Informe del Desafío II de Informática II *Fecha: 15/10/24*

Sistema de comercialización de combustible TerMax

Santiago Cortés Osorio **s.cortes@udea.edu.co**
Sara María Hincapié Hincapié **sara.hincapie1@udea.edu.co**

Resumen: Este proyecto se enfoca en la optimización de la gestión de estaciones de servicio de TerMax, con un enfoque particular en la administración de los surtidores de combustible. El sistema diseñado permite una eficiente asignación de recursos, simulación de ventas y control de las transacciones. A través del uso de un enfoque basado en programación orientada a objetos (POO) en C++, se busca mejorar la operación y la toma de decisiones dentro de la red nacional de estaciones.

1) Análisis del Problema:

La solución del problema se plantea mediante la utilización de la POO, por consiguiente, la primera parte del análisis se centró en generar una lista de criterios que nos permitieran determinar qué ha de ser un objeto y qué no.
Los criterios que definimos fueron los siguientes:

1. Que la clase interactuara con el resto del sistema.
2. Los métodos asociados a la clase hicieran parte de los requisitos funcionales.

Bajo estos 2 criterios, definimos 3 clases: RedNacional, Estacion y Surtidor.

RedNacional representa la Red Nacional de Gasolineras de TerMax, esta permite hacer gestión de las Estaciones de Servicio asociadas a una instancia específica de esta, aparte de ciertas funcionalidades específicas a esta como el reporte de ventas general.

Estacion simboliza las Estaciones de Servicio asociadas a una Red Nacional de Gasolineras, y permite hacer gestión de sus surtidores asociados, cada instancia de esta, como revisar cierta información asociada, junto con la cantidad de combustible vendida.

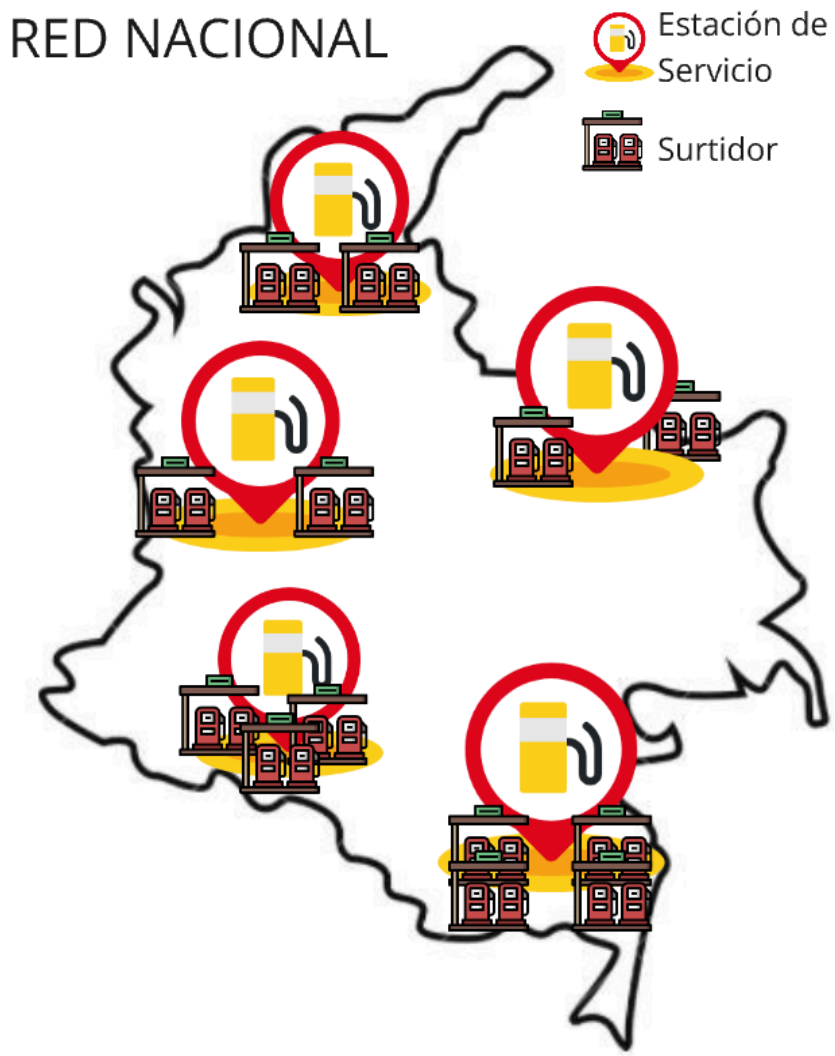
Surtidor representa los Surtidores asociados a una Estacion, y se encargan de realizar ventas y guardar los datos de estas.

Algunas otras "entidades" que fueron consideradas para ser clase, pero que al finalmente no se elaboraron, debido al cumplimiento de los criterios, fueron:

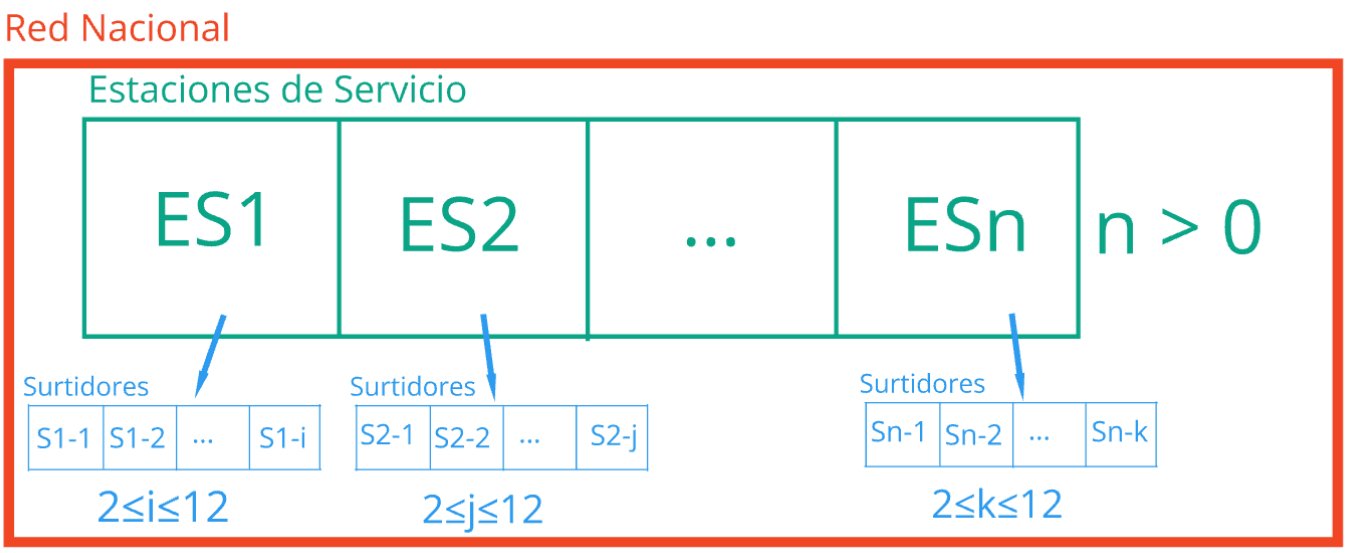
Islas: A pesar que se podrían considerar una clase, a efectos de este desafío no es el caso, puesto que no interactúan de ninguna manera con el resto del sistema, así que su implementación fue gestionada por medio de un arreglo de números que cambian los códigos de los surtidores.

Ventas: Si bien las ventas de combustible son una funcionalidad esencial y el almacenamiento de los datos de las ventas es parte fundamental del desarrollo del desafío, no hace falta crear un objeto solo para implementar getters y setters, esto se puede hacer directamente sobre un arreglo, por lo que la clase Ventas quedó descartada.

1.1 Representación gráfica del problema:



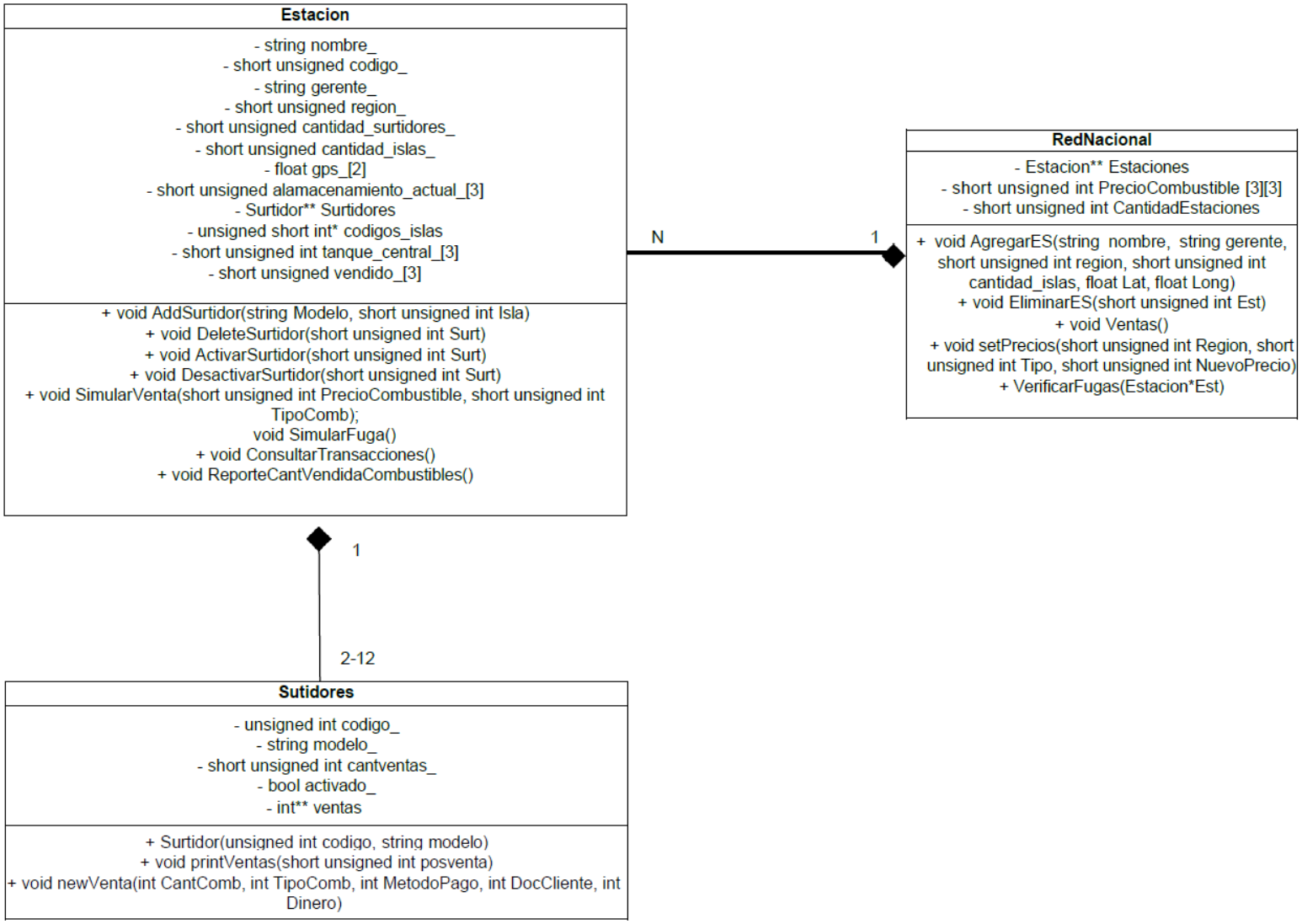
1.2 Representación del agrupamiento de clases:



2) Diagramas y análisis de alto nivel:

En primer lugar, se abordó el problema por medio de la creación de tres clases principales: RedNacional, Estacion y Surtidor, por medio de estas se da el funcionamiento de las estaciones de servicio, los surtidores y se busca cumplir cada requerimiento del programa.

2.1 Diagrama de clases:



2.2 Análisis de alto nivel:

DELETESURTIDOR
Elimina un surtidor desactivado de la estación. Primero, verifica si el surtidor pertenece a una isla que quedaría sin otros surtidores, y si es así, también elimina la isla. Luego, ajusta la memoria dinámica de los surtidores y actualiza los códigos de las islas para reflejar los cambios.

ADDSURTIDOR
Agrega un nuevo surtidor a la estación, generando su código basado en la estación y la isla asignada. Si no se especifica una isla, crea una nueva isla y actualiza los códigos de las islas. Luego, ajusta dinámicamente el arreglo de surtidores, agrega el nuevo surtidor, y actualiza la cantidad total de surtidores.

DESACTIVARSURTIDOR
Desactiva un surtidor si está activado, actualizando su estado a "desactivado" y mostrando un mensaje de confirmación. Si ya estaba desactivado, informa que el surtidor ya se encontraba en ese estado.

ACTIVARSURTIDOR
Verifica si el surtidor especificado está desactivado. Si es así, lo activa y muestra un mensaje confirmando la activación exitosa. Si ya estaba activado, informa que el surtidor ya se encontraba en ese estado.

CONSULTARTRANSACCIONES
Recorre todos los surtidores de la estación y muestra las transacciones registradas por cada uno. Si no hay ventas registradas en ningún surtidor, muestra un mensaje indicando que la estación no ha realizado ventas.

SIMULARVENTA
Genera una venta aleatoria de combustible, seleccionando una cantidad entre 3 y 20 litros. Luego, identifica los surtidores activos y los almacena en un arreglo, asegurándose de asignar la venta solo a un surtidor operativo.

SIMILARFUGA
Crea una fuga aleatoria de combustible, seleccionando el tipo y tamaño, sin exceder el combustible disponible. Luego, reduce el almacenamiento y muestra la cantidad de litros perdidos y el código de la estación.

PRINTVENTAS
Despliega los detalles de una venta específica en el surtidor, incluyendo fecha, combustible vendido, tipo, método de pago, y datos del cliente. Si no hay ventas o ya se han mostrado todas, notifica al usuario.

NEWVENTAS
Copia las ventas anteriores a un nuevo arreglo y añade la nueva venta, incluyendo la fecha, hora, cantidad de combustible, tipo, método de pago, documento del cliente y el monto.

AGREGARES
Crea una nueva estación en caso de que no haya otra con el mismo nombre o coordenadas GPS. Asigna un código único y ajusta el arreglo de estaciones, incrementando la cantidad total si es válida.

ELIMINARES
Elimina una estación si no tiene surtidores activos. Si la estación tiene surtidores activos, muestra un mensaje de error. Si es eliminable, ajusta dinámicamente el arreglo de estaciones y reduce la cantidad total.

VENTAS
Muestra un reporte de ventas por estación y por tipo de combustible, acumulando las ventas de cada surtidor y generando un total nacional. Al final, imprime los totales de ventas por estación y el resumen global para el país.

VERIFICARFUGAS
Comprueba si el combustible vendido y almacenado en los tanques es menor al 95% de su capacidad. Si se detecta una fuga, muestra el volumen y tipo de combustible perdido. Si no hay fuga, informa que la estación está en orden.

3) Algoritmos implementados:

Clase RedNacional: Se pensó en ella como la clase encargada de gestionar toda la red de estaciones de servicio en el país, lo que abarca la administración de las estaciones, los precios del combustible y las ventas realizadas en cada estación, de esta clase se resalta el uso de la memoria dinámica en estaciones y precios de combustible.

Se establecen como atributos de la clase un puntero a puntero de Estaciones, un arreglo de 3x3, que busca ajustar los precios del combustible dependiendo la región y un entero que dirá la cantidad de estaciones.

Con respecto a los métodos de la clase, dejando de lado los setters y getters incluidos, se busca hacer uso de AgregarES y EliminarES donde se permite la creación y eliminación de estaciones haciendo uso de la memoria dinámica.

Por otro lado, el método Ventas genera un reporte detallado de las ventas que se realizan en cada surtidor y discrimina el tipo de combustible, no obstante, este método interactúa con las clases Estación y Surtidor.

Por último, se desarrolló Verificar Fugas la cual comprueba si el combustible vendido y almacenado dentro de los tanques es inferior al 95% de su capacidad, si hay fuga lo informa, por el contrario si no hay una le informa al usuario la ausencia de fugas.

Clase Estacion: Se diseña como la clase con la labor de gestionar diversas operaciones en una estación de servicio, por medio de ella se administran los surtidores, el inventario de

combustible y registros de las transacciones de venta. Para la creación de surtidores en una estación se busca el uso de memoria dinámica.

En cuanto a los atributos de la clase, se incluyen diversos como los datos sobre la estación de servicios, sin embargo también se incluyen elementos clave, como la creación de un arreglo que representa la capacidad de los tanques dependiendo el tipo de combustible y otro arreglo para gestionar la capacidad en los tanques de almacenamiento, existen otros como un puntero dinámico hacia surtidores que conforman la estación.

Asimismo, en esta clase se permite la gestión dinámica de surtidores mediante los métodos AddSurtidor y DeleteSurtidor, los cuales agregan o eliminan surtidores según las decisiones del usuario.

El método SimularVenta registra ventas simuladas, asignando surtidores activos y actualizando el inventario de combustible tras cada venta. Esto permite mantener un control preciso de los niveles de combustible, además de ajustar la operatividad de los surtidores según sea el caso.

Por otro lado, existen los métodos de activar y desactivar surtidor, este que será relevante para cumplir el requisito de solo poder eliminar una estación en caso de tener todos sus surtidores desactivados.

No obstante, el método ConsultarTransacciones genera reportes detallados de las ventas realizadas, clasificándolas por surtidor y tipo de combustible.

Por último, se creó el método de SimularFuga, la cual crea una fuga en una estación, con un tipo de combustible aleatorio, asimismo determina el tamaño de la fuga aleatoriamente y verifica que no exceda la capacidad del tanque.

Clase Surtidor: Esta clase es la responsable de gestionar las ventas de cada surtidor dentro de una estación, por consiguiente los únicos atributos que se tienen son los datos del surtidor, un booleano que indica si el surtidor está activado y un arreglo de ventas.

Los métodos incluyen newVenta, que registra una nueva transacción, y printVentas, que permite visualizar los datos de cada venta realizada. Cada transacción almacenada contiene información como la fecha y hora de la venta, la cantidad de combustible vendida, el tipo de combustible, el método de pago y la identificación del cliente. Conforme se incrementa el número de ventas, el sistema adapta el tamaño de un arreglo que almacena dichas transacciones.

4) Problemas enfrentados en el desarrollo:

Uno de los problemas que enfrentamos fue a causa de ser nuestra primer experiencia rigurosa con la POO, por lo que definir qué iba en lo público y qué en lo privado fue la primera complicación, ya que no se debe dar la opción de modificar un atributo como el código de un surtidor fuera de la propia clase. Se tuvo que analizar bastante la necesidad de ciertos getters y setters y su ámbito (público o privado). Por ejemplo, el setActivado de un surtidor debe ser público, en vista de que es una decisión del usuario activar o desactivar un surtidor, pero un setCapacidadTanque, por ejemplo, no hace falta, visto que este valor después de la creación del objeto no debería poder ser modificado.

El segundo problema enfrentado fue en la creación del menú, considerando que al tener tantas opciones la realización de este resulta de gran extensión al momento de su elaboración. La comunicación con el usuario es, en este caso, la parte más larga del desarrollo y la implementación de un buen formato de

despliegue es requerido para que esta se dé de la manera más clara y precisa posible.

El último problema fue el correcto manejo de la memoria dinámica, específicamente a la hora de crear atributos que hacen uso de esta en los objetos, ya que es fácil olvidar liberar la memoria en el destructor y esto puede causar fugas de memoria.

5) Evolución y consideraciones

5.1 Evolución de la solución

5.2 Consideraciones

1. Tener claridad en la teoría de la POO, en razón de que la programación modular viene como consecuencia de este paradigma, y su implementación puede ser complicada en un principio, especialmente cuando uno se acostumbra a una programación más lineal, donde todo se realiza en un mismo archivo fuente y se tiene acceso a todo en todo momento. Esto último es especialmente importante en la POO ya que no siempre se tiene acceso a todos los datos de todos los objetos, específicamente, de los datos almacenados en el ámbito privado de la clase, por lo que una buena planeación se hace necesaria para mantener los datos protegidos, a la vez que la implementación de métodos que permitan acceder a los que hagan falta.

2. Debe ser necesario un buen manejo de la memoria dinámica, visto qué atributos que impliquen la creación de arreglos que puedan contener a otros objetos dentro de ellos implica el uso de memoria dinámica, y aún más importante que su uso, su correcta liberación.

3. Es importante destacar la importancia de la etapa de análisis ya que es fácil crear atributos o métodos innecesarios que nunca se utilizarán, o que guardan información que ya se tiene almacenada de manera implícita o explícita en otro lugar, y para aumentar la eficiencia del programa es de vital importancia que todos los datos almacenados en cada instancia de un objeto aprovechen al máximo el espacio en memoria que se les está otorgando.