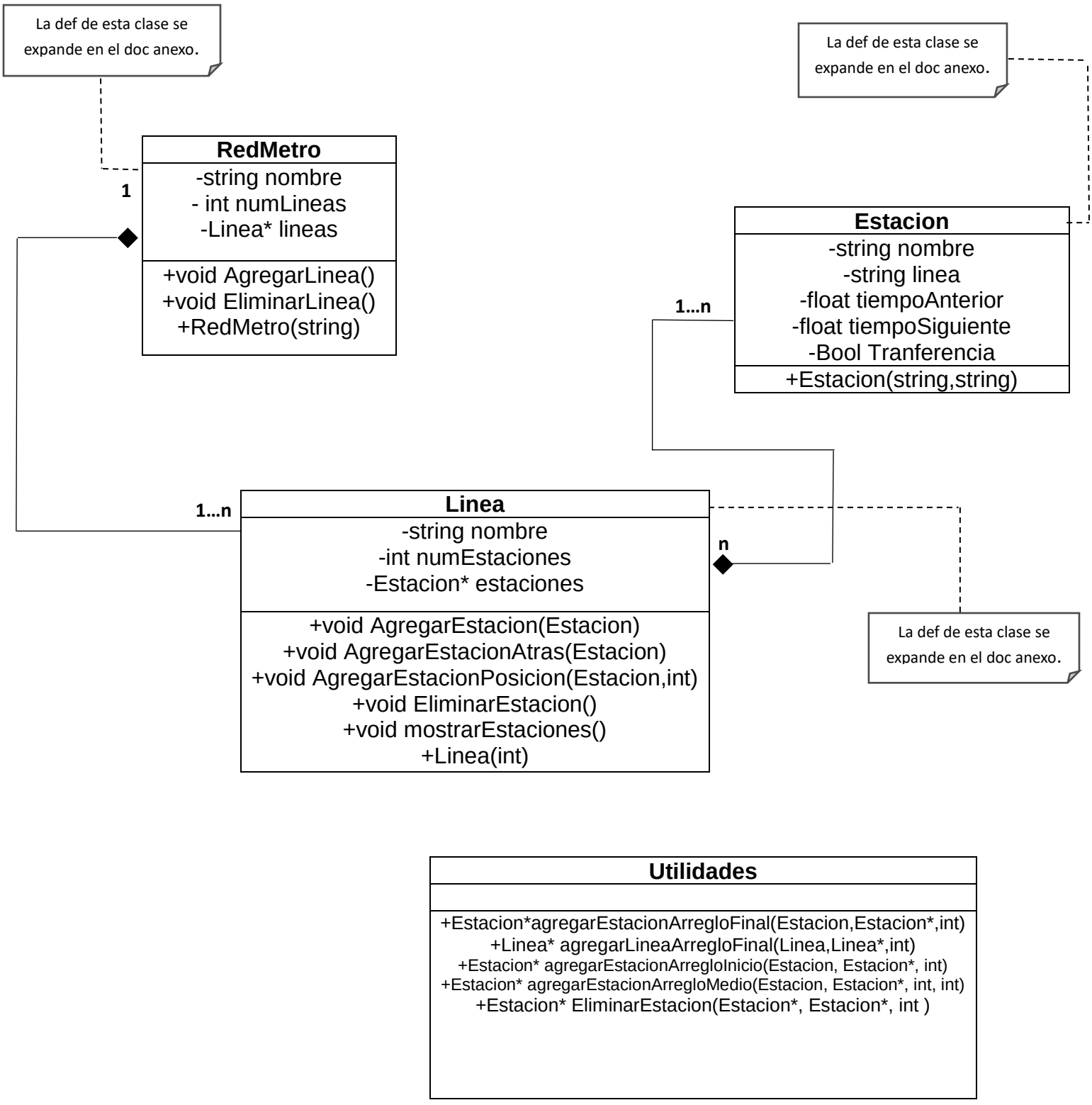
Proceso de análisis y documentación - Desafío 2.

a) Diagrama de clases



Documento anexo al Diagrama de clases (UML simplificado)

En este documento sé pondrán algunos métodos que se omitieron en el diagrama de clases, debido a exceso de información; en particular, se pondrán: getters, setters, destructores y algunos constructores.

- **Extensión de la clase “RedMetro”:**

void setNombre(string nombre)

string getNombre()

void setLineas(Linea* lineas)

Linea* getLineas() const

void setNumLineas(int num)

int getNumLineas()

- **Extensión de la clase “Linea”:**

Linea();

string getNombre()

void setNombre(string nombre)

void setEstaciones(Estacion* estaciones)

Estacion* getEstaciones()

void setNumEstaciones(int num)

int getNumEstaciones()

- **Extensión de la clase “Estacion”:**

Estacion()

~Estacion()

void setNombre(string nombre)

string getNombre()

void setTransferencia(bool transferencia)

bool getTransferencia()

void setTiempoSiguiente(float tiempoSiguiente)

float getTiempoSiguiente()

void setTiempoAnterior(float tiempoAnterior)

float getTiempoAnterior()

void setLinea(string linea)

string getLinea()

b) Análisis del problema y consideraciones para la alternativa de solución propuesta.

Explicación detallada de las clases, sus atributos y sus métodos (se omiten la explicación de getters, setters, destructores y algunos constructores**)**

Clase "RedMetro"

Hemos decidido crear una clase llamada **"RedMetro"** con el fin de que esta contenga un conjunto de atributos y métodos que describan cómo se verá y se comportará un objeto de este tipo.

Para la clase "RedMetro" hemos creado los siguientes atributos:

- **String nombre:** Este atributo es de tipo string porque almacenará el nombre que el usuario le pondrá a la "red metro".
- **Int numlineas**: Este atributo almacenará un número entero positivo y diferente de cero; el cuál, representará la cantidad de líneas que compondrán la "red metro". (recordar hacer validaciones para que el número ingresado siga las características)
- **Linea* líneas:** Este atributo es un puntero llamado "líneas" que apunta a un arreglo unidimensional; el cuál, almacena objetos tipo "linea"; en otras palabras, este arreglo almacena el nombre de las líneas que contendrá la red metro.

Para la clase "RedMetro" hemos creado los siguientes métodos:

- **void AgregarLinea(**): Este método toma un objeto de la clase línea como parámetro, la agrega al atributo de la clase Redmetro que es un arreglo que almacena objetos de la clase línea, también actualiza la lista de líneas y aumenta el contador de líneas en uno para reflejar la adición de la nueva línea.
- **void EliminarLinea()**: Este método elimina la primera línea del arreglo que almacena objetos tipo línea; en primera instancia, reemplaza el arreglo de líneas con un nuevo arreglo de tamaño 1 y establece el contador de líneas en 0. Finalmente, imprime un mensaje indicando que la operación ha sido completada. En resumen, este método limpia el arreglo que contiene las líneas de la red metro, dejándola vacía.
- **RedMetro(string nombre**): Este método es un contructor de la clase "RedMetro", el cuál inicializa un objeto de la clase RedMetro con un nombre dado, posterior a esto crea un arreglo dinámico para almacenar las líneas que pertenecerán a la red metro. También inicializa el contador de líneas en "0", indicando que aún no hay líneas en la red metro cuando se crea un objeto de esta clase.

¿Cuántos objetos tendrá la clase "RedMetro"? Según nuestro análisis, la clase "red metro" tendrá una sólo objeto, pues nuestro programa sólo se encargará de modelar una red.

Clase "Linea"

Hemos decidido crear una clase llamada **"Línea"** con el fin de que esta tenga un conjunto de atributos y métodos que describan cómo se verá y se comportará un objeto de este tipo.

Para la clase "Linea" hemos creado los siguientes atributos:

- **String nombre:** Este atributo almacenará un nombre; en este caso particular, el nombre que el usuario le pondrá a cada uno de los objetos de la clase "Linea".
- **Int numEstaciones:** Este atributo almacenará un número entero positivo y diferente de cero; el cuál, representará la cantidad de estaciones que tendrán cada uno de los objetos tipo linea. Este dato será útil para controlar la dimensión de los arreglos.
- **Estacion* estaciones:** Este atributo es un puntero llamado "estaciones" que apunta a un arreglo unidimensional que contiene objetos de la clase "estación". Este arreglo almacenará la cantidad de estaciones que compondrán a cada una de las líneas

Para la clase "Línea" hemos creado los siguientes métodos:

- **void AgregarEstacion(Estacion estacion):** Este método permite agregar una estación en la última posición del arreglo que contiene las estaciones de una línea en específico (la línea es escogida por el usuario, mediante el despliegue de un menú que se imprimirá por consola) Adicional a esto, este método también pide al usuario que ingrese el tiempo que desea que haya entre la estación que se acaba de agregar y la estación anterior.
- **void AgregarEstacionAtras(Estacion estacion):** Este método permite agregar una estación en la primera posición del arreglo que contiene las estaciones de una línea en específico (la línea es escogida por el usuario, mediante el despliegue de un menú que se imprimirá por consola) Adicional a esto, este método también pide al usuario que ingrese el tiempo que desea que haya entre la estación que se acaba de agregar y la estación siguiente.

  void AgregarEstacionPosicion(Estacion estacion, int indice): Tiene como objetivo agregar una estación en una posición específica dentro del arreglo de estaciones perteneciente a una línea en particular. En primer lugar, este método verifica que el número de estaciones sea menor o igual a cero; en caso de ser verdadero; la estación que se está agregando pasa a

ser el primer elemento del arreglo; es decir, la primera estación de la línea. De lo contrario; es decir, que si la línea ya tiene al menos una estación se le pide al usuario por consola los tiempos que hay entre las estaciones adyacentes a la estación que sé acaba de agregar.
Es importante resaltar que se deben actualizar los tiempos de la estaciones que son adyacentes a la estación que se acaba de agregar, porque debido a esa inmersión de un nuevo dato en el arreglo, los tiempos que se tenían antes han cambiado, esto lo hacemos por medio de algunos métodos que están definidos en la clase "Utilidades", específicamente con "setters", pues son estos, los que permiten modificar valores en ciertos atributos.

- **void EliminarEstacion**(): Este método elimina una estación específica que está contenida en una arreglo tipo línea, siempre y cuando no sea una estación de transferencia, ajusta el arreglo de estaciones correspondiente y actualiza el contador de estas.
- **void mostrarEstaciones() :** Este método imprime en consola las estaciones que pertenecen a una línea determinada, separándolas con flechas para indicar la secuencia que hay entre ellas; para hacerlo, hemos utilizado un ciclo for, pues este permite recorre todos los elementos de arreglo que contiene las estaciones.
- **Linea(int num):** Este método es uno de los constructores que se crearon para la clase Linea y sé invoca cada vez que se crea un nuevo objeto de la clase línea. Este constructor se encarga de solicitar al usuario el nombre de la nueva línea, inicializa el número de estaciones en cero y asigna un arreglo de estaciones (un arreglo que contiene objetos de tipo estaciones) de tamaño 1 a la línea.

¿Cuántos objetos tendrá la clase "Linea"? La clase línea tendrá la cantidad de objetos que almacene el atributo definido como **Int numlineas** en la clase "red metro"; cabe aclarar, que el número de líneas es un dato que se conocerá en tiempo de ejecución.

Clase “Estacion”

Hemos decidido crear una clase llamada “Estacion” con el fin de que esta contenga un conjunto de atributos y métodos que describan cómo se verá y se comportará un objeto de este tipo.

Para la clase “estacion” hemos creado los siguientes <u>atributos:</u>

- **Float tiemposiguiente:** Tiempo que hay entre una estación y la siguiente.
- **Float tiempoAnterior:** Tiempo que hay entre una estación y la anterior.
- **String nombre**: Atributo de tipo string que almacena el nombre de la estación.
- **String línea:** Atributo de tipo string que almacena el nombre de la Linea a la que pertenece el objeto de tipo estación.
- **Bool Transferencia:** Este atributo sirve para reconocer si una estación es de transferencia o no, en caso de que una estación sea de transferencia, este atributo almacena un booleano “True”; de lo contrario, almacena un booleano “False”.

Para la clase “estacion” hemos creado los siguientes <u>métodos:</u>

Estacion(string nombre,string linea): Este método es un constructor de la clase Estacion, y es invocado cuando se crea una nueva instancia de esta clase. Este constructor inicializa los atributos de un objeto de la clase Estacion con el nombre que va a tener el objeto de esta clase y el nombre de la línea a la que pertenece.

¿Cuántos objetos tendrá la clase “estacion”? El número de objetos que tendrá la clase estación se conocerá en el tiempo de ejecución del programa, pues depende directamente del número de líneas que el usuario cree dentro de la red metro, y de la cantidad de estaciones que el usuario desee que haya; además de esto, también es importante mencionar que las estaciones de trasferencia solo se contaran una vez, sin importar que estén en dos o más líneas.

Clase “Utilidades”

Hemos decidido crear una clase llamada “Utilidades” con el fin de que esta contenga los métodos que nos servirán para poder modificar un arreglo de tipo estación y de tipo línea. Es importante mencionar que esta clase no tiene atributos, sólo métodos.

Para la clase “Utilidades” hemos creado los siguientes métodos:

- **static Estacion* agregarEstacionArregloFinal(Estacion estacion, Estacion* arreglo, int longitud):**

Este método tiene como objetivo agregar un objeto de la clase “Estacion” al final de un arreglo de objetos tipo Estacion, este método recibe 3 parámetros: un objeto de la clase Estacion que se va a agregar al nuevo arreglo, un puntero a un objeto de la clase estación que representa el arreglo existente y un entero que indica la longitud actual del arreglo. Sé retornará un puntero al nuevo arreglo que contiene la nueva estación que ha sido agregada.

- **static Estacion* agregarEstacionArregloInicio(Estacion estacion, Estacion* arreglo, int longitud):**

Este método agrega una estación al principio de un arreglo de estaciones; en otras palabras, esta función inserta una estación al inicio del arreglo y ajusta la longitud del arreglo, pues la inmersión de una nueva estación hizo que la dimensión del arreglo cambie.

Es importante mencionar que en este método crearemos un nuevo arreglo de estaciones con una longitud aumentada en 1, posterior a esto copiaremos todos los elementos que había en el arreglo original al arreglo nuevo y los desplazaremos una posición hacia la derecha, para poder así, agregar una nueva estación al principio del arreglo.

- **static Estacion* agregarEstacionArregloMedio(Estacion estacion, Estacion* arreglo, int indice, int longitud):**

Este método agrega una estación en una posición específica dentro de un arreglo de estaciones; es decir, inserta una estación en una posición determinada del arreglo, ajustando adecuadamente las posiciones de los elementos antes y después de la inserción.

- **static Linea* agregarLineaArregloFinal(Linea linea, Linea* arreglo, int longitud):**

Este método inserta una línea al final del arreglo, y se asegura de reajustar la longitud del arreglo.

- **static Linea* agregarLineaArregloInicio(Linea linea, Linea* arreglo, int longitud):**

Este método agrega una línea al inicio del arreglo, asegurándose de ajustar adecuadamente la longitud del arreglo y de copiar correctamente los elementos existentes en el arreglo antiguo. También se encarga de liberar la memoria del arreglo original para evitar fugas de memoria.

- **static Linea* agregarLineaArregloMedio(Linea linea, Linea* arreglo, int indice, int longitud):**

Este método inserta una línea en una posición determinada del arreglo, ajustando adecuadamente las posiciones de los elementos antes y después de la inserción. También se encarga de libera la memoria del arreglo original para evitar fugas de memoria.

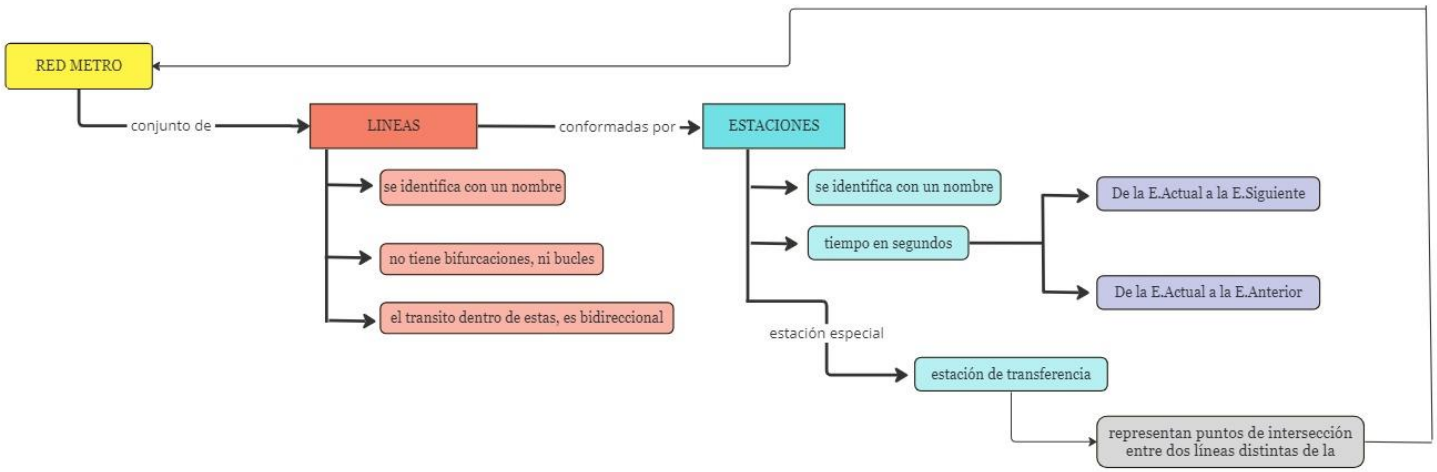
- **static Linea* EliminarLinea(Linea linea, Linea* arreglo, int longitud):**

Este método se encarga de eliminar una línea del arreglo, ajustando adecuadamente las posiciones de los elementos después de la eliminación.

- **static Estacion* EliminarEstacion(Estacion estacion, Estacion* arreglo, int longitud):**

Este método se encarga de eliminar una estación del arreglo, ajustando adecuadamente las posiciones de los elementos después de la eliminación.

Diagrama simplificado de las condiciones del problema.



Condiciones a tener en cuenta por parte del problema:

A) Agregar una estación a una línea, en los extremos o en posiciones intermedias.
B) Eliminar una estación de una línea. No se pueden eliminar estaciones de transferencia.
C) Saber cuántas líneas tiene una red Metro.
D) Saber cuántas estaciones tiene una línea dada.
E) Saber si una estación dada pertenece a una línea específica.
F) Agregar una línea a la red Metro.
G) Eliminar una línea de la red Metro (sólo puede eliminarse si no posee estaciones de transferencia).
H) Saber cuántas estaciones tiene una red Metro (precaución con las estaciones de transferencia).

Teniendo en cuenta el esquema y las condiciones anteriores, hemos diseñado un plan de desarrollo con el fin de resolver el problema planteado en este desafío. Se pondrán las ideas a manera de ítems; en donde cada uno de ellos,

corresponderá a la descripción de las estrategias que se utilizarán para resolver cada una de las tareas.

Estrategia de solución para el inciso “A”:
Para darle solución a este inciso, se crearon 3 métodos de la clase utilidades, los cuáles son: **“agregarLineaArregloFinal” “agregarLineaArregloInicio”, “agregarLineaArregloInicio”.**

Estrategia de solución para el inciso “B”:
Para darle solución a este inciso, se creó el método de la clase línea, llamado “**EliminarEstacion”.**

Estrategia de solución para el inciso “C”:
Usaremos el atributo, “numLineas“de la clase “RedMetro”, pues este nos dará el número de líneas que conforman la red.

Estrategia de solución para el inciso “D”:
Usaremos el atributo, “numEstaciones” de la clase de la “Linea”, pues este nos dará el número de estaciones que conforman la línea.

Estrategia de solución para el inciso “E”:
Para darle solución a este inciso, se creó el método de la clase línea, llamado “**EstacionPertenece”.**

Estrategia de solución para el inciso “F”:
Para darle solución a este inciso, se creó el método de la clase RedMetro, llamado **“AgregarLinea”.**

Estrategia de solución para el inciso “G”:
Para darle solución a este inciso, se creó el método de la clase RedMetro, llamado **“EliminarLinea”.**

Estrategia de solución para el inciso “H”:

c) Algoritmos implementados debidamente intra-documentados.

Este requisito se ve reflejado en el código por medio de comentarios en partes necesarias de la implementación, denotados con "//".

d) Problemas de desarrollo que afrontó.

Uno de los principales problemas que afrontamos, fue tomar la decisión de establecer cuantas clases queríamos que tuviera el programa y de encontrar la relación de dependencia que había entre cada una de ellas; además de esto, la cardinalidad que estas comparten.

También fue un poco complejo saber qué relación había entre la clase "Utilidades" y las demás clases, pues teniendo en cuenta que el objetivo de la clase utilidades es que contenga cada uno de los métodos que nos servirán para poder modificar un arreglo de tipo estación y de tipo línea, no encontrábamos una relación tan obvia en un primer lugar.

e) Evolución de la solución y consideraciones para tener en cuenta en la implementación.

Partir del diseño de un informe inicial se convierte en un ejercicio positivo y negativo a la vez, pues el determinar una ruta es un avance para el inicio, pero reconocer que no todo lo planteado funciona y que requiere de cambios significativos cuando se empieza a implementar lo propuesto, juega significativamente en todo el proceso, donde el tiempo es una de las variables más importantes a tener en cuenta, pues se debe contemplar acciones no solo de diseño de códigos si no también acciones de análisis, comprensión, interpretación creatividad, ensayo y error que fueron cambiando en la medida en que se iba reconociendo cuales serían las rutas más largas o cortas para hacer mucho más optimo el código.

La evolución de la solución empieza a coger forma cuando se lleva lo que se piensa al plano de la gráfica o del dibujo, facilitando el análisis y la interpretación de lo que se debía trabajar.

Es importante resaltar que tuvimos un problema con el uso de las ramas "miguel" y "sofia" en el repositorio de Github; pues tuvimos un percance, y casi corremos el riesg de perder todo lo que habíamos avanzado en el código; es por esto que tomamos la decisión de borrar las ramas y solamente hacer commits sobre la rama main, pues el tiempo que tardábamos en entender que estaba pasango en Github, era tiempo que necesitábamos para seguir avanzando en la implementación del código y la documentación.