	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): M.C. RENE ADRIAN DAVILA PEREZ

Asignatura: PROGRAMACIÓN ORIENTADA A OBJETOS

Grupo: 01

No de Práctica(s): 11,12,13

Integrante(s): 322118311

322094028

322092842

322078673

322067738

No. de lista o brigada: 03

Semestre: 2026-1

Fecha de entrega: 28/11/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

- **Planteamiento del problema.** El problema de esta práctica consiste en analizar aplicaciónes desarrolladas en Dart que contemplan los temas de manejo de archivos, uso de hilos y patrones de diseño.
- **Motivación.** El estudio de archivos, hilos y patrones de diseño esfundamental dentro de la programacion orientada a objetos, documentar estos elementos permite cómo un sistema administra datos externos, cómo ejecuta tareas concurrentes y cómo emplea soluciones arquitectónicas reutilizables.
- **Objetivos.** Los objetivos de esta práctica son:
  - Elaborar un diagrama UML estático que describa la estructura general de las clases y las relaciones existentes entre ellas.
  - Diseñar un diagrama UML dinámico que represente la interacción entre los componentes del sistema en un flujo de ejecución significativo.
  - Interpretar la aplicación de los conceptos de manejo de archivos, hilos y patrones de diseño dentro de los códigos analizado.

2. Marco Teórico

2.1. Archivos

Un archivo se puede considerar un objeto dentro de una computadora, el cual puede almacenar información y ademas puede ser manipulado como una identidad por el sistema operativo, estos requieren tener un nombre unico y una extensión. [1]

2.1.1. Flujo de Datos:

En lenguaje dart se utilizan los **Stream**, los cuales representan la conexión entre el programa(lectura) y la fuente(escritura).

2.1.2. Repositorio:

Se puede considerar como un almacen fisico y logico, en donde se almacenara la información registrada en Binario.[2]

2.2. Hilos de Ejecución:

Un hilo de ejecución es un unico flujo de ejecución dentro de un proceso, en otras palabras, es una secuencia de codigo dentro de un programa. Dart no usa hilos múltiples compartiendo memoria. En su lugar, utiliza un modelo llamado Isolates. [1]

2.2.1. Isolates:

Es una unidad de ejecución independiente con su propia memoria, tiene su propio heap y su propio event loop, lo que permite que un programa no existan condiciones de carrera. [1]

Etapas de Isolate:

1. Crear (spawn)
2. Ejecutar (running)
3. Comunicación (messaging)
4. Terminación (shutdown)

[1]

2.2.2. Event loop:

Es la manera en que el lenguaje dart maneja las tareas concurrentes, mediante algunos metodos como **Future, async, await y stream**, las tareas parecen ser ejecutadas de manera concurrrente, sin embargo todo funciona desde el mismo hilo de ejecucion. [1]

2.3. Patrones de diseño:

Un patrón de diseño es una solución general y reutilizable para resolver problemas recurrentes en el diseño de software. En otras palabras es un modelo (template) que ayuda a agilizar el proceso de solución al problema al organizar mejor las clases, objetos y sus interacciones, promoviendo buenas prácticas como modularidad, reutilización y mantenimiento del sistema. [1] **Niveles de patrones de software:**

- **Patrones arquitectónicos:** Definen la estructura general y organización de un sistema, es decir describen soluciones al mas alto nivel de software y hardware.
- **Patrones de diseño:** Proponen soluciones reutilizables a problemas a nivel medio de software
- **Patrones de programación:** Describen técnicas específicas para resolver tareas concretas a bajo nivel en el código (clases y metodos). [1]

2.3.1. Singleton

Una clase que implementa el patron de diseño "Singleton"garantiza que solo exista una única instancia en toda la aplicación y que todos los clientes accedan a ella mediante un punto de acceso común. Esto permite controlar que no se creen más instancias y asegura una vía centralizada para obtener el mismo objeto durante toda la ejecución. En el patrón Singleton se debe asegurar que la clase no pueda ser

instanciada libremente. Para lograrlo, su constructor debe ser privado, debe existir una única instancia declarada como privada y estática, y se debe proporcionar un método público y estático que permita acceder a esa instancia desde cualquier parte del programa. [1]

3. Desarrollo

3.1. Archivos

Para el analisis de este codigo notamos que inicia con la importación de la biblioteca dart:io, la cual permite trabajar con archivos y operaciones de entrada y salida dentro de la consola. la función principal main() presenta un menú interactivo con las opciones de crear archivo y escribir texto, leer archivo, sobrescribir archivo y salir. La función **crearYEscribirArchivo()** permitira crear un archivo .txt y escribir en él hasta que el usuario ingresa la palabra "FIN", demostrando el uso directo de la clase File y del método **writeAsStringSync** para almacenar información. La función **leerArchivoExistente()** emplea **readAsStringSync** para abrir y mostrar el de un archivo previamente creado(lectura). Finalmente, **sobrescribirArchivo()** ejemplifica cómo reemplazar por completo el contenido de un archivo ya existente mediante escritura síncrona, integrando validaciones de seguridad y confirmación del usuario.

3.1.1. Diagrama Estatico:

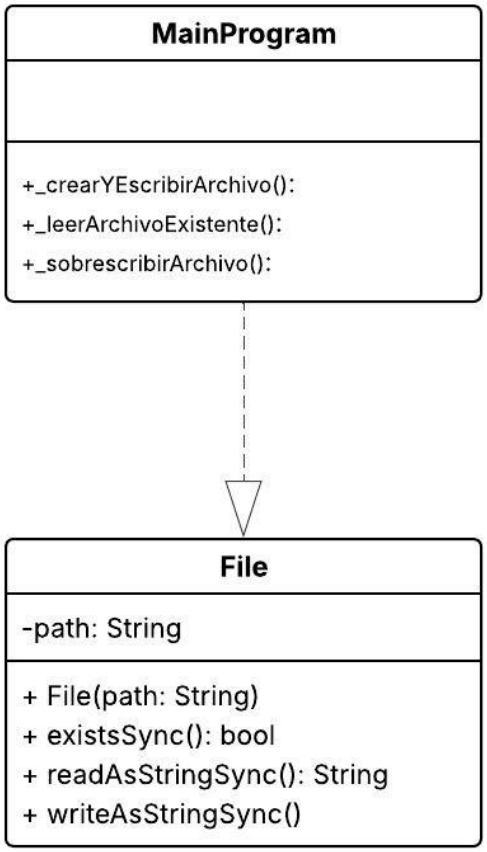

Figura 1: Diagrama de clases

3.1.2. Diagrama Dinamico:

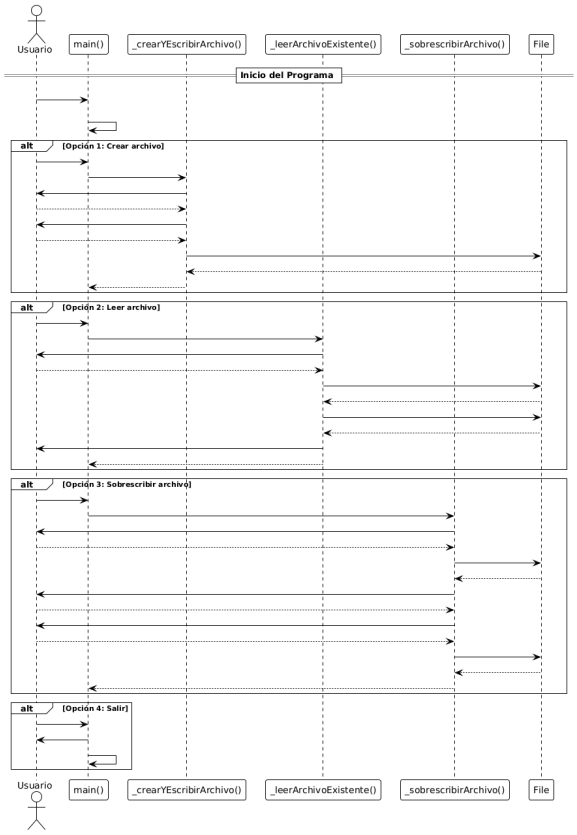

Figura 2: Diagrama de Secuencia

3.2. Hilos

Ejemplos de aplicacion de los Isolates de Dart mostrados en el video

1. **Ejemplo 1: Uso de `Future.delayed`** Este código inicia mostrando **Inicio**, programa una tarea asíncrona que se ejecutará dos segundos después, y finaliza inmediatamente sin esperar su resultado. Dart continúa su flujo principal mientras una tarea queda pendiente.

2. **Ejemplo 2: Espera con `await`** En este ejemplo, el programa se detiene hasta que la tarea asíncrona programada con `Future.delayed` termina. Aquí sí se espera la finalización antes de continuar con el resto del código.

3. **Ejemplo 3: Comunicación entre isolates** Este programa crea un nuevo isolate que envía un mensaje al isolate principal usando `SendPort`. ejemplifica cómo dos hilos independientes se comunican mediante mensajes.

4. **Ejemplo 4: Tarea pesada ejecutada en un isolate** Se ejecuta una operación larga (sumar hasta 500 millones) dentro de un isolate separado, lo cual

evita bloquear el hilo principal, que puede seguir ejecutándose mientras la tarea pesada se procesa en paralelo.

5. **Ejemplo 5: Envío y recepción de mensajes entre isolates** En este ejemplo el isolate secundario crea su propio puerto de escucha y envía su `SendPort` al hilo principal, permitiendo comunicación. El main envía un mensaje y recibe la respuesta del worker.

6. **Ejemplo 6: Tarea pesada en un solo hilo** Aquí una suma muy grande se ejecuta completamente en el hilo principal, bloqueándolo hasta terminar. Sirve para comparar cómo, sin isolates, el programa se detiene hasta completar la operación.

3.3. Patrones de diseño:

Este código inicia con la definición del modelo,la clase Pokemon, que encapsula los atributos de un pokemon dentro del combate, tales como su nombre, nivel, tipo elemental, vida y velocidad. La función constructor, incorpora el uso de la clase Random para generar estadísticas aleatorias que se basaran en el nivel. Posteriormente, se implementan clases hijas como PokemonFuego y PokemonHierba, las cuales heredan las características de Pokemon y especializan el atributo correspondiente a su tipo. Despues, se introduce el modelo de llos ataques mediante la clase Ataque y subclases (AtaqueFuego, AtaqueHierba y AtaqueNormal).

Después del modelo, el código presenta su **componente de vista**, como una interfaz llamada **CombateView**. Ahi estan las operaciones que se necesitan para mostrar la (imprimir los datos de cada Pokémon, mostrar ataques, indicar si un ataque es efectivo o poco efectivo, desplegar el daño realizado, anunciar desmayos y mostrar al ganador). La clase **ConsoleCombateView**es un implementacion de esta vista y se encarga de imprimir cada uno de los eventos del combate. Esta separación cumple con la estructura del patrón Modelo–Vista–Controlador (MVC). ya que aisla la presentación de la logica

El controlador del sistema es la clase **CombateController**, cuya función principal es coordinar el flujo del combate y coordinar la interacción entre el modelo (Pokémon y ataques) y la vista (la consola). El controlador determinara el orden de los turnos en base a la velocidad de los Pokémon, selecciona cuall ataca primero y administra el ciclo completo del combate hasta que uno de los participantes pierda la vida. Dentro del método privado **turno**, se establece la dinámica de intercambio de ataques entre atacante y defensor; mientras que en el método **atacar** calcula el daño considerando la efectividad del tipo del ataque frente al rival, aplicando multiplicadores de daño.

El programa culmina con la función main. Aquí se crea la vista concreta, se inicializa el controlador que gestionará el combate y se instancian dos Pokémon (Charmander y Bulbasaur), así como un ataque por defecto. A partir de estos elementos, se llama el método **iniciarCombate**, iniciando así todo el flujo controlado por la arquitectura MVC. el modelo concentra la información y reglas fundamentales de los objetos, la vista administra la forma de mostrar la información al usuario y el con-

trolador coordina la interacción entre ambos, tomando decisiones sobre la secuencia del combate.

3.3.1. Diagrama estatico:

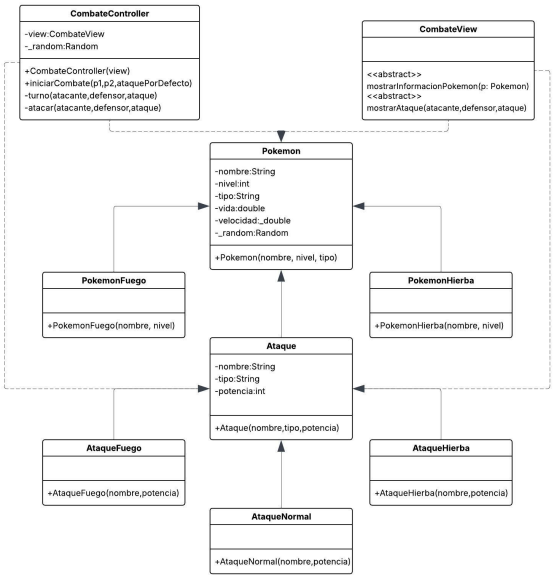

Figura 3: Diagrama de clases

3.3.2. Diagrama dinamico:

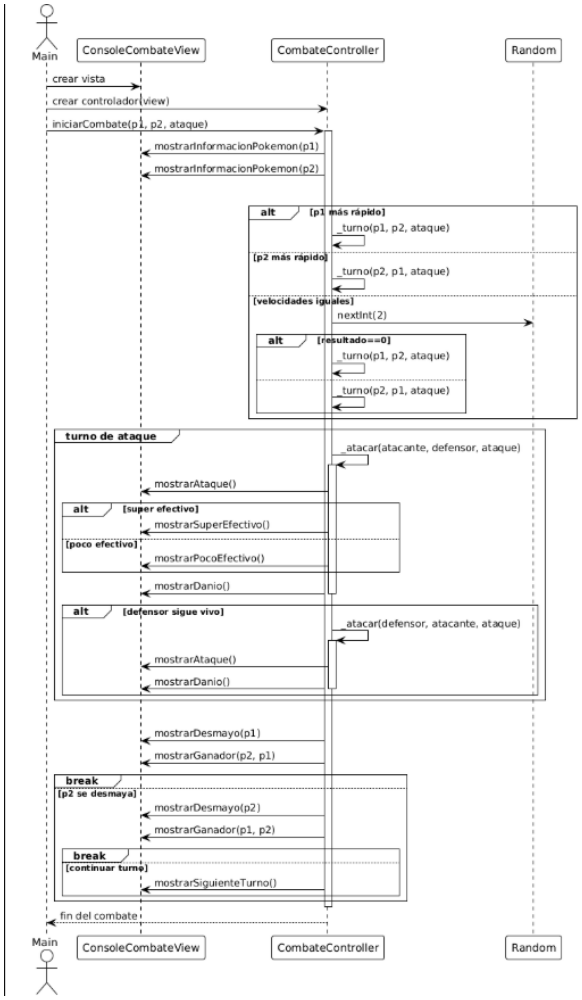

Figura 4: Diagrama de secuencia

4. Resultados

4.1. Archivos

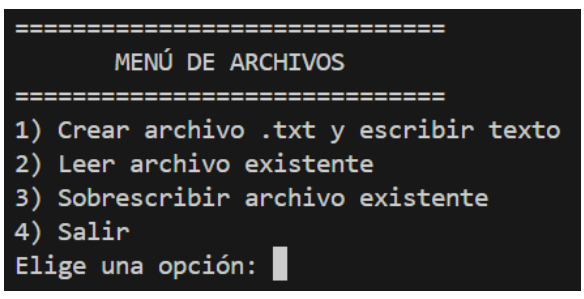

Figura 5: Menú

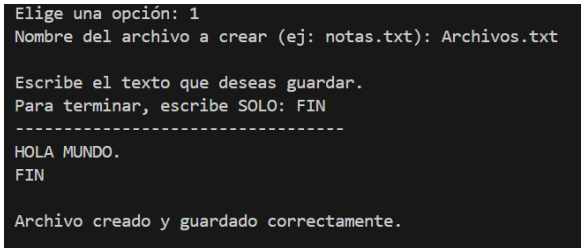

Figura 6: Salida de Opción 1

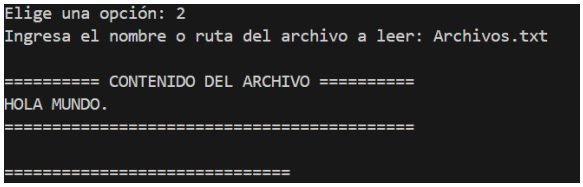

Figura 7: Salida de Opción 2

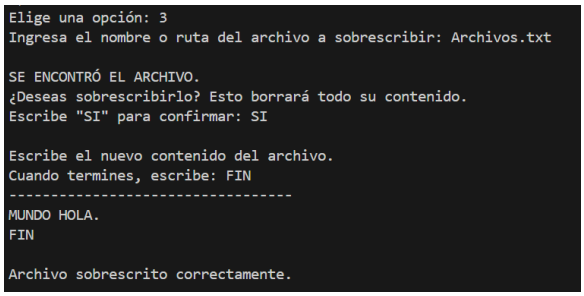

Figura 8: Salida de Opción 3

4.2. Patrones de diseño:

```
PS C:\Users\marti\downloads> dart patrones.dart
Nombre: Charmander
Nivel: 50
Tipo: Fuego
Vida: 82
Velocidad: 62
----------------------
Nombre: Bulbasaur
Nivel: 50
Tipo: Hierba
Vida: 154
Velocidad: 119
----------------------
========== COMBATE INICIADO ==========
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
Charmander ataca a Bulbasaur con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
-- Siguiente turno --
Bulbasaur ataca a Charmander con Tacleada!
El Pokémon rival recibe 35.0 puntos de daño.
¡Charmander se ha desmayado!
¡Charmander ha sido derrotado!, ¡Bulbasaur gana el combate!
========== COMBATE TERMINADO ==========
```

Figura 9: Combate

5. Conclusiones

Analizar los programas desarrollados en Dart nos permite concluir que la integración de los conceptos de manejo de archivos, hilos y patrones de diseño es importante para desarrollar aplicaciones capaces de responder de manera eficiente a distintas necesidades del sistema. El uso correcto de archivos asegura la persistencia y consistencia de la información, mientras que la implementación de hilos permite ejecutar tareas concurrentes sin bloquear el flujo principal del programa, optimizando así el rendimiento de nuestros proyectos y aumentando la eficiencia.

Asimismo, la aplicación de patrones de diseño refuerza la estructura del sistema al promover soluciones reutilizables, y alineadas con los principios de la Programación Orientada a Objetos. En conjunto, estos elementos nos permiten construir aplicaciones más organizadas.

6. Bibliografía

- **[1]** R. A. Dávila, Programación Orientada a Objetos – Serie de videos sobre Archivos, Hilos e Isolates en Dart, Facultad de Ingeniería, Universidad Nacional Autónoma de México (UNAM). Material audiovisual proporcionado en el curso, 2025.
- **[2]** Google LLC, "Dart Language Tour,"dart.dev, 2025. Accedido: Nov. 2025. Disponible en: https://dart.dev/language