (la consola que es lo que el usuario puede ver, su proposito es mostrar lo que da el modelo sin alterarlo) y el Controlador (intermediario entre el modelo y vista)

Diagramas UML estático:
Trabajado por Alfaro Carreto Irving Alberto

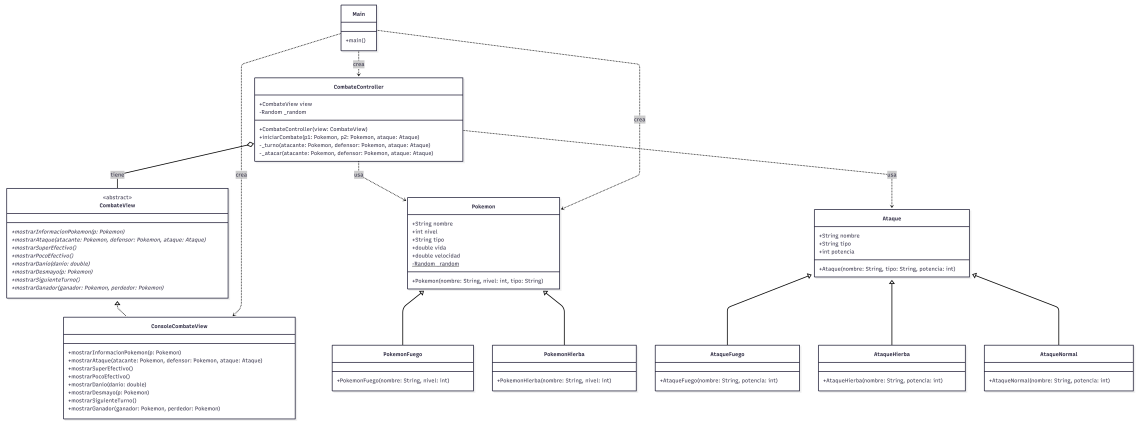
Hilos (Ejemplo 4):



Archivos:

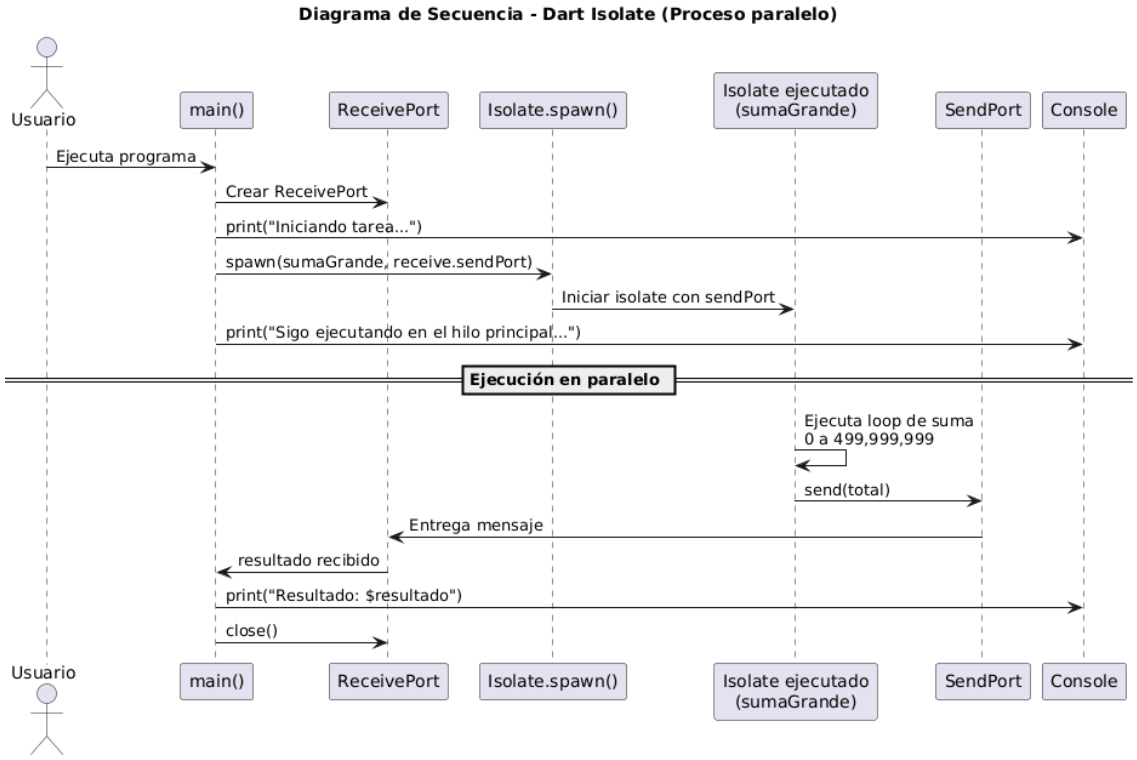


Patrones:



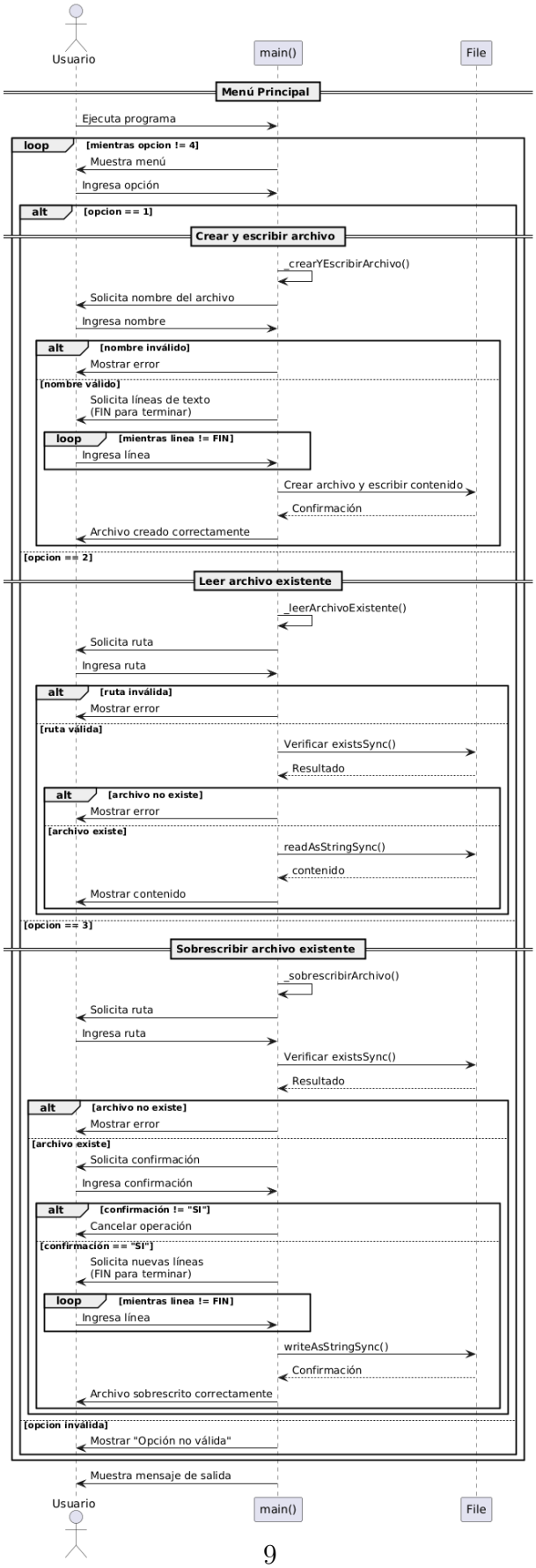
Diagramas UML Dinámico:
Trabajado por Urzúa Sánchez Enrique

Hilos, cuarto ejemplo:

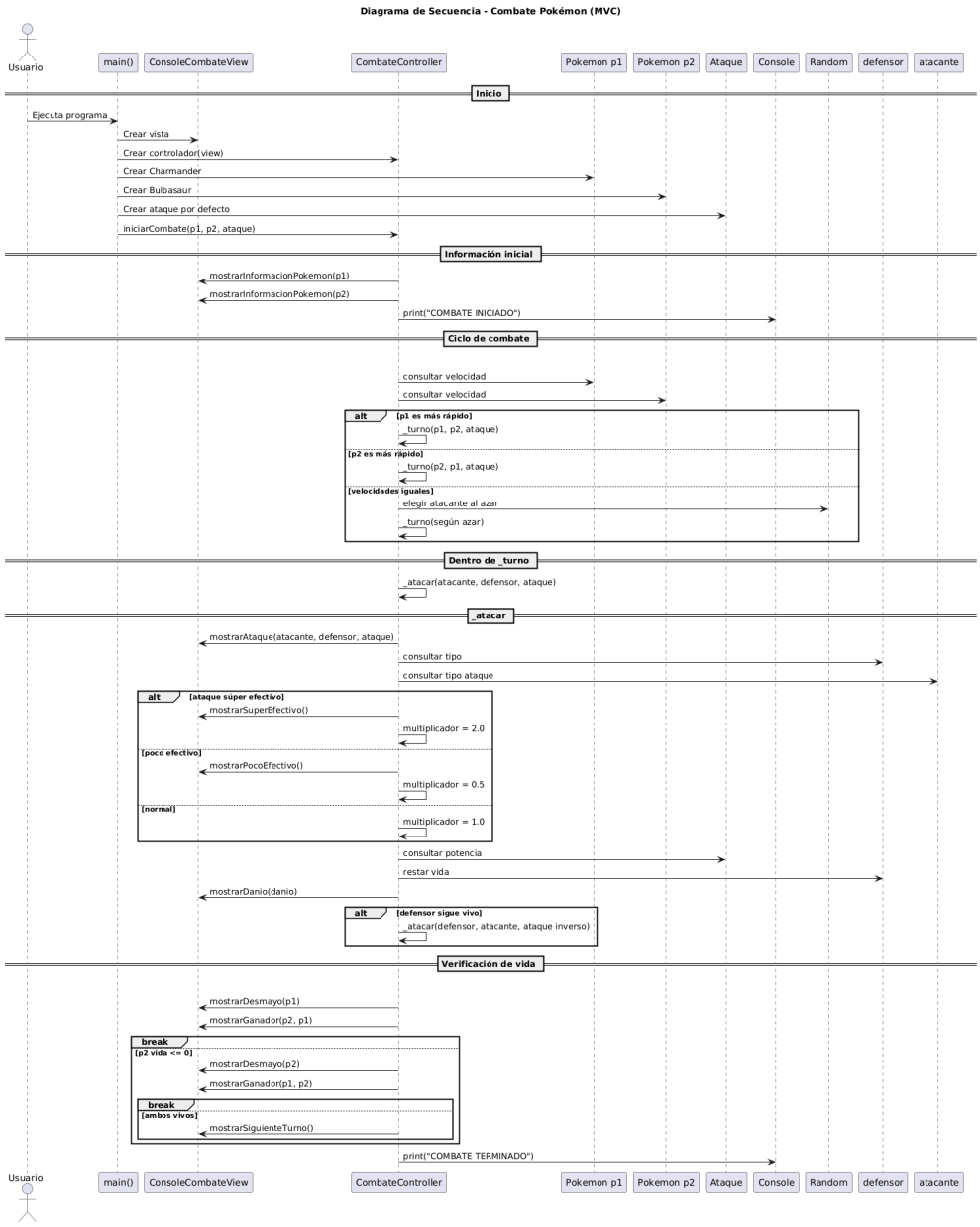


Archivos:

Diagrama de Secuencia - Programa de Gestión de Archivos (Dart)



Patrones:



4. Resultados

La utilidad central del código de archivos está en la persistencia de datos, el uso de la librería dart:io permite que la información ingresada (como el texto .ªrchivo de texto en dart") se mantenga con vida útil en el proceso actual y se almacene físicamente en el disco.

Prueba de creacion de archivos de texto

```
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart run main.dart
============================
       MENÚ DE ARCHIVOS
============================
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 1
Nombre del archivo a crear (ej: notas.txt): texto

Escribe el texto que deseas guardar.
Para terminar, escribe SOLO: FIN
--------------------------------
Archivo de texto en dart FIN
FIN

Archivo creado y guardado correctamente.

============================
       MENÚ DE ARCHIVOS
============================
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 2
Ingresa el nombre o ruta del archivo a leer: texto

========== CONTENIDO DEL ARCHIVO ==========
Archivo de texto en dart FIN
==========================================

============================
       MENÚ DE ARCHIVOS
============================
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 3
Ingresa el nombre o ruta del archivo a sobrescribir: texto

SE ENCONTRÓ EL ARCHIVO.
¿Deseas sobrescribirlo? Esto borrará todo su contenido.
Escribe "SI" para confirmar: SI
```

A partir de los codigos de hilos realizamos el analisis planteado en el desarrollo.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Ejemplo1.dart
Inicio
Fin inmediato (sin esperar)
Tarea asíncrona completada
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Ejemplo2.dart
Inicio
Tarea completada con await
Fin
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Ejemplo3.dart
Hola desde otro isolate
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Ejemplo4.dart
Iniciando tarea pesada en hilo paralelo...
Mientras tanto, sigo ejecutando en el hilo principal...
Resultado: 124999999750000000
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Ejemplo5.dart
Main: recibí el SendPort del worker.
Main: respuesta del worker -> Recibido en worker: Mensaje desde main
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Ejemplo6.dart
Inicio
Resultado: 124999999750000000
Fin
PS C:\Users\gato2\OneDrive\Escritorio\DART>
```

En la ejecucion del codigo observamos la eficacia del patron de diseño de Singleton para que se pueda resolver un problema sin que se tenga que estar haciendo otro tipo de tareas en la que se requiera un constante paso de mensajes entre instancias.

```
PS C:\Users\gato2\OneDrive\Escritorio\DART> dart Singleton.dart
¿Es la misma instancia? true

Enviando a impresión: Documento #1 - "Reporte mensual" (Usuario: Alice)
Enviando a impresión: Documento #2 - "Contrato de servicio" (Usuario: Bob)
Enviando a impresión: Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
=== Cola de impresión pendiente ===
Documento #1 - "Reporte mensual" (Usuario: Alice)
Documento #2 - "Contrato de servicio" (Usuario: Bob)
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
==================================

--- Imprimiendo documento ---
Documento #1 - "Reporte mensual" (Usuario: Alice)
Contenido: Contenido del reporte mensual...
--- Fin de impresión ---

--- Imprimiendo documento ---
Documento #2 - "Contrato de servicio" (Usuario: Bob)
Contenido: Términos y condiciones del servicio...
--- Fin de impresión ---

=== Cola de impresión pendiente ===
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
==================================

--- Imprimiendo documento ---
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
Contenido: Diapositivas del proyecto X...
--- Fin de impresión ---

No hay documentos en la cola de impresión.
=== Historial de documentos impresos ===
Documento #1 - "Reporte mensual" (Usuario: Alice)
Documento #2 - "Contrato de servicio" (Usuario: Bob)
Documento #3 - "Presentación proyecto X" (Usuario: Carlos)
======================================

PS C:\Users\gato2\OneDrive\Escritorio\DART>
```

5. Conclusiones

El análisis del manejo de Archivos confirma que la utilización de flujos de datos (streams) es el mecanismo principal que permite transformar información de la memoria en datos que persistan. La correcta manipulación de la clase File y los métodos de entrada/salida garantiza que la información sobreviviera a la ejecución del programa, estableciendo un puente efectivo entre la lógica de la aplicación y el almacenamiento físico.
Sobre la optimización del rendimiento con Hilos se vio la distinción fundamental entre la concurrencia lógica y el paralelismo real. Mientras que el Event Loop y el uso de async/await demostraron ser eficientes para gestionar tareas diferidas sin bloquear el flujo principal , fue la implementación de Isolates la que resolvió el problema de los cuellos de botella en tareas computacionalmente pesadas. Los resultados obtenidos muestran cómo el aislamiento de memoria y la comunicación por puertos evitan que la interfaz o el hilo principal se congelen, lo cual quedó claro al compararlo con el ejemplo secuencial que detenía el programa.
Finalmente, la aplicación del Patrón de Diseño Singleton en el simulador de la cola de impresión nos mostró la importancia de controlar el acceso a recursos compartidos. Al asegurar que existiera una única instancia encargada de gestionar las solicitudes de impresión, se pudo centralizar el control y mantener los datos coherentes, demostrando que los patrones de diseño no son solo teoría, sino soluciones estandarizadas necesarias para evitar conflictos en la arquitectura del software.

6. Referencias bibliográficas

[1] Oracle. (s.f.). Lesson: Basic I/O (The Java™ Tutorials >Essential Java Classes). Oracle Help Center. https://docs.oracle.com/javase/tutorial/essential/io/

[2] Oracle. (s.f.). Lesson: Concurrency (The Java™ Tutorials >Essential Java Classes). Oracle Help Center. https://docs.oracle.com/javase/tutorial/essential/concurrency/

[3] Solano, J. A. (2020). Patrones de diseño. Unidades de Apoyo para el Aprendizaje CUAED/Facultad de Ingeniería-UNAM.
https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3061/mod$_r esource/content/1/UAPA - Patrones - Diseno/index.html$