	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios 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): 320065570
425133840
423020252
322229916
425032224

No. de lista o brigada: 3

Semestre: 2026-1

Fecha de entrega: 28/11/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

En la implementación de códigos en el paradigma orientado a objetos destacan algunos mecanismos que aseguran el buen funcionamiento de todos nuestros proyectos, entre estos destaca el manejo de excepciones, que constituye la base para construir sistemas resilientes. Las excepciones nos permiten anticipar y gestionar lo inesperado evitando que fallos localizados derriben toda la aplicación y permitiendo, en cambio, una degradación controlada o una recuperación efectiva.

Por otro lado, tenemos los hilos de ejecución, que permiten que diferentes tareas avancen simultáneamente, aprovechando al máximo la capacidad de cálculo disponible.

Mientras las excepciones gestionan los contratiempos y los hilos optimizan el rendimiento, el trabajo con archivos asegura que nuestra información sobreviva más allá del ciclo de vida del programa.

Finalmente, para orquestar armoniosamente todos estos elementos, se recurre a los patrones de diseño, que son soluciones consolidadas a problemas recurrentes de arquitectura. Estos patrones no solo proporcionan un lenguaje común entre desarrolladores, sino que establecen estructuras probadas que hacen que el código sea más mantenible, flexible y preparado para evolucionar.

Estos términos se desarrollarán en este trabajo, a partir de un análisis a los videos proporcionados por el profesor los cuales trataron justamente estos ya mencionados.

2. Marco Teórico

Para este análisis se recurrió, en primera instancia, al término de excepciones en el paradigma orientado a objetos. Con excepciones nos referimos a eventos anómalos que pueden ocurrir durante la ejecución de un programa y que pueden repercutir en este, alterando el flujo normal de su ejecución. estos errores deben ser manejados para garantizar el correcto funcionamiento del programa.

Tener conocimiento sobre las excepciones es importante, ya que, con ello, podemos tener un manejo de errores de manera estructurada, además de anticipar y gestionar situaciones excepcionales de forma adecuada.

Igualmente, se le puede dar retroalimentación al usuario para que este pueda observar el error y, posteriormente, resolverlo. además, el manejo adecuado de excepciones permite a los desarrolladores adaptar el comportamiento de la aplicación según las circunstancias. [1]

En este contexto de las excepciones, debemos tener en claro la regla de Pareto, que nos dice que el 80 % de las consecuencias proviene del 20 % por ciento de las causas, en este caso serian los errores que tengamos en el código, por lo cual, es importante encontrar este 20 % generador de problemas.

Igualmente, es importante señalar los tipos de errores a los que nos podemos enfrentar. en ellos, vamos a encontrarnos con errores semánticos, que son aquellos que se generan en el momento en que infringimos las normas de escritura de algún lenguaje. También tenemos los errores semánticos, que ocurren cuando el código tiene una sintaxis correcta pero presenta errores lógicos, ya que el significado que buscamos no es correcto. Por último, tenemos los errores de ejecución, los cuales ocurren cuando el programa se está ejecutando y corresponden a usos no considerados en la aplicación que generan errores, entre otros casos.

Por parte del tema de archivos, las aplicaciones pueden comunicarse con su entorno para obtener y otorgar información. en este sentido, el manejo de archivos se realiza a través de un flujo de datos en el que existe un flujo de entrada y un flujo de salida, los cuales son la fuente y el repositorio, respectivamente. Un repositorio se entiende como un

almacén, el cual puede ser de carácter físico o lógico.

Cuando nos referimos a un archivo, lo entendemos como un objeto dentro de la computadora el cual puede almacenar información, configuraciones o comandos, el cual puede ser modificado por cualquier sistema operativo, programa o aplicación. Un archivo tiene un nombre que lo caracteriza y una extensión que permite reconocer el tipo de archivo.

Dentro del tema de los hilos, tenemos que por lo general en una aplicación, las tareas se realizan de forma secuencial, es decir, los procesos se ejecutan uno después de otro, esto hace que si alguno de estos procesos tarda demasiado se provoque una fila de espera con todos los procesos que le siguen, causando una especie de "cuello de botella". Por esta razón surge la necesidad de tener varios flujos de ejecución para poder realizar una tarea sin la necesidad de esperar la culminación de otras.

Estos flujos se realizan en hilos, donde un hilo es un código en ejecución dentro de un proceso, es decir, un subproceso. Mientras que un proceso es un programa en ejecución dentro de su propio espacio de direcciones, es decir, una aplicación que se ejecuta en el sistema operativo. Un hilo no puede ejecutarse solo, requiere la supervisión de un proceso; por ello su ejecución está controlada por el contexto de un proceso donde se ejecuta. [3]

Por último, tenemos el tema de los patrones, entendemos un patrón como una solución general, reutilizable y aplicable a diferentes problemas en el desarrollo de software. Estos patrones proporcionan un enfoque estructurado y reutilizable para resolver situaciones recurrentes en el desarrollo de software, además ayudan a los desarrolladores a comunicarse y compartir soluciones eficientes y efectivas que han demostrado ser exitosas en el pasado. [2]

Podemos encontrar 3 niveles de patrones de software:

- Patrones arquitectónicos: Describen soluciones al más alto nivel de software y hardware.
- Patrones de diseño: Describen soluciones en un nivel medio de estructuras de software.
- Patrones de programación: describen soluciones al nivel más bajo de software, al nivel de clases y métodos.

3. Desarrollo

3.1. Archivos

Como se vio en la explicación del tema de archivos, retomaremos el concepto de este, que es un objeto en una computadora que puede almacenar información, configuraciones o comandos, el cual puede ser manipulado como una entidad por el sistema operativo, programa o aplicación. A continuación se explicará el código proporcionado por el profesor y su interpretación en los conceptos aplicados en el código.

El primer paso fue importar la biblioteca `dart:io` con el fin de poder hacer operaciones con archivos, y después se hizo un menú para poder realizar las siguientes operaciones:

1) crear un archivo txt y escribir sobre este, 2) leer un archivo existente, 3) sobreescribir en un archivo existente y 4) salir.

En la creación del menú usamos `stdin.readlineSync()` con la intención de leer el valor que el usuario introduzca y eso pasarlo a un valor entero.

Para la opción 1 de crear y escribir sobre un archivo, se uso `stdout.write` y en ese momento, ya esta activa la opción de escritura lo que nos permite ya crear el archivo. Lo primero que se hizo fue verificar que nuestro archivo no estuviera vacío. se dan instrucciones de escribir el texto que se quiere guardar y usamos nuevamente el método stdin.readlineSync() para leer lo que el usuario quiere guarda; es importante resaltar que para la opción 1 del menú, nuestra fuente de datos(lectura) será lo que ingrese el usuario y que haya leído el método. Continuará así nuestro programa hasta que el valor de nuestra linea sea nulo o escribamos la palabra "FINz así se escribirá linea a linea y esta será nuestra fuente de datos(escritura de datos). finalmente se lanza la excepción en el caso de que haya ocurrido un error con la creación de nuestro archivo, y si no es el caso; este se crea correctamente.

Para la opción 2 de leer un archivo existente se hizo uso de stdout.write para ingresar la ruta del archivo a leer y stdin.readLineSync() para leer lo que se ingrese y con un ciclo if se implementó el método `existsSync()` para verificar si existe o no el archivo, en el caso de que no exista se imprimirá que el archivo no existe, posteriormente se utilizó `readAsStringSync()` para leer todo el contenido del archivo y hacemos una excepción

en el caso de que haya ocurrido un error. En este caso podemos observar que solo hay lectura de datos y no hay escritura de datos.

Para finalizar, en la opción 3 de sobrescribir sobre un archivo se combinarán procedimientos de la opción 1 y la 2. se uso stdout.write para el nombre o la ruta del archivo a escribir y con un ciclo if se verifica si la ruta es valida o no y después stdin.readLineSync(); para leer este valor, se hace otro ciclo if para verificar que el archivo exista y si no existe se imprimirá que no es posible hacer esa operación por que el archiono existe. Si el archivo se encuentra se le dice al usuario si quiere sobrescribir el texto, se leerá la línea de confirmación del usuario, si el usuario no dice que "SI"para confirmar la operación se cancelará. En el caso de que si se haya confirmado la operación a traves de una lista se guardaran las palabras ingresadas hasta que se lea la palabra "FIN.º que sea nula la línea y por ultimo arrojamos la excepción en el caso de que haya habido un error al sobrescribir el archivo.

En la compilación del programa, se utilizó el archivo de la opción 1 para poder realizar la opción 2 y 3, esto gracias al repositorio que almacena la información que se registró en el programa.

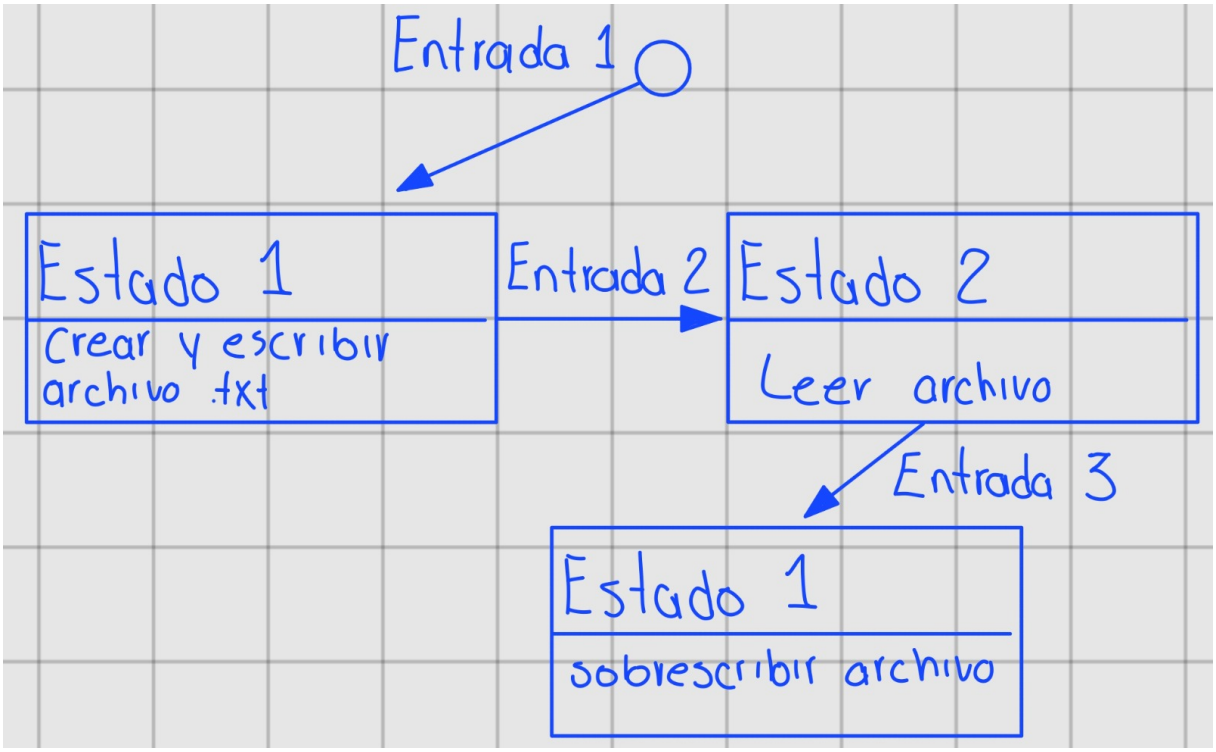

Figura 1: Diagrama de estados que muestra como fue la compilación

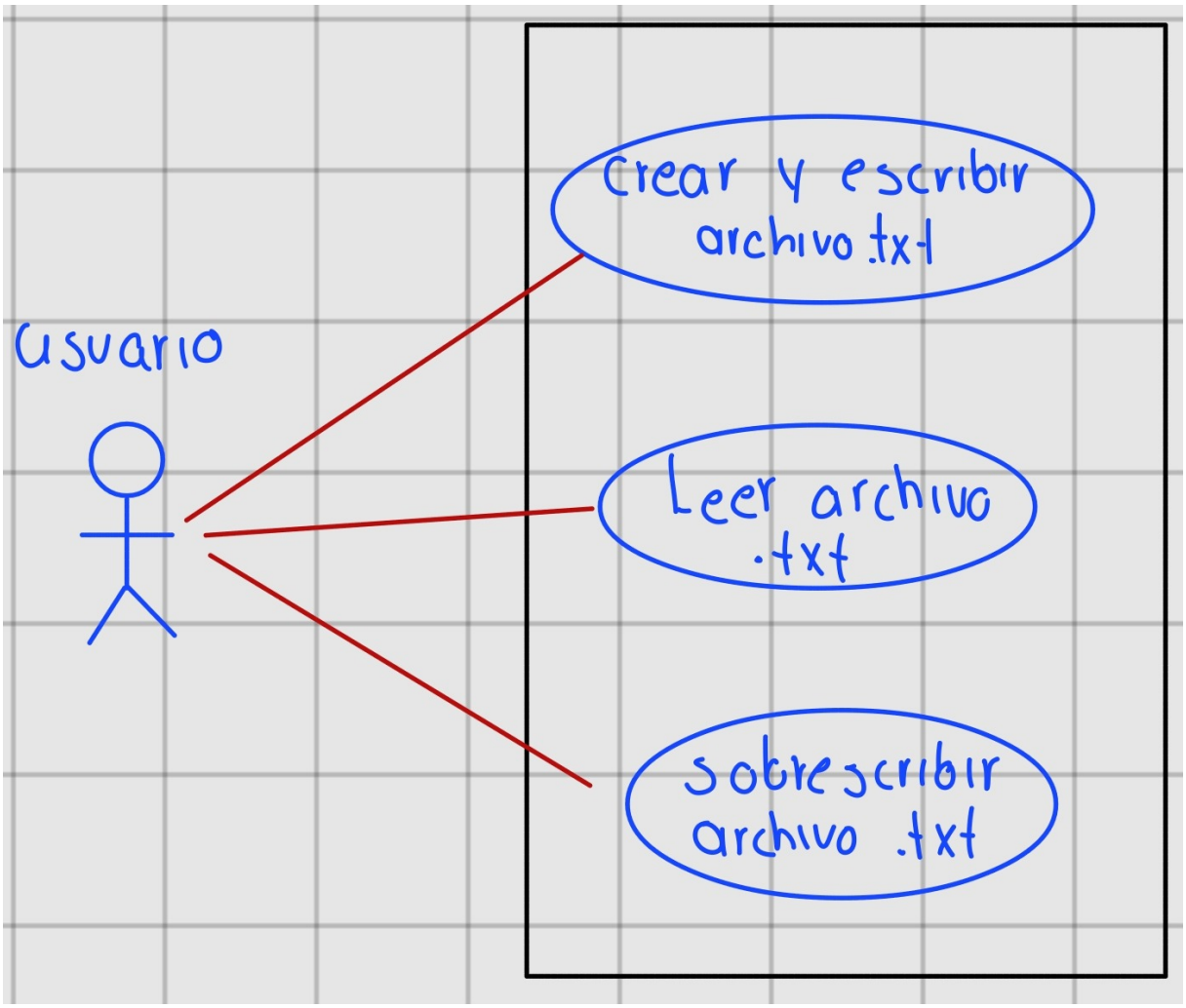

Figura 2: Diagrama de uso

3.2. Hilos

Como vimos anteriormente los hilos permiten al programador actuar como un titiritero, que controla sus aplicaciones mediante de hilos (De ahí el nombre), logrando que una aplicación realice varios procesos simultáneamente, como en una marioneta donde sus partes se mueven individualmente.

A través de los distintos ejemplos proporcionados por el profesor, logramos notar diferentes aplicaciones de los hilos en el lenguaje –Dart–, donde se hace uso de un modelo basado en ejecuciones independientes con `isolate` y `event loop`, aunque estos últimos son de concurrencia ligera, ya que no están pensados para ser usados en gran escala. Para

ver los usos que tienen estos modelos de ejecuciones empezaremos con el Ejemplo 6.

- Ejemplo 6: En este ejemplo se nos muestra una ejecución normal de un código, con procesos de manera secuencial, donde imprime un mensaje, realiza un ciclo `for` de 500 millones de iteraciones, donde el único hilo en uso, se detiene un momento y por ultimo imprime un segundo mensaje de finalización. Este ciclo `for` es bastante pesado, y al ser llevado a cabo por un solo núcleo, sumado a imprimir los mensajes, hace que el consumo de este núcleo sea mucho mayor.

- Ejemplo 4: En este ejemplo se nos muestra una ejecución paralela mediante `isolate`, que aunque no es paralelismo real, ya que un solo procesador hace el proceso del programa completo, se toma como paralelo por dividir la tarea en 2 hilos distintos, en este caso uno se encarga de los mensajes y otro del ciclo `for`, haciendo en conjunto el procesamiento de la aplicación y por consecuencia que ésta misma sea más rápida.

- Ejemplos 1 y 2: Estos ejemplos nos muestran el uso de `event loops` a través de `future` y `await`, donde estos no generan nuevos hilos, sino que todo funciona en un solo isolate, pero estas tareas se ejecutan de forma asíncrona. El `event loop` toma tareas que se pueden sincronizar y las realiza cuando hay tiempo, dando la ilusión de concurrencia.

- Ejemplo 3: En este ejemplo, el proceso que se lleva a cabo es completamente independiente y es la primera vez que se usa `isolate.spawn`, lo cuál nos permite usar isolate y realizar una independencia de procesos.

- Ejemplo 5: Aquí por último tenemos un ejemplo de ciclo completo de el uso de estos modelos de ejecución, con una ejecución en paralelo que se cierra correctamente, y así mismo un ejemplo del paso de mensajes, donde se establece un `handshake`.

3.3. Patrones de Diseño

En la clase sobre patrones de diseño se abarcaron dos patrones, cada uno con su respectivo ejemplo. El enfoque de esta práctica es el primero – el patrón *Singleton*.

El código ejemplar que muestra la manera de implementar y utilizar el patrón *Singleton* en el lenguaje de programación Dart, modela la relación entre una impresora y las solicitudes de impresión que esta recibe. Lo anterior se consigue a través de las clases `Impresora` –la cual se implementa con el patrón en cuestión– y `Documento`.

La clase `Impresora` define tres atributos inmutables con la palabra clave `final`. Estos son: **instancia** – de tipo `Impresora`–, **colaImpresion** –de tipo `Queue<Documento>`– e **historial** –de tipo `List<Documento>`, los cuales también son privados. Con el atributo **instancia** se cubre uno de los requerimientos de la implementación del patrón *Singleton* – contiene la única instancia que se permite para esta clase. La palabra clave `static` en la definición del atributo anterior establece que este no es un atributo propio de los objetos de tipo `Impresora`, sino que existe como parte de este tipo. Otro requerimiento se cumple con la definición de un constructor privado –llamado **interna**– que es el que se utiliza para construir la instancia única. Para que la instancia única sea accesible, con la palabra clave `factory` se establece que el constructor por defecto regresa una instancia de la clase en cuestión o su subclase, que en este caso es la instancia única. El último requerimiento se cumple por el hecho de que no se implementan otros constructores que se puedan utilizar para instanciar otros objetos de esta clase. La clase `Impresora` también define cuatro métodos: **enviarDocumento()**, **imprimirSiguiente()**, **mostrarCola()** y **mostrarHistorial()**, los cuales se encargan de agregar un documento a la cola de impresión, escribir a la salida estándar el documento que sigue en la cola de impresión, mostrar todos los documentos en la cola de impresión y mostrar el historial de impresiones – respectivamente.

La clase `Documento` define cuatro atributos inmutables: **id** de tipo `int`, y **usuario**, **nombre**, **contenido** de tipo `String`. Para el constructor por defecto de esta clase se establece –mediante el uso de la palabra clave `required`– que se requiere ingresar valores para todos sus atributos a la hora de instanciar un objeto. Por último, en la clase `Documento` se sobre-escribe el método `toString()` de manera que este regrese una cadena con los valores de todos sus atributos en un formato específico.

El punto de inicio del programa es el método `main()`; aquí se agregan tres objetos de tipo `Documento` a la cola de impresión de la impresora, posteriormente se utiliza el

método **mostrarCola()** para escribir el contenido de la anterior a la salida estándar. Luego se utiliza el método **imprimirSiguiente()** para simular la impresión de dos de los documentos anteriormente agregados a la cola, y se vuelve a mostrar su contenido. Se llama otras dos veces el método **imprimirSiguiente()** y, por último, se muestra el historial de impresiones a través el método **mostrarHistorial()**. A lo largo de la ejecución de las instrucciones anteriores, se manejan dos referencias de tipo `Impresora`, individualmente creadas por el constructor por defecto. Como se había establecido, el constructor por defecto regresa la única instancia permitida para la clase mencionada, por lo que dichas referencias ambas están ligadas al mismo objeto. Esto se hizo para resaltar el hecho de que la clase `Impresora` se implementó como *Singleton* exitosamente.

4. Resultados

```
==============================
       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): hola.txt

Escribe el texto que deseas guardar.
Para terminar, escribe SOLO: FIN
--------------------------------
hola
no soy
emo
FIN

Archivo creado y guardado correctamente.
```

Figura 3: Opción 1: crear y escribir archivo .txt

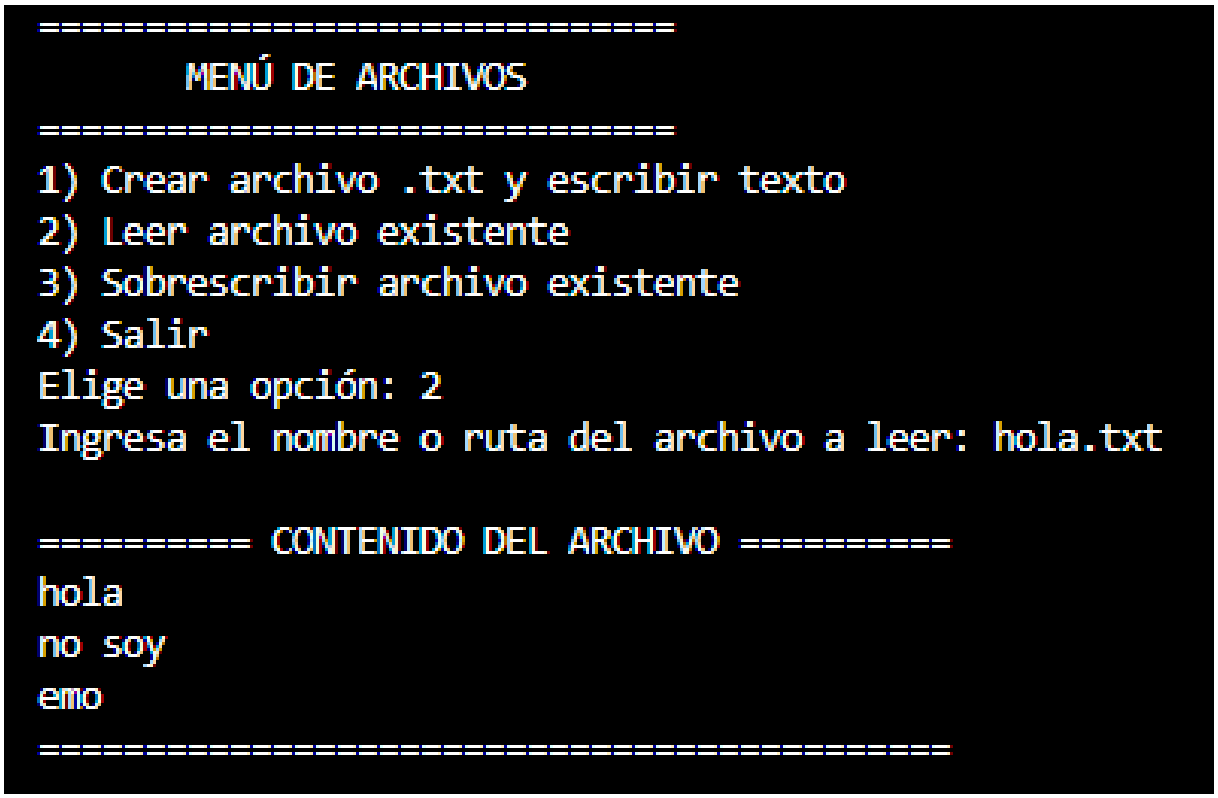

Figura 4: Opción 2: leer un archivo .txt

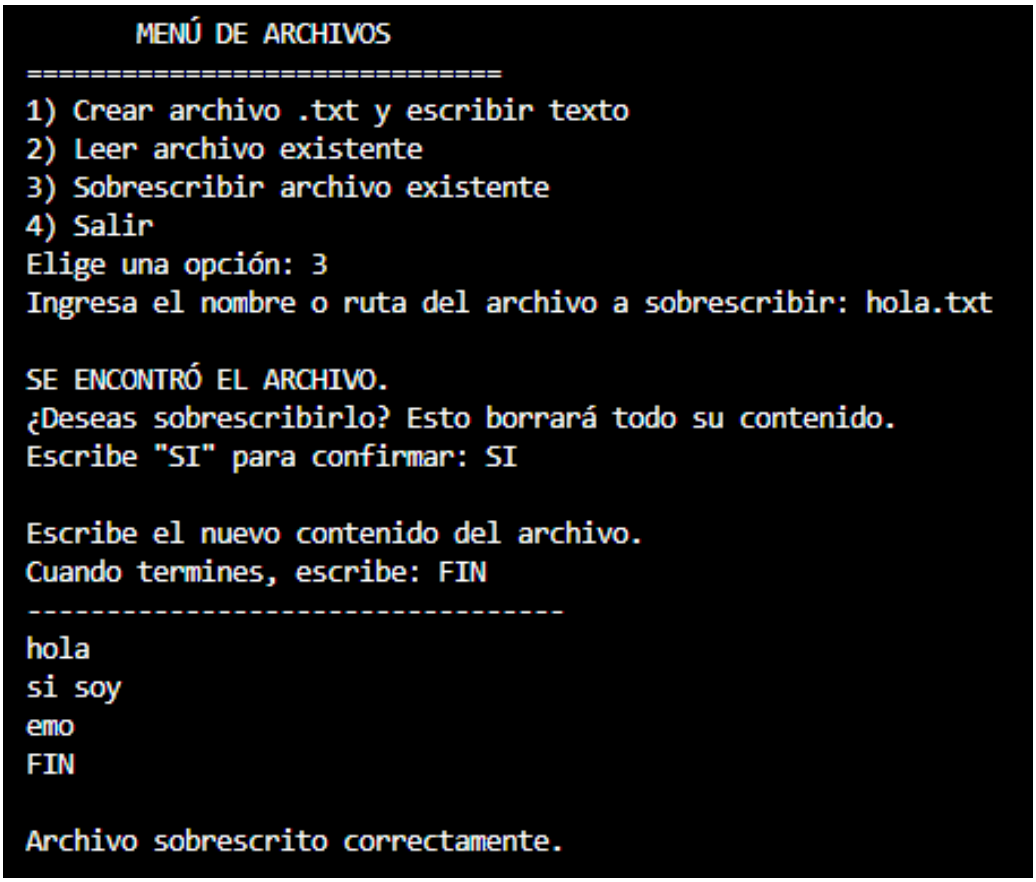

Figura 5: Opción 3: sobreescribir un archivo .txt

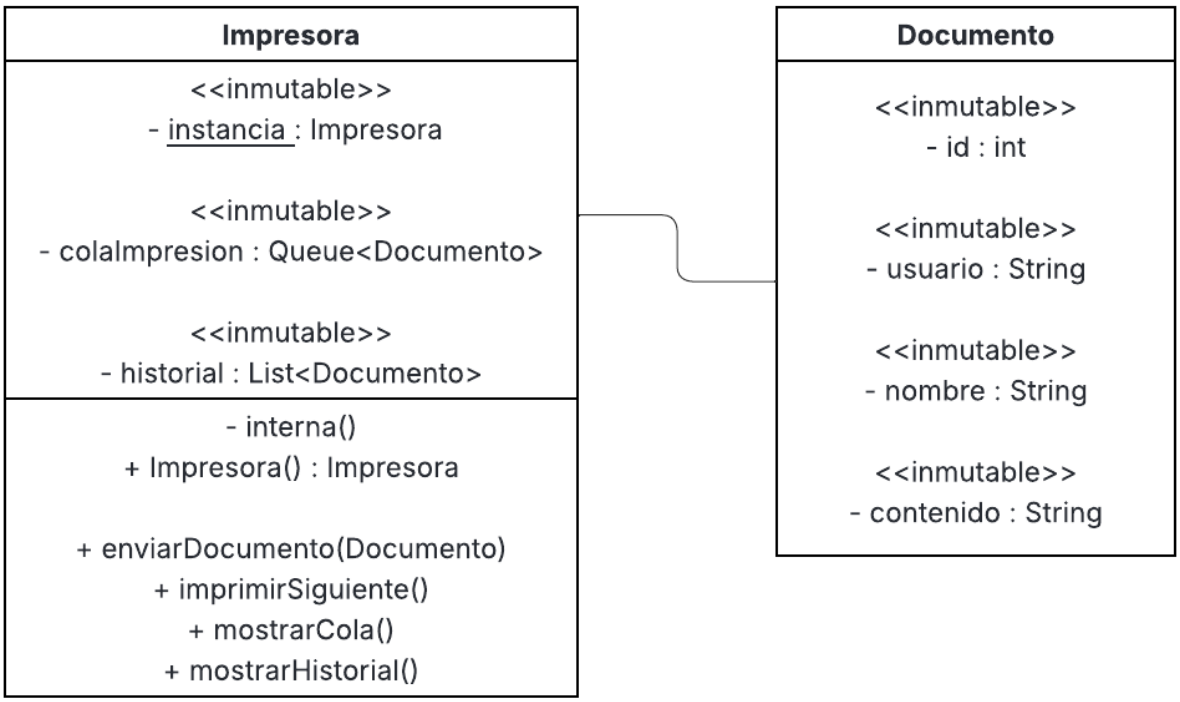

Figura 6: Diagrama de clases del ejemplo del patrón Singleton

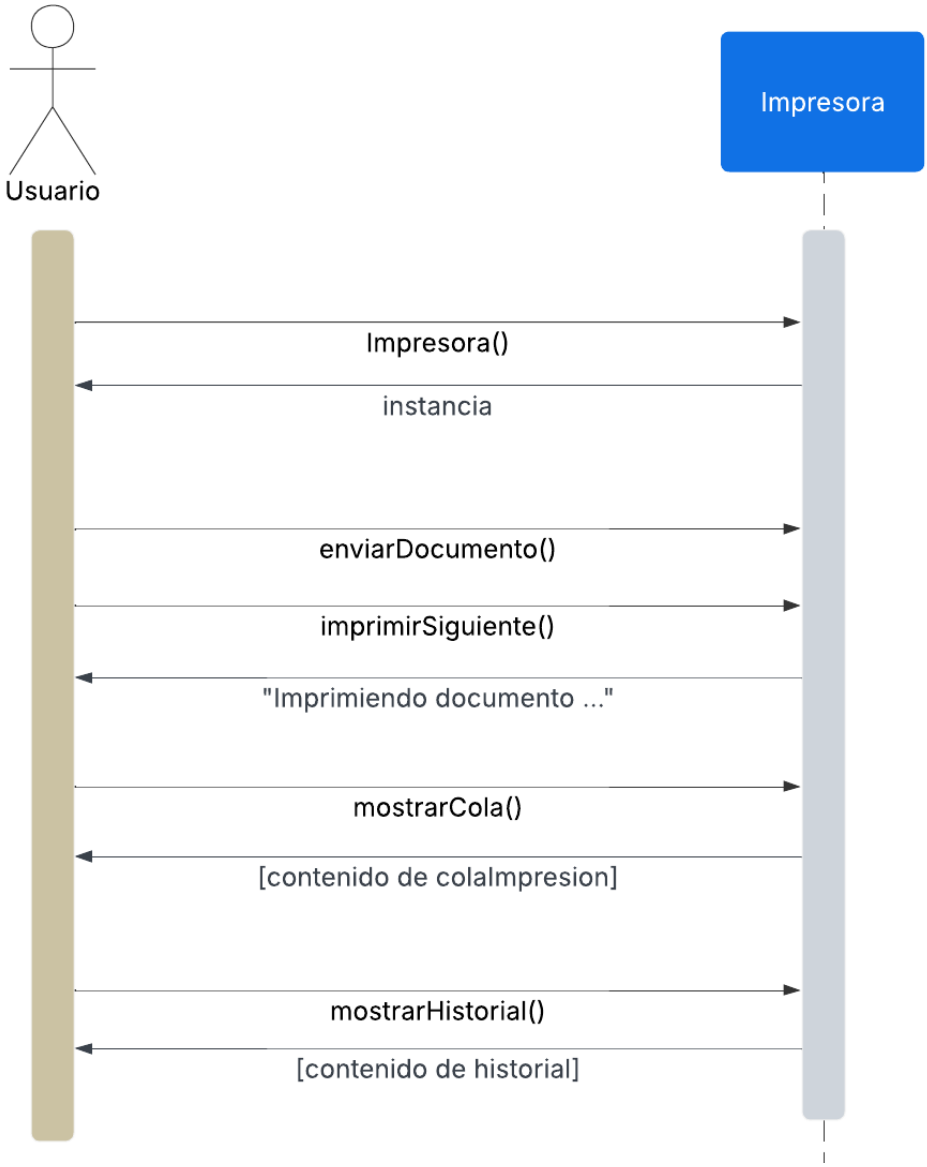

Figura 7: Diagrama de secuencia del ejemplo del patrón Singleton

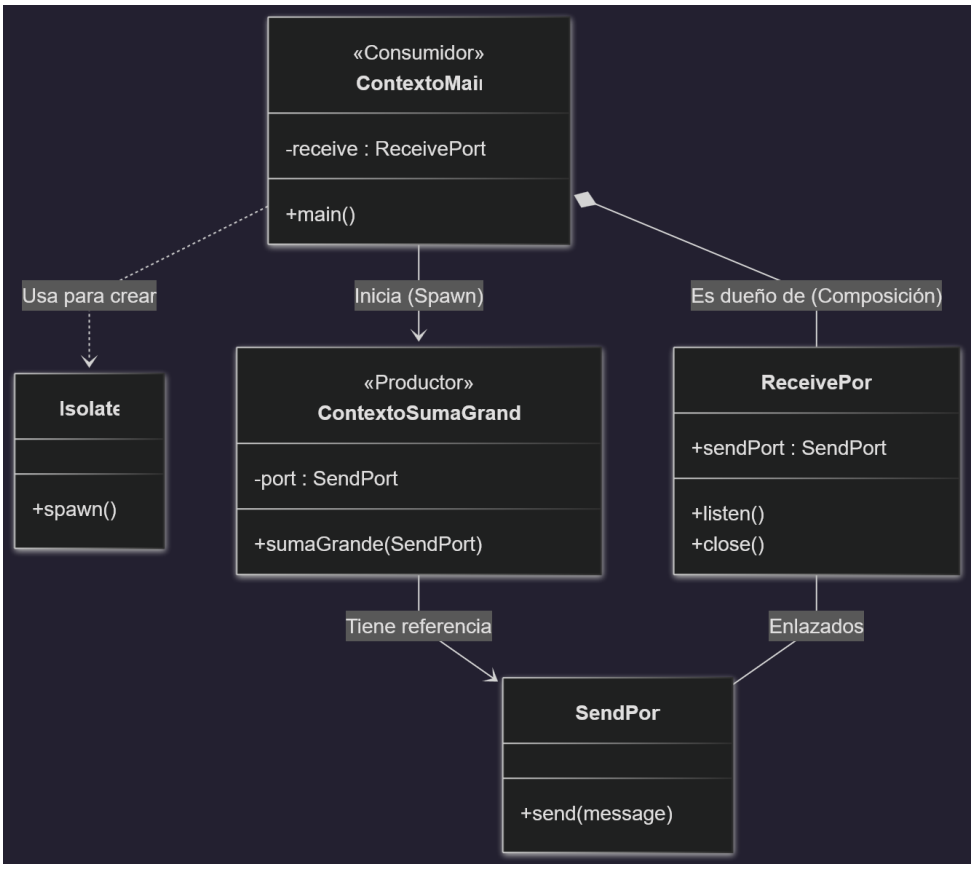

Figura 8: Diagrama de clases del ejemplo 4 del tema Hilos

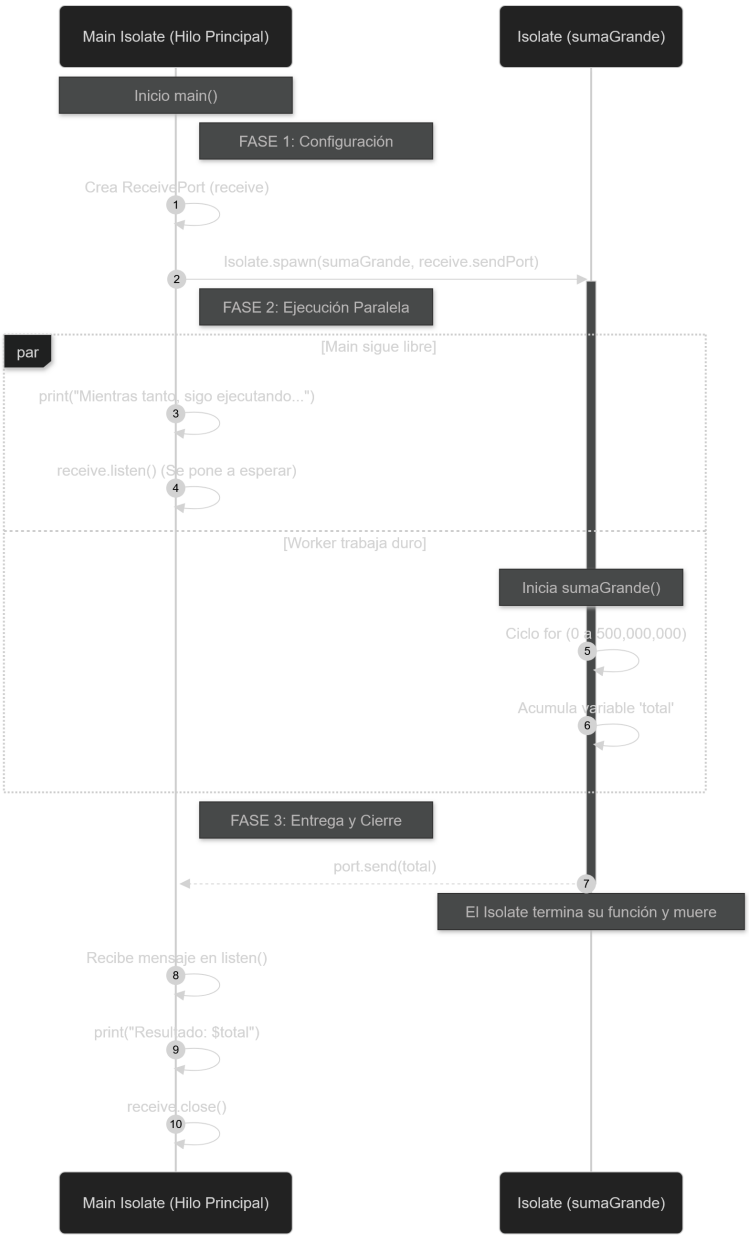

Figura 9: Diagrama de secuencia del ejemplo 4 del tema Hilos

5. Conclusiones

Es satisfactorio cerrar el trabajo de múltiples practicas a lo largo de este curso, y en estas ultimas hemos consolidando todo el conocimiento teórico en primera instancia, la gestión de Archivos se reveló como un mecanismo crucial para asegurar la persistencia de los datos, permitiendo que el estado de la aplicación se mantenga más allá de la ejecución del programa. Simultáneamente, la identificación e implementación de Patrones de Diseño no solo sirvió para resolver problemas comunes de arquitectura de manera robusta y probada, sino que también garantizó que el sistema mantenga una alta flexibilidad, facilitando futuras expansiones sin alterar el código base. Se abordaron con éxito dos aspectos cruciales de la programación: la persistencia de datos y la eficiencia. Se utilizó la biblioteca dart:io para gestionar las operaciones de Entrada y Salida de archivos de manera controlada como crear, leer, sobrescribir y manejo de excepciones. Simultáneamente, se implementaron modelos de concurrencia de Dart (Future/await con Event Loop para tareas asíncronas) y paralelismo real para evitar que las operaciones lentas bloqueen el único hilo de ejecución, asegurando así que la aplicación sea rápida y responsiva. Finalmente, el Modelado UML fue una herramienta indispensable, los diagramas estáticos como Clases proveyeron una visión clara de la estructura del sistema y sus relaciones, mientras que los diagramas dinámicos permitieron visualizar el comportamiento y la interacción de los objetos en tiempo de ejecución.

Referencias

[1] L. Jorge. *Introducción a POO en Java: Excepciones*. Nov. de 2023. URL: https://openwebinars.net/blog/introduccion-a-poo-en-java-excepciones/.

[2] M. miriam. *¿Qué son los patrones de diseño de software?* URL: https://profile.es/blog/patrones-de-diseno-de-software/.

[3] J.A. Solano. *Hilos*. 2020. URL: https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3084/mod_resource/content/1/UAPA-Hilos/index.html.

6. Datos de trabajo

Trabajo en el reporte:

- Introducción: Colaborado por 320065570
- Marco teórico: Colaborado por 320065570
- Desarrollo y resultados: Colaborado por 425032224, 425133840 (archivos)
- Conclusiones: Colaborado por 322229916
- Diagramas: Colaborado por 425032224, 425133840