	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación
Salas A y B

Profesor(a): _____

Asignatura: _____

Grupo: _____

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

Integrante(s): 322246320

322267567

322012051

322236688

322330872

No. de lista o brigada: _____

Semestre _____

Fecha de entrega: _____

Observaciones: _____

_____

CALIFICACIÓN: _____

Índice

1. Introducción

1.1. Planteamiento del Problema:

Se busca la implementación, comprensión y explicación de los temas de Archivos, Hilos y Patrones en Dart, mediante el análisis de múltiples códigos para cada tema.

1.2. Motivación:

En esta práctica aprenderemos sobre la correcta aplicación de los temas anteriormente mencionados en programas de Dart para la resolución de problemas con mayor eficiencia.

1.3. Objetivos:

Crear programas en Dart que aprovechen de forma sencilla y práctica el uso de archivos, hilos y patrones de diseño, para que la aplicación funcione de manera fluida, organizada y pueda reaccionar bien incluso cuando ocurran tareas simultáneas o situaciones inesperadas.

2. Marco Teórico

2.1. Archivos:

Los archivos, en términos computacionales, son conjuntos de datos con un nombre y extensión que permite almacenar información de manera permanente en memoria secundaria, para que pueda ser consultada, utilizada y modificada posteriormente tanto por el usuario como diferentes programas.

Los programas se comunican con los archivos mediante *flujos de datos*. Estos flujos se clasifican en **flujos de entrada** y **flujos de salida**.

- **Flujos de entrada:** cuando los programas reciben información desde archivos o una fuente externa.
- **Flujos de salida:** cuando los programas envían esa información a un destino externo, como lo puede ser el usuario.

[1] [2]

2.2. Hilos:

Un hilo es la unidad mínima de ejecución dentro de un proceso, dichos hilos contienen una sucesión de instrucciones que se ejecutan de manera independiente pero no es totalmente autónomo del proceso. En el caso de Dart, los hilos no se comunican entre sí directamente ya que no comparten memoria, éstos se comunican mediante **SendPort** y **ReceivePort**, los cuáles enviarán y recibirán mensajes respectivamente. [3]

2.3. Patrones:

Los patrones de diseño son soluciones aprobadas y documentadas para problemas comunes en el desarrollo de software. Son guías o modelos que muestran cómo estructurar clases, objetos y responsabilidades dentro de un programa para que sea más fácil de mantener, extender y comprender. Su principal propósito es mejorar la calidad del software. Se considerar tres tipos de patrones de software:

- **Patrones arquitectónicos:** Describen soluciones al mas alto nivel de software y hardware. Normalmente soportan requerimientos no funcionales.
- **Patrones de diseño:** Describen soluciones en un nivel medio de estructuras de software. Normalmente soportan requerimientos funcionales
- **Patrones de programación:** Describen soluciones en un nivel medio de estructuras de software. Normalmente soportan requerimientos funcionales.

[4]

Patrón Modelo-Vista-Controlador: También llamado MVC, es un patrón de arquitectura de software en el cual se utilizan tres componentes, separando la lógica de la aplicación de la lógica de la vista, algo muy útil ya que permite realizar cambios en parte de la aplicación sin afectar a las demás, encargándose el modelo de cuestiones como manejo de datos, generalmente, aunque no obligatoriamente, consultando bases de datos para actualizaciones, consultas, búsquedas, etc.
EL controlador es quien se encarga de procesar las solicitudes del usuario, pedirle los datos correspondientes al modelo y de comunicárselos a vista; vista se encarga de la representación visual de los datos, encargándose de toda la interfaz gráfica, no preocupándose el modelo ni el controlador por lo que sucede en esta parte. [5]

3. Desarrollo

3.1. Explicación del código de archivos:

El código implementado en dart **main.dart** consta de un menú de opciones que nos permite entender y trabajar con las operaciones básicas de los archivos. El menú cuenta con 3 opciones que explicaremos a continuación y la opción salir. Cada opción manda a llamar a una función que permite realizar la operación ingresada.

Opción 1. Crear archivo .txt y escribir texto: Esta función le pide al usuario que ingrese un nombre para crear un archivo, si la cadena ingresada no esta vacía, crea ese nuevo archivo. Al crear el archivo, se le pide al usuario que ingrese el texto que contendrá el archivo, y para indicar que ya termino de escribir lo que se quiere guardar en el archivo, que escriba 'FIN'. El programa, mediante un ciclo *while*, va almacenando las cadenas ingresadas por el usuario en una lista, y si se detecta una cadena 'FIN', se termina el ciclo. Finalmente con un *try-catch*, se intentará crear un archivo con el nombre ingresado inicialmente por el usuario (creando un objeto

de tipo **FILE** e indicando su nombre en el constructor), y con el método **writeAsStringSync** se escriben las cadenas guardadas en la lista de cadenas, dentro del archivo creado. En caso de no poder crear o escribir en el archivo, el bloque *catch* muestra que ocurrio un error al guardar el archivo.

Opción 2. Leer archivo existente: Esta función le pide al usuario que ingrese el nombre o ruta del archivo que desea leer, si el nombre o ruta ingresado no se encuentra se termina esta funcion. Si la ejecución continua, con un *try-catch* se intentará abrir un archivo con el nombre o ruta (indicándolo en el constructor), y leerlo con el método **readAsStringSync**, para posteriormente imprimirlo.

Opción 3. Sobrescribir archivo existente: Esta función le pide al usuario que ingrese el nombre o ruta del archivo a sobrescribir, si el nombre o ruta no se encuentra, se termina la función. Si el archivo se encuentra, se abre el archivo y se le pide confirmación al usuario para sobrescribir el archivo, indicándole que esta acción borrara todo su contenido. En caso de aceptar la sobrescritura, de la misma manera que la primera opción, se le pide al usuario que ingrese el texto y para terminar escriba 'FIN', y con un ciclo while se almacenan las cadenas en una lista, para que finalmente con un *try-catch*, se intente sobrescribir el archivo con el método **writeAsStringSync**, agregando las cadenas guardadas en la lista al archivo.

3.2. Explicación de los códigos de Hilos:

En el **Ejemplo1.dart** se comienza creando el método asíncrono main, dentro del mismo se manda a imprimir 'Inicio', después se crea una tarea asíncrona la cuál tendrá un retraso de 2 segundos pero no se interrumpirá el flujo del programa sino que este sigue hasta su final imprimiendo 'Fin inmediato (sin esperar)', lo cuál nos indica que el programa no ha esperado a que la tarea retrasada termine para que el programa concluya, dos segundos después de la ejecución finalizará la tarea con retraso y se imprime 'Tarea asíncrona completada'.

En **Ejemplo2.dart** tenemos la misma base que en el ejemplo anterior, pero en esta ocasión a la tarea asíncrona se le añade la palabra **await** al inicio de la línea. Con esto es conseguido que al ejecutar el programa, se inicie **main** pero no seguirá hasta el final, sino que el flujo se ve interrumpido hasta que la tarea asíncrona termine (2 segundos) para posteriormente terminar la ejecución.

En **Ejemplo3.dart** Comienza importando la librería **dart:isolate** que permite crear y manejar isolates. Luego se define el método tarea que enviará el mensaje con **SendPort**, posteriormente en main, se crea un **ReceivePort** que recibirá el mensaje del isolate tarea ya definido. Después se crea un isolate principal con **isolate.spawn**, al cuál se le pasará el **SendPort** asociado al **ReceivePort** para finalmente hacer que nuestro isolate principal escuche los mensajes con **ReceivePort.listen** y después los mande a imprimir.

En **Ejemplo4.dart** nuevamente se utiliza la librería **dart:isolate**, después se define la función **sumaGrande**, en esta se hace la suma de los números desde 0 hasta 500 millones y se envía el resultado al isolate principal en main. Por su lado, en main se crean el ReceivePort y el isolate principal al cuál se le pasa el total de la suma con **ReceivePort.listen** y se manda a imprimir, finalmente añaden algunas impresiones para indicar al usuario que está pasando al ejecutar.

En **Ejemplo5.dart** se implementa una comunicación bidireccional más compleja entre hilos. Primero se define la función **worker**, la cual crea su propio **ReceivePort** interno y envía su dirección (**SendPort**) al hilo principal, permitiendo así que el main sepa a dónde enviarle mensajes. En **main**, al escuchar a través de su **ReceivePort**, se distingue si el mensaje recibido es un puerto (para establecer la conexión) o una cadena de texto (una respuesta). Una vez que **main** obtiene el puerto worker, le envía un mensaje; el worker lo procesa, devuelve una respuesta y finalmente el programa principal cierra los puertos y termina el isolate con **isolate.kill**.

En **Ejemplo6.dart** se realiza la misma operación matemática que en el ejemplo 4, pero esta vez de forma síncrona y sin utilizar la librería **dart:isolate**. El programa inicia imprimiendo 'Inicio' y procede a ejecutar el ciclo **for** de 500 millones de iteraciones directamente en el hilo principal. Esto provoca que el flujo de ejecución se bloquee totalmente esperando a que termine el cálculo; solo cuando la suma ha finalizado se imprime el 'Resultado' y posteriormente 'Fin', demostrando cómo una tarea pesada puede congelar la ejecución si no se delega a un hilo paralelo.

3.3. Explicación del código de patrones:

Considerando **main.dart** como el código principal proporcionado, este código implementa un patrón Modelo-Vista-Controlador(MVC), las clases que funcionan como Modelo son los pokemón y sus ataques, teniendo primero una clase *pokemon* en la cual solo se establecen atributos como el nombre, algunas estadísticas y el tipo, teniendo unicamente un constructor el cual establece la vida, nivel y tipo a los dados, mientras que para la vida y velocidad utiliza un número aleatorio; luego se crean dos clases que heredan, para los pokemón tipo fuego y hierba que utilizan el constructor de la clase padre cambiando unicamente el tipo.
Luego se tiene la clase *ataque*, la cual cuenta con el nombre del ataque, la potencia y el tipo, contando con un constructor que solo relaciona los valores dados con su respectivo atributo, y al igual que con la clase *pokemon*, se tienen tres clases que heredan para movimientos tipo fuego, normal y hierba, las cuales usan el constructor del padre cambiando unicamente el tipo de ataque.

Para la parte de Vista, se tiene primero una clase abstracta que se utilizara de interfaz, en donde se establece que se tendrán los métodos *mostrarInformcacion-Pokemon, mostrarAtaque, mostrarSuperEfectivo, mostrarPocoEfectivo, mostrarDanio, mostrarDesmayo, mostrarSiguienteTurno, mostrarGanador*, ya dentro de la clase

ConsoleCombateView se implementa la interfaz, y se escribe lo que mostrara cada método en consola, accediendo a los datos de los pokemón creados y en combate, de ser necesario.

Para la parte del controlador, se tiene una clase *CombatController*, la cual se encarga de manejar la lógica de los ataques, de los turnos, y de lo que se mostrara en pantalla, usando sobretodo un objeto de la clase padre de vista *CombateView*, siendo realmente impresiones sencillas de puro texto las que realiza por si sola la clase, y estableciendo el objeto según el tipo de objeto hijo de *CombateView* que se pase al crear el objeto *CombateController*, aunque manejando eventos como los ataques a usar sin intervención del usuario, por el momento.

Por ultimo en un método *main*, ya fuera de cualquier clase, se representa lo que podría ser una interacción con el usuario, creando un objeto *ConsoleCombateView* para dárselo a un objeto de clase *CombateController*, se crean dos pokemón diferentes, un ataque y se inicializa un combate con el método *iniciarCombate* del controlador creado, dándole como datos los pokemón creados y el ataque.

También se nos proporciona el código **singleton.dart**. Este programa nos muestra una plantilla para crear el sistema de impresión de una impresora. Cuenta con la clase **Documento** que contendra los atributos de un archivo a imprimir: *id, usuario, nombre, contenido*. Esta clase sobrescribe el método **toString** para darle un formato bonito a la impresión de los atributos.

Luego cuenta con la clase **Impresora** implementando el diseño de singleton, que se caracteriza por tener una instancia estática, un constructor privado, y un factory que devuelve siempre la misma instancia.

La clase **Impresora**, ademas de las características anteriores, cuenta con una cola para representar la cola de impresión, y una lista para el historial de documentos impresos. Así como los métodos **enviarDocumento**, para agregar un documento a la cola de impresión; **imprimirSiguiente**, para imprimir el siguiente elemento de la cola de impresión con un formato especifico; **mostrarCola**, para imprimir los elementos de la cola con un formato especifico; y **mostrarHistorial**, para mostrar el historial de documentos impresos con un formato especifico.

En el main, se crean 2 impresoras con ayuda del patrón de diseño, y se agregan algunos documentos a ambas impresoras, para posteriormente utilizar los métodos anteriores para mostrar la cola de impresión, el historial de documentos impresos, e imprimir los documentos.

4. Diagramas UML

4.1. Diagramas del código de archivos:

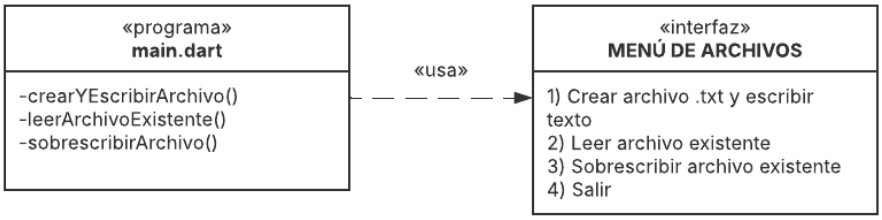


Diagrama UML de Clases

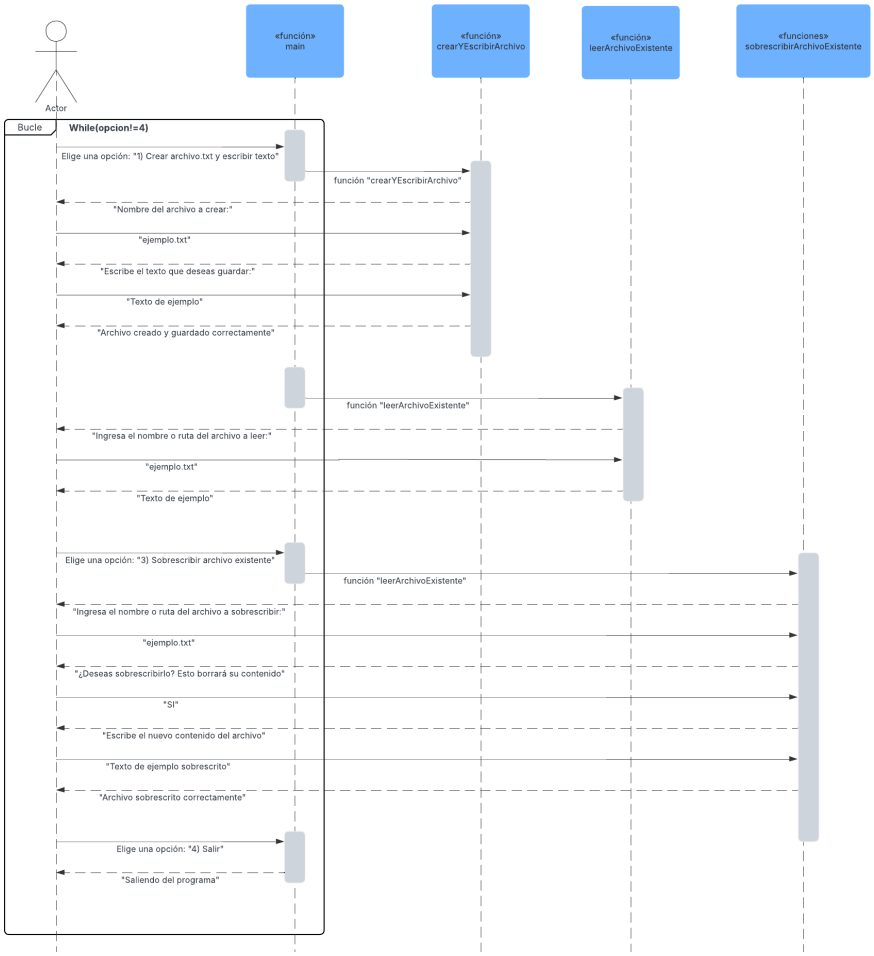


Diagrama UML de Secuencia

4.2. Diagramas de los códigos de Hilos:

Algunos UML de los ejemplos:

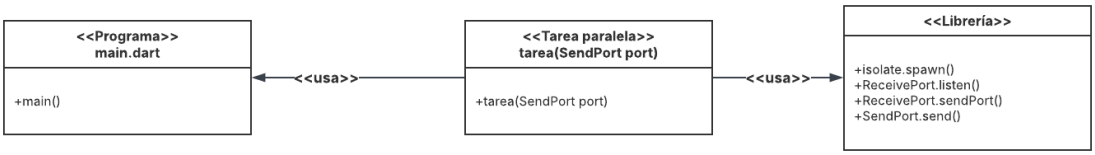


Diagrama UML estático del Ejemplo 3

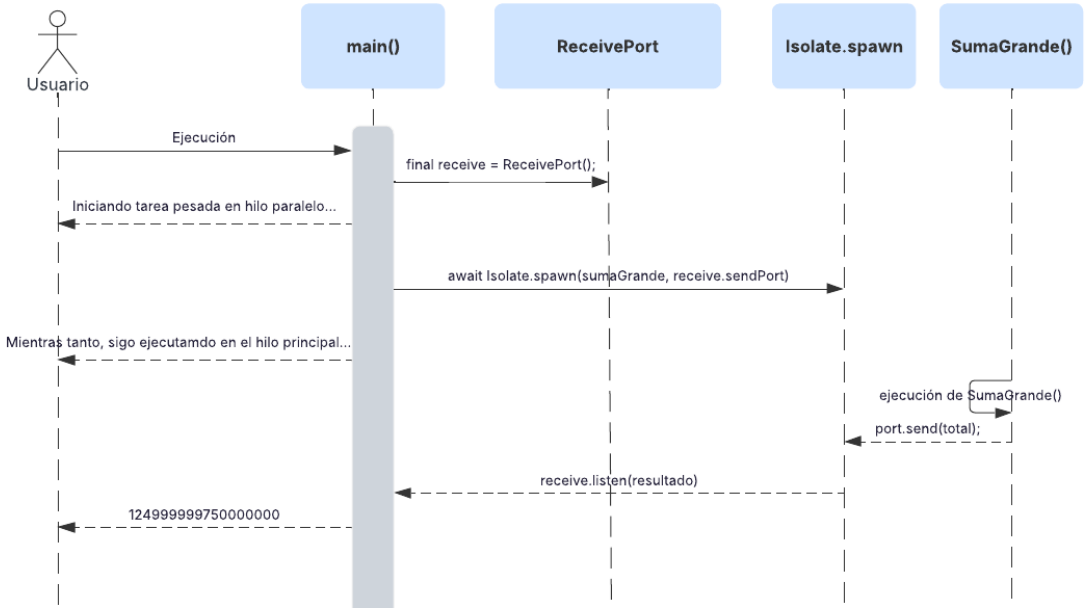


Diagrama UML dinámico del Ejemplo 3

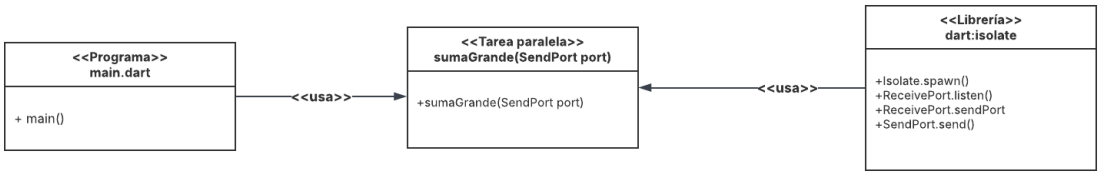


Diagrama UML estático del Ejemplo 4

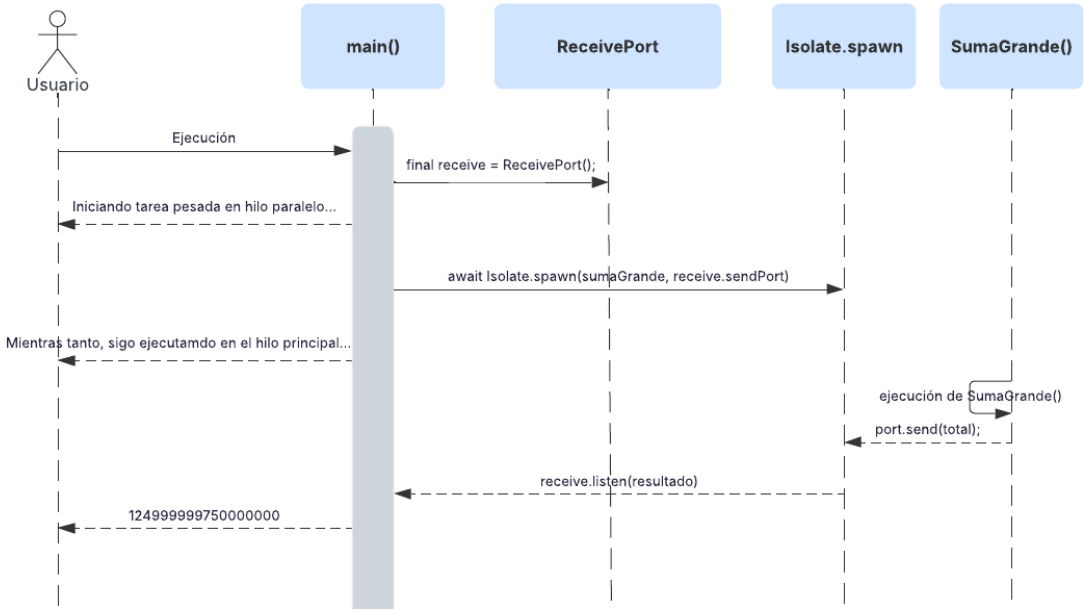


Diagrama UML dinámico del Ejemplo 4

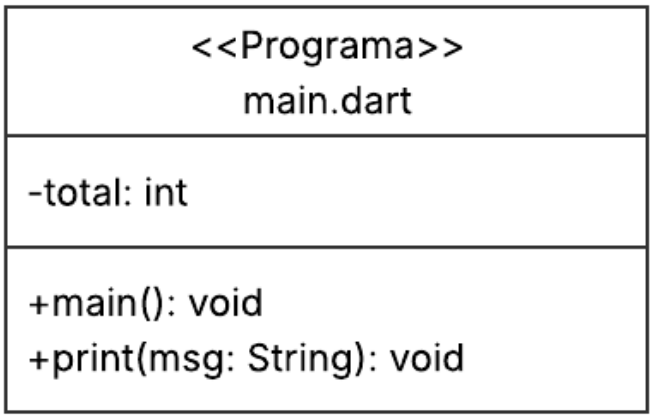


Diagrama UML estático del Ejemplo 6

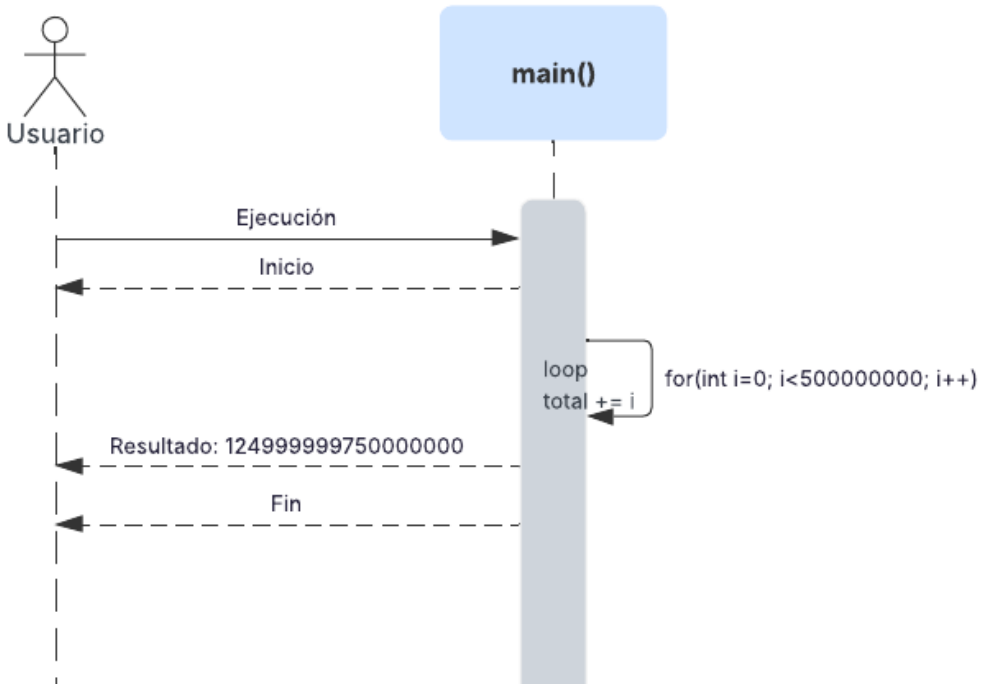


Diagrama UML dinámico del Ejemplo 6

4.3. Diagramas de código de patrones de diseño

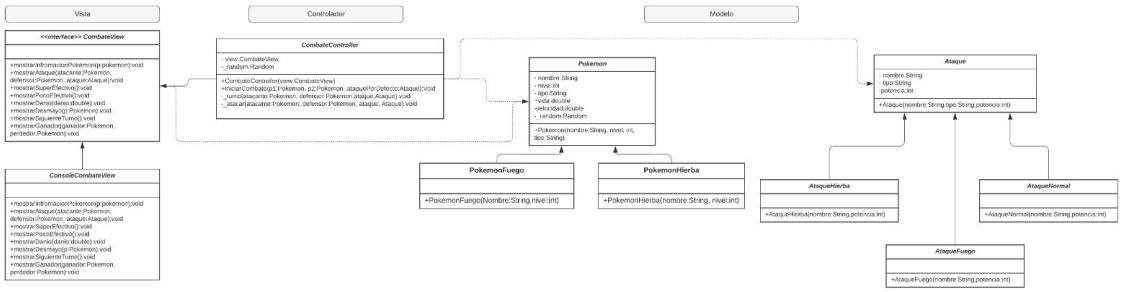


Diagrama UML estático del código de patrón MVC

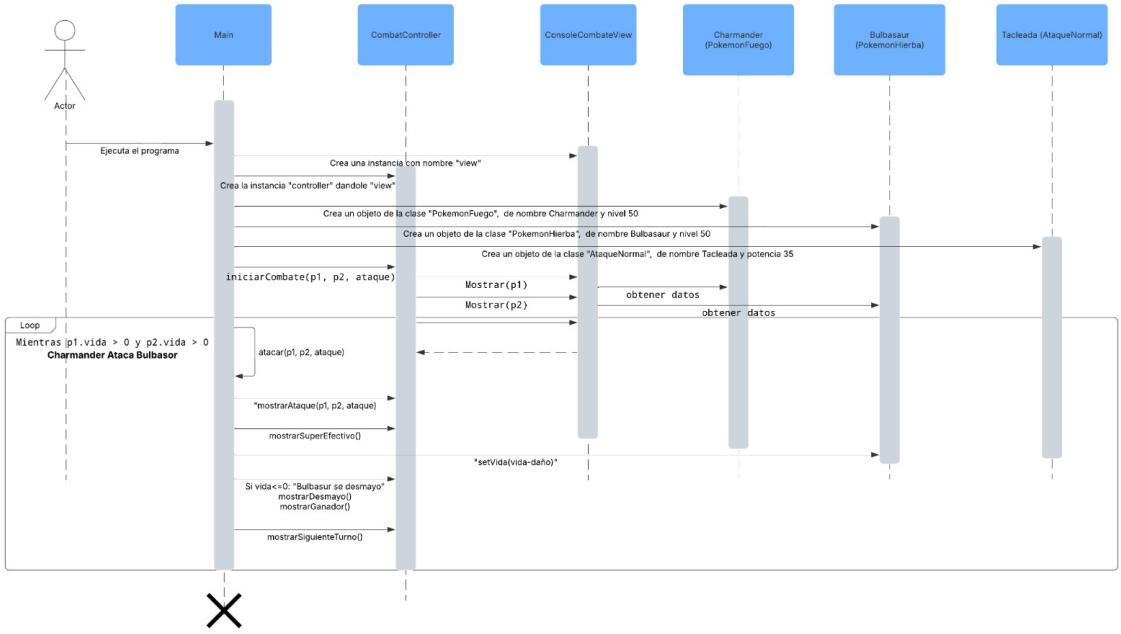


Diagrama UML dinámico del código de patrón MVC

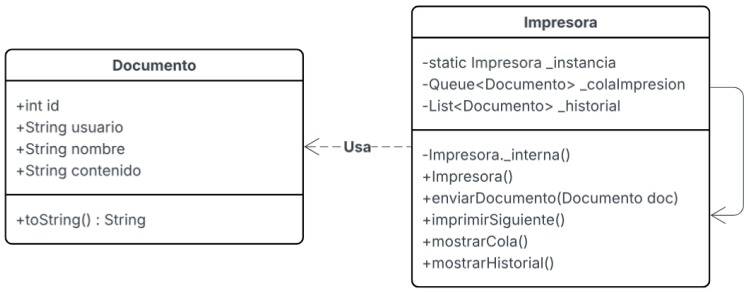


Diagrama UML estático del código Singleton

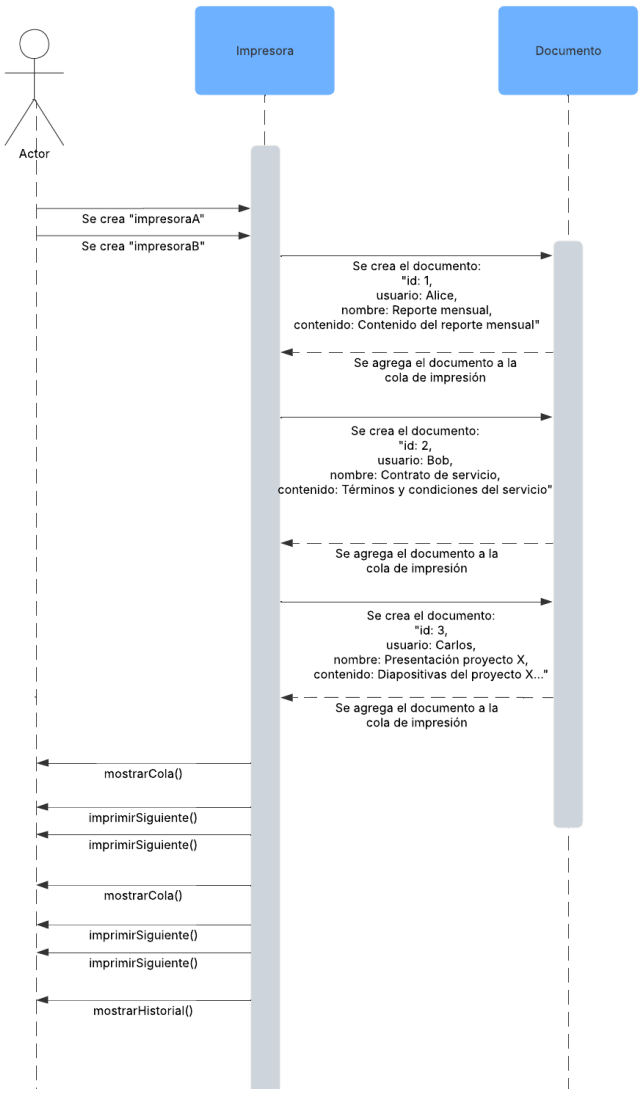
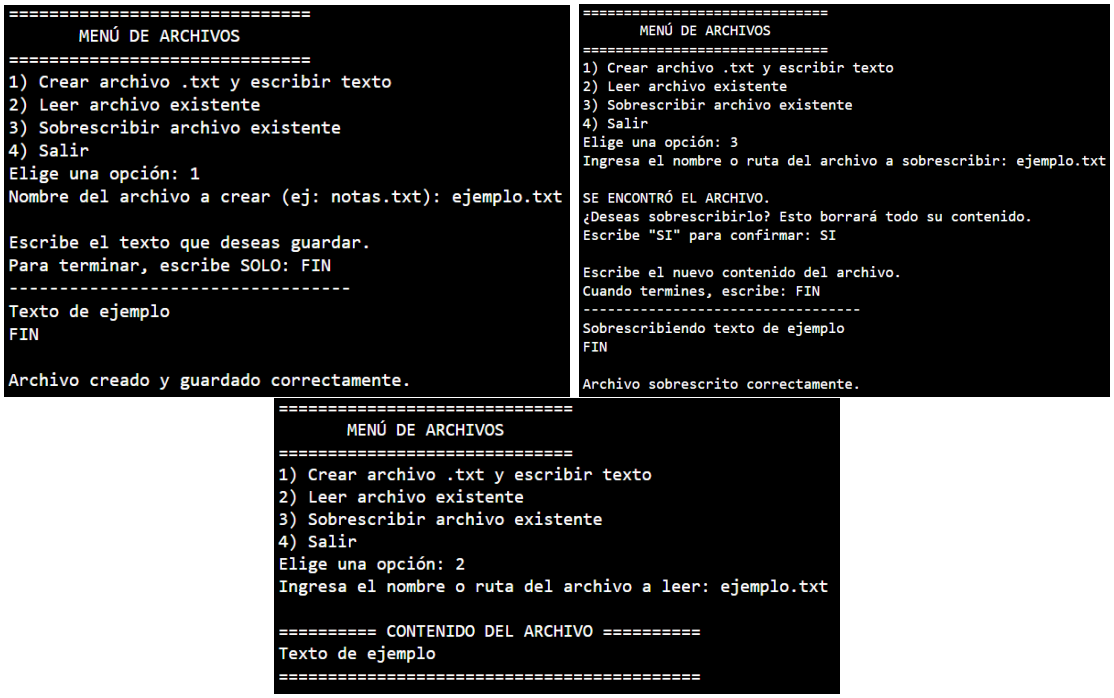


Diagrama UML dinámico del código Singleton

5. Resultados

5.1. Resultados del código de archivos



Ejecución del código de archivos

5.2. Resultados del código de hilos

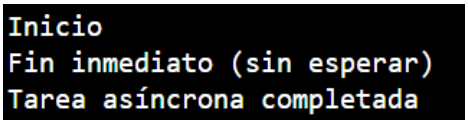
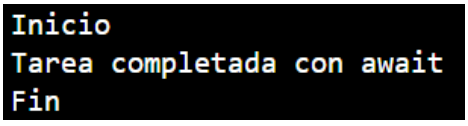
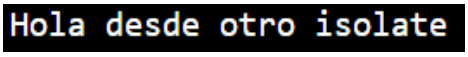


Figura 1: Ëjemplo 1



Ejemplo 2



Ejemplo 3

```
Iniciando tarea pesada en hilo paralelo...
Mientras tanto, sigo ejecutando en el hilo principal...
Resultado: 124999999750000000
```

Ejemplo 4

```
Main: recibí el SendPort del worker.
Main: respuesta del worker -> Recibido en worker: Mensaje desde main
```

Ejemplo 5

```
Inicio
Resultado: 124999999750000000
Fin
```

Ejemplo 6

5.3. Resultados del código de patrón

```
maximogab@maximogab-Inspiron-3576:~/POO/Practica11-13$ dart main.dart
Nombre: Charmander
Nivel: 50
Tipo: Fuego
Vida: 159
Velocidad: 116
---------------------
Nombre: Bulbasaur
Nivel: 50
Tipo: Hierba
Vida: 106
Velocidad: 197
---------------------
========== 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 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.
¡Bulbasaur se ha desmayado!
¡Bulbasaur ha sido derrotado!, ¡Charmander gana el combate!
========== COMBATE TERMINADO ==========
```

Ejecución del patrón MVC

```
¿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)
========================================
```

Ejecución del patrón de Singleton

6. Conclusiones

En esta práctica logramos comprender cómo trabajar de manera correcta con archivos, para almacenar información que puede ser modificada y consultada posteriormente; así como hilos de ejecución, que permiten la realización de diferentes tareas al mismo tiempo (de manera paralela); y patrones de diseño en Dart, que nos permiten ajustar una plantilla para ser utilizada para un mismo tipo de problema, cambiando sus caracteristicas pero manteniendo una misma estructura. Identificamos la forma en como cada uno de estos elementos son esenciales para así construir aplicaciones más eficientes, organizadas y preparadas para situaciones complejas.

Referencias

[1] J. A. Sandoval Acosta. *Programación Orientada a Objetos -Unidad 6 archivos.* slideshare. URL: https://es.slideshare.net/slideshow/programacin-orientada-a-objetos-unidad-6-archivos/77441967 (visitado 27-11-2025).

[2] D. Pérez Pérez. *6 Flujos y Archivos.* Blogger. 2012-06-04. URL: https://pooitsavlerdo.blogspot.com/2012/06/6-flujos-y-archivos.html (visitado 27-11-2025).

[3] Google Translator. *Concurrencia en Dart.* Google. URL: https://dart-dev.translate.goog/language/concurrency?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc (visitado 27-11-2025).

[4] Facultad de Ingeniería-UNAM. *Patrones de diseño.* UNAM. URL: https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3061/mod_resource/content/1/UAPA-Patrones-Diseno/index.html (visitado 27-11-2025).

[5] Uriel Hernandez. *MVC (Model, View, Controller) explicado.* UNAM. URL: https://codigofacilito.com/articulos/mvc-model-view-controller-explicado (visitado 27-11-2025).