**Informe del proyecto sobre desarrollo de un sistema que gestiona una red de estaciones de combustible.**

**Gestión dinámica a la red y sus estaciones utilizando el paradigma de programación orientada a objetos (POO).**

Autor: Martín Santiago Rodríguez Forero

1. **Análisis del problema:**

La empresa TerMax, lider en la industria de combustibles en Colombia requiere un sistema que permita una gestión eficiente de sus estaciones de combustible teniendo en cuenta diversos aspectos de las mismas que se mencionarán a continuación.

1. **Consideraciones para la alternativa de solución propuesta:**

* **Relaciones entre los objetos a gestionar:** Se evidencia que la empresa menciona una relación muy estrecha entre distintos elementos que deben ser tenidos en cuenta, tanto a la hora de realizar la codificación como para realizar los diagramas y esquemas solicitados.
* **Uso de POO:** Este permitirá una gestión más eficiente y cercana al mundo real para la solución de la problemática.
* **Uso de memoria dinámica:** Para la actualización del combustible dependiendo del caso, las modificaciones a los precios del combustible, la activación de surtidores de forma eficiente, entre otros.
* **Funcionalidades esenciales:** TerMax menciona varias funcionalidades que debe tener el programa bajo un menú con el que los usuarios puedan interactuar para realizar las diversas modificaciones que se requieran, las cuales son las siguientes:
* **Gestión de la red:**

- Agregar estaciones de servicio.

- Eliminar una E/S de la red nacional (sólo si no posee surtidores activos).

- Calcular el monto total de las ventas en cada E/S del país, discriminado por categoría de combustible.

- Fijar los precios del combustible para toda la red.

* **Gestión de estaciones de servicio:**

- Agregar/eliminar un surtidor a una E/S.

- Activar/desactivar un surtidor de una E/S.

- Consultar el histórico de transacciones de cada surtidor de la E/S.

- Reportar la cantidad de litros vendida según cada categoría de combustible.

- Simular una venta de combustible.

- Asignar la capacidad del tanque de suministro, con un valor aleatorio entre 100 y 200 litros para cada una de las categorías

* **Sistema nacional de verificación de fugas:**

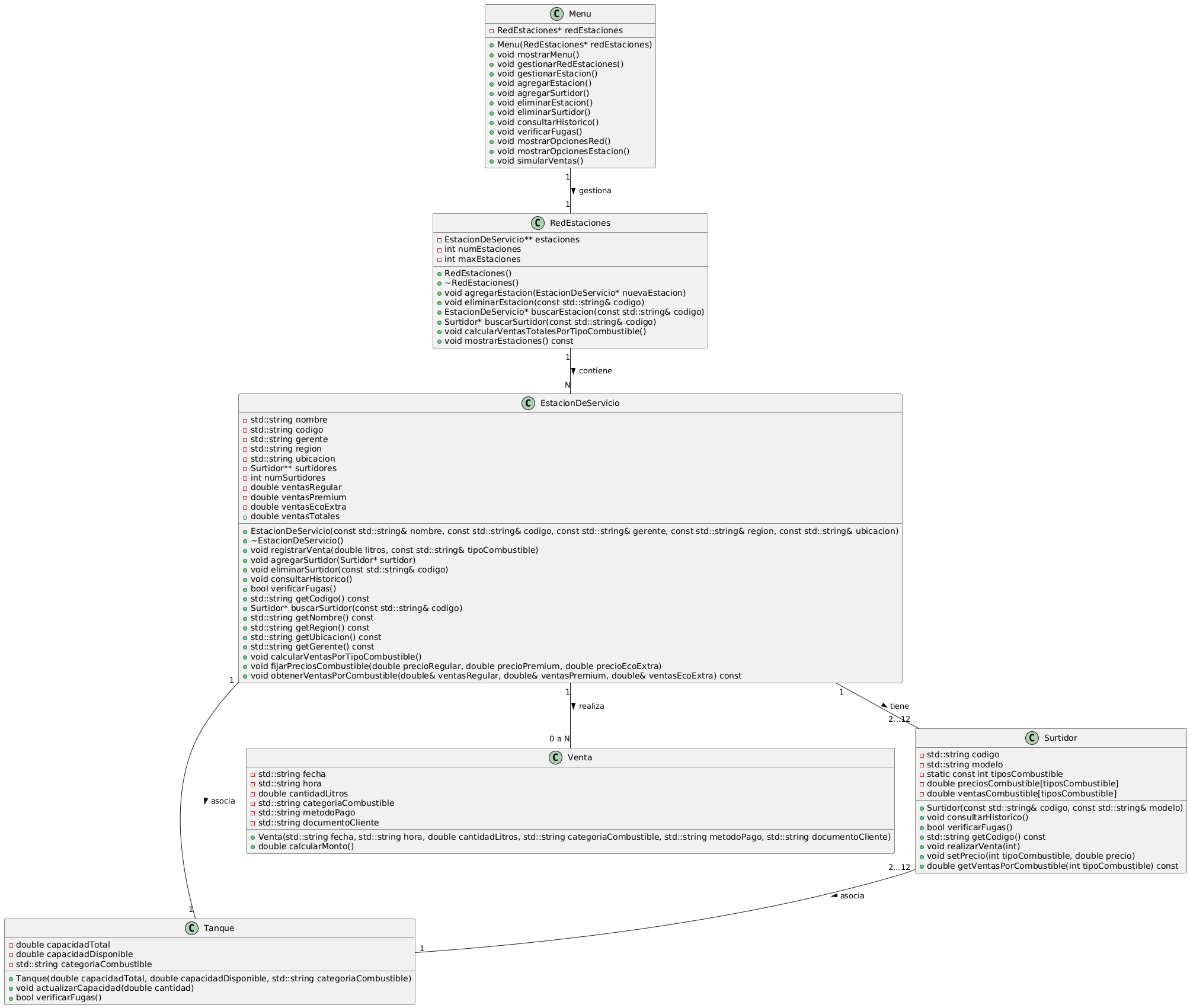
-El programa debe permitir detectar la existencia de fugas de combustible en cualquiera de las estaciones del país. Para ello, según cada categoría de combustible, debe verificarse que lo vendido más lo almacenado en el tanque de la E/S corresponda a más del 95% de la capacidad original del tanque. Esta verificación opera sobre una estación de servicio específica que sea seleccionada entre todas las del país.

* **Simulación de ventas:**

-Dada una E/S, asigne de forma aleatoria uno de los surtidores activos para que gestione la transacción. Suponga que en una venta se solicita en forma aleatoria entre 3 y 20 litros de gasolina. Una vez que se ha bombeado la gasolina deberán desplegarse los datos de la transacción. El precio por litro de cada categoría de combustible depende de la región a la que pertenece la estación. Sólo se consideran tres regiones: Norte, Centro y Sur. Dicho valor es temporal, pudiendo variar entre días.

* Por último se deben considerar subprogramas que gestionen todos los códigos de forma automática sin que estos impidan la interacción con los menús.

1. **Diagrama de clases de la solución planteada.**



### Algoritmos Implementados

A continuación se describen los principales algoritmos implementados en cada clase, con la documentación adecuada para cada uno.

#### Clase EstacionDeServicio:

#### EstacionDeServicio::EstacionDeServicio(const std::string& nombre, const std::string& codigo, const std::string& gerente, const std::string& region, const std::string& ubicacion)

* **Descripción**: Constructor de la clase. Inicializa una nueva estación de servicio con los parámetros proporcionados.

- void EstacionDeServicio::agregarSurtidor(Surtidor\* surtidor):

* **Descripción**: Agrega un nuevo surtidor a la estación. Verifica si hay espacio para más surtidores.
* **Funcionamiento:**
  1. Comprobar si el número actual de surtidores es menor que el máximo permitido.
  2. Si es posible, agregar el surtidor y aumentar el contador.

- void EstacionDeServicio::eliminarSurtidor(const std::string& codigo)

* **Funcionamiento:**
* Iterar sobre el arreglo de surtidores.
* Comparar el código de cada surtidor con el código proporcionado.
* Si se encuentra, liberar la memoria y desplazar los elementos restantes.

- Surtidor\* EstacionDeServicio::buscarSurtidor(const std::string& codigo)

* **Descripción**: Busca un surtidor por su código.
* **Algoritmo**:
* Iterar sobre el array de surtidores.
* Comparar el código de cada surtidor con el código proporcionado.
* Retornar el surtidor si se encuentra; si no, retornar nullptr.

#### 4.2 Clase Surtidor

- Surtidor::Surtidor(const std::string& codigo, const std::string& modelo)

* **Descripción**: Constructor de la clase Surtidor. Inicializa un nuevo surtidor.

- void Surtidor::consultarHistorico()

* **Descripción**: Método para consultar el histórico de ventas del surtidor.

- bool Surtidor::verificarFugas()

* **Descripción**: Verifica si el surtidor tiene fugas.

#### 4.3 Clase RedEstaciones

- void RedEstaciones::agregarEstacion(EstacionDeServicio\* nuevaEstacion)

* **Descripción**: Agrega una nueva estación a la red de estaciones.
* **Algoritmo**:
* Comprobar si el número actual de estaciones es menor que el máximo permitido.
* Si es posible, agregar la estación y aumentar el contador.

- Surtidor\* RedEstaciones::buscarSurtidor(const std::string& codigo)

* **Descripción**: Busca un surtidor en todas las estaciones de servicio.
* **Algoritmo**:
* Iterar sobre cada estación en la red.
* Llamar a buscarSurtidor en cada estación.
* Retornar el surtidor si se encuentra; si no, retornar nullptr.

Es necesario aclarar que tanto el diagrama de clases como los algoritmos implementados en este informe pueden tener modificaciones en un futuro.

1. **Dificultades de desarrollo:**

Durante el desarrollo del sistema de gestión de estaciones de servicio, se presentaron varios problemas que afectaron el avance del proyecto. Los más destacados son:

### a. Definición de Clases y Relaciones

* **Desafío**: Inicialmente, hubo dificultades en la definición clara de las clases y sus relaciones. Esto generó confusión sobre cómo estructurar los datos y cómo las distintas clases interactuarían entre sí.
* **Solución**: Se utilizó un diagrama de clases en PlantUML para visualizar las relaciones, lo que facilitó la identificación de los atributos y métodos necesarios para cada clase.

### b. Manejo de la Memoria Dinámica

* **Desafío**: El uso de memoria dinámica fue obligatorio, lo que llevó a complicaciones en la gestión de la memoria, como la liberación de recursos y la prevención de fugas de memoria.
* **Solución**: Se implementaron destructores apropiados y se realizaron pruebas exhaustivas para asegurar que toda la memoria se liberara correctamente.

### c. Validación de Datos de Entrada

* **Desafío**: Se presentó el problema de la validación de datos al ingresar información, especialmente en los campos críticos como el código de la estación y la región.
* **Solución**: Se implementaron bucles de validación que obligan al usuario a ingresar datos correctos antes de proceder, garantizando la integridad de los datos.

### d. Implementación de Funcionalidades Complejas

* **Desafío**: Algunas funcionalidades, como la verificación de fugas y la simulación de ventas, requirieron un diseño cuidadoso para asegurar que los cálculos y las lógicas estuvieran alineados con los requisitos del proyecto.
* **Solución**: Se realizó una planificación detallada de cada funcionalidad, y se implementaron pruebas unitarias para verificar su correcto funcionamiento.

## 6. Evolución de la Solución

El enfoque inicial del proyecto evolucionó significativamente a medida que se desarrollaba. A continuación, se describen las etapas clave de esta evolución:

### a. Planificación Inicial

* La primera fase se centró en definir las clases básicas y sus relaciones. Se identificaron los requisitos funcionales y no funcionales.

### b. Prototipo Inicial

* Se desarrolló un prototipo básico con las clases EstacionDeServicio, Surtidor, y Venta. Este prototipo permitía registrar ventas y gestionar surtidores.

### c. Mejora de Funcionalidades

* A medida que se realizaban pruebas, se añadieron características como la gestión de históricos de ventas, verificación de fugas y manejo de múltiples estaciones de servicio.

### d. Refactorización y Optimización

* Se revisó el código para optimizar el uso de memoria y mejorar la legibilidad. Se implementaron buenas prácticas de programación orientada a objetos y se introdujeron patrones de diseño donde fueron necesarios.

### e. Pruebas y Validación

* Se realizó un ciclo completo de pruebas para cada módulo, incluyendo pruebas unitarias y de integración, asegurando que todas las funcionalidades trabajaran de manera cohesiva.

## 3. Consideraciones para la Implementación

Al implementar el sistema de gestión de estaciones de servicio, se deben tener en cuenta las siguientes consideraciones:

### a. Escalabilidad

* El diseño del sistema debe permitir la fácil incorporación de nuevas funcionalidades y la capacidad de gestionar un número creciente de estaciones y ventas.

### b. Gestión de Errores

* Se debe implementar un manejo robusto de excepciones para prevenir fallos del sistema debido a entradas no válidas o problemas de comunicación entre clases.

### c. Documentación

* Es crucial mantener una documentación clara y concisa del código y de las funcionalidades implementadas para facilitar el mantenimiento y futuras expansiones del sistema.

### d. Interacción del Usuario

* Se debe considerar la experiencia del usuario al diseñar el menú y las interacciones, asegurando que el sistema sea intuitivo y fácil de usar.

### e. Pruebas Continuas

* Se realizan pruebas para identificar errores en la ejecución del programa.