UNIVERSIDAD DE ANTIOQUIA
FACULTA DE INGENIERÍA
PROGRAMA DE INGENIERÍA DE TELECOMUNICACIONES
SEMESTRE II

Desafío 2
Sistema de comercialización de combustible TerMax
Paradigma programación orientada a objetos - POO
Informe detallado del proyecto

Asignatura:
Informática II Teoría

Autores:
Oscar Miguel López Peñata
Luis Carlos Romero Cardenas

Tutores:
Anibal Guerra
Augusto Salazar

Medellín, Octubre de 2024

INTRODUCCIÓN

El presente informe aborda el desarrollo de un sistema de gestión para una red de estaciones de servicio de combustible, utilizando los principios de la programación orientada a objetos (POO). El objetivo es modelar un sistema realista que permita la administración eficiente de las estaciones, los surtidores de combustible y las transacciones de venta, asegurando un control preciso sobre los recursos disponibles y las ventas realizadas. La compañía TerMax, una de las principales proveedoras de combustible en Colombia, ha establecido una red de estaciones de servicio a nivel nacional, lo que implica la necesidad de un sistema robusto y flexible que facilite la supervisión y operación de cada estación.

En este contexto, cada estación de servicio cuenta con tanques de almacenamiento que distribuyen tres tipos de combustibles: Regular, Premium y EcoExtra. Los surtidores conectados a estos tanques se encargan de suministrar el combustible a los vehículos, registrando detalladamente las transacciones que incluyen información sobre la cantidad de combustible despachada, la categoría del producto, el método de pago y los datos del cliente. Adicionalmente, el sistema debe gestionar la capacidad de los tanques, calcular el monto total de las ventas discriminado por tipo de combustible y permitir la detección de posibles fugas, contribuyendo así a un control exhaustivo de los recursos.

El enfoque propuesto en este sistema incluye la implementación de múltiples funcionalidades basadas en los principios de abstracción, encapsulación y modularidad, características esenciales de la programación orientada a objetos. Entre las funcionalidades más relevantes se encuentran la gestión de la red nacional de estaciones, la activación y desactivación de surtidores, la simulación de ventas y la verificación de fugas de combustible. Además, el sistema permitirá la creación y eliminación de estaciones de servicio y surtidores, así como la asignación de precios del combustible según la región geográfica.

A través del desarrollo de este sistema, se busca no solo automatizar las operaciones de las estaciones de servicio, sino también optimizar la toma de decisiones a nivel gerencial, asegurando un mayor control sobre las ventas y la operación diaria de la red de estaciones de TerMax. El enfoque basado en la programación orientada a objetos facilitará la escalabilidad y el mantenimiento del sistema, permitiendo su adaptación futura a nuevos requerimientos y características.

PROPÓSITOS

General

Desarrollar un sistema de gestión integral basado en el paradigma de la Programación Orientada a Objetos para la red nacional de estaciones de servicio de TerMax, que automatice, supervise y controle eficientemente las operaciones diarias de venta de combustibles, optimizando el manejo de recursos, la administración de transacciones y la toma de decisiones estratégicas en tiempo real.

Específicos

Implementar la gestión de la red nacional de estaciones de servicio, que incluya la posibilidad de agregar o eliminar estaciones, así como fijar los precios del combustible según la región geográfica, garantizando la escalabilidad y flexibilidad del sistema.

Desarrollar un módulo para la administración de las estaciones de servicio, que permita agregar, activar o desactivar surtidores, consultar el historial de ventas y reportar la cantidad de litros vendidos por categoría de combustible en cada estación, asegurando un control preciso de las operaciones.

Crear un sistema de simulación de ventas de combustible, que asigne aleatoriamente un surtidor activo para realizar la transacción, registrando los datos de la venta, incluyendo la fecha, hora, cantidad y método de pago, con el fin de replicar las dinámicas comerciales reales.

Implementar un sistema de verificación de fugas de combustible, que compare la cantidad de combustible vendido y almacenado en los tanques para asegurar que corresponda con el inventario disponible, minimizando pérdidas y mejorando la seguridad en las estaciones de servicio.

Desarrollar un menú interactivo que permita acceder a las diferentes funcionalidades del sistema, facilitando la interacción del usuario con las opciones de gestión y simulación de manera amigable y eficiente.

Asegurar que el sistema sea modular y escalable, permitiendo futuras expansiones, como la incorporación de nuevas estaciones de servicio o surtidores, sin comprometer la funcionalidad propuesta.

DESCRIPCIÓN DEL PROBLEMA

La empresa TerMax, líder en la distribución de combustibles en Colombia, enfrenta un desafío importante en la gestión eficiente de su red nacional de estaciones de servicio. Actualmente, las operaciones relacionadas con la venta de combustibles, la administración de recursos y el control de transacciones se realizan de manera dispersa y sin un sistema integral que centralice la información. Este escenario genera problemas en la supervisión efectiva de las estaciones, dificultad en el control de inventarios, y una falta de visibilidad en tiempo real de las transacciones y el estado de los recursos.

Uno de los principales problemas identificados es la *falta de automatización* en las operaciones diarias de las estaciones de servicio, lo que obliga a los administradores locales a realizar procesos manuales de control, tales como la actualización del inventario de combustibles, el registro de ventas y la gestión de surtidores. Esto no solo incrementa la posibilidad de errores humanos, sino que también limita la capacidad de respuesta ante situaciones como la baja disponibilidad de combustible o el mal funcionamiento de surtidores.

Adicionalmente, *la carencia de un sistema de monitoreo centralizado* para las estaciones de servicio imposibilita una supervisión adecuada de los recursos y ventas a nivel nacional. Actualmente, no existe un mecanismo que permita a la gerencia central de TerMax tener un control detallado del inventario de cada estación, del rendimiento de los surtidores, ni de las ventas diarias, discriminadas por tipo de combustible y métodos de pago. Esta falta de visibilidad complica la toma de decisiones estratégicas, ya que se carece de información actualizada y precisa para ajustar precios, gestionar el suministro de combustible, o detectar posibles pérdidas, como fugas en los tanques de almacenamiento.

Otro problema clave *es la ausencia de un sistema para la detección automática de fugas de combustible*. En la situación actual, no se realiza un seguimiento adecuado de las discrepancias entre las cantidades de combustible despachadas y las disponibles en los tanques. Este vacío en el control puede resultar en pérdidas significativas para la empresa, además de presentar riesgos de seguridad.

Además, la *gestión manual de precios* por región y la incapacidad de realizar simulaciones de ventas son factores que incrementan la ineficiencia operativa. La variabilidad de los precios según la ubicación de las estaciones debería estar automatizada para evitar errores y facilitar el ajuste dinámico de los precios en función de la demanda y las condiciones de

mercado. De igual forma, la simulación de ventas permitiría prever escenarios futuros, evaluar el rendimiento de las estaciones y anticipar problemas operativos.

Por tanto, el problema principal se puede resumir en la necesidad de un *sistema de gestión integral* que permita a TerMax centralizar la operación de su red de estaciones de servicio, automatizar los procesos de venta, controlar el inventario de combustibles en tiempo real, detectar posibles fugas, y gestionar de manera eficiente los surtidores y las transacciones. Este sistema debe basarse en un enfoque robusto de programación orientada a objetos (POO), que permita un diseño modular, flexible y escalable, para que la solución sea adaptable a las futuras necesidades del negocio.

El desarrollo de este sistema abordará estos problemas mediante la creación de una plataforma que integre todas las operaciones relacionadas con la venta de combustibles, garantizando la integridad de los datos y proporcionando información clave para la toma de decisiones estratégicas, mejorando así la eficiencia operativa y reduciendo las pérdidas potenciales.

DESCRIPCIÓN DE TAREAS

Fecha	Nombre tarea	Actividades	Resultado
Octubre 9 a 10	**Análisis del problema:** Esta fase consiste en realizar una comprensión profunda del problema planteado, identificando los requerimientos clave del sistema y definiendo claramente las funcionalidades solicitadas.	Leer y analizar todo el documento del desafío. Identificar las entidades principales del sistema (estaciones de servicio, surtidores, transacciones, combustible). Definir los requisitos funcionales y no funcionales. Delimitar las relaciones entre los diferentes componentes del sistema (gestión de estaciones, surtidores, ventas y fugas de combustible).	Documento que describe claramente el análisis del problema y los requisitos del sistema.

Fecha	Nombre tarea	Actividades	Resultado
Octubre 10 a 11	**Análisis de Datos y Estructuras:** Determinar las estructuras de datos más adecuadas para representar los elementos del sistema (listas, mapas, arreglos dinámicos) y analizar la eficiencia de las mismas.	Identificar las estructuras de datos necesarias para representar estaciones, surtidores, transacciones y tanques de combustible. Elegir los tipos de datos correctos para atributos como cantidades de combustible, métodos de pago, precios por región, entre otros. Definir las funciones matemáticas o lógicas requeridas para cálculos como las ventas totales y la verificación de fugas.	Documento de especificación de los tipos de datos y estructuras de datos seleccionadas, con justificación para cada elección.

Fecha	Nombre tarea	Actividades	Resultado
Octubre 11 a 12	**Elaboración del Diagrama UML:** Crear un diagrama UML que represente visualmente las clases del sistema, sus atributos, métodos y relaciones entre ellas.	Diseñar las clases principales del sistema (RedNacional, EstacionServicio, Surtidor, Transaccion, Combustible). Definir las relaciones de agregación, asociación y dependencia entre las clases. Asegurarse de que todos los métodos públicos y privados estén correctamente representados en el diagrama. Revisar el diagrama para asegurar que todas las funcionalidades del sistema estén reflejadas en el diseño.	Diagrama UML completo y revisado, con una estructura clara de las clases y relaciones del sistema.

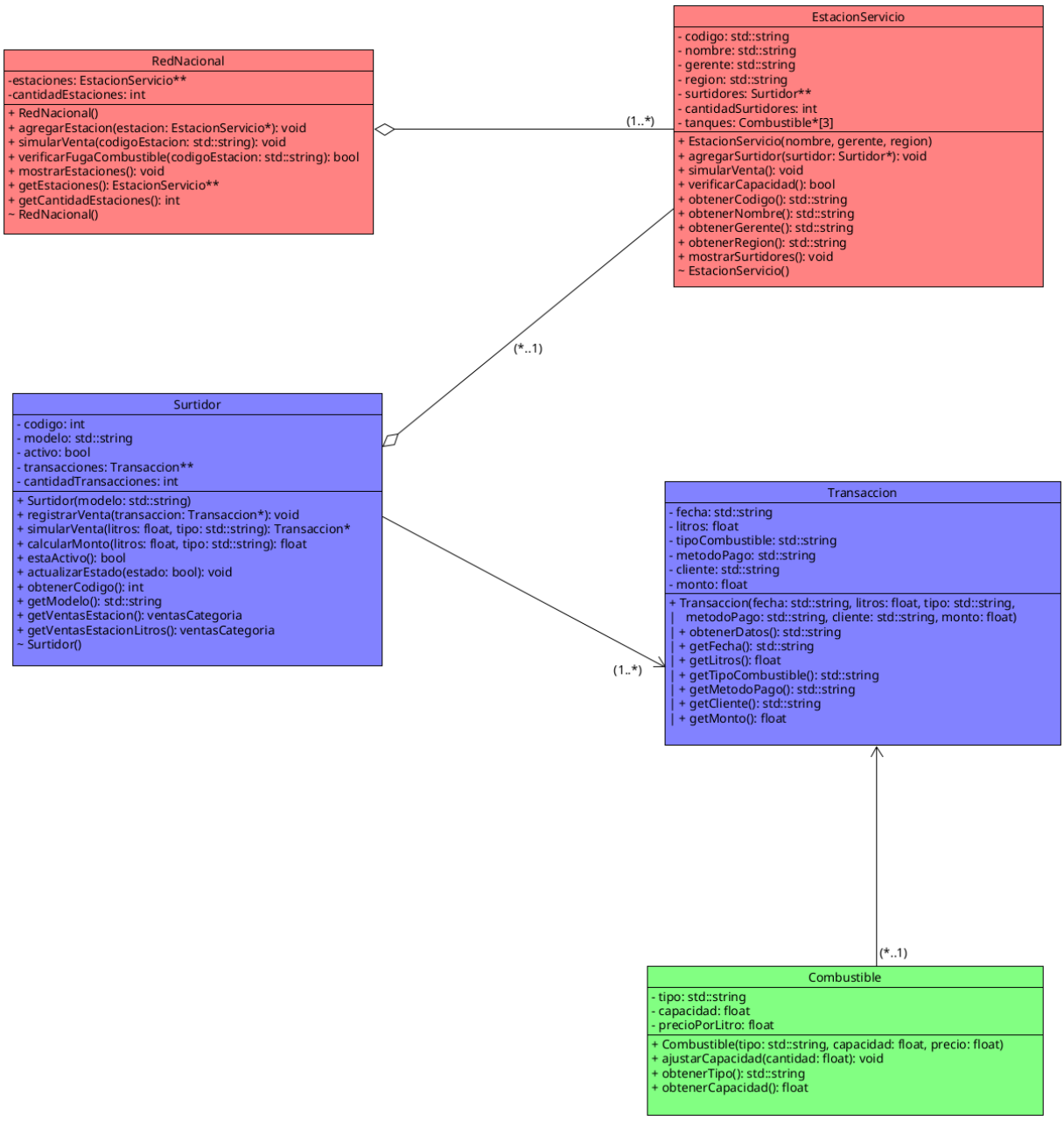
Fecha	Nombre tarea	Actividades	Resultado
Octubre 12	**Diseño de la implementación:** Basado en el análisis de datos y el diagrama UML, diseñar la implementación del sistema en C++, dividiendo el trabajo en módulos lógicos y definiendo los métodos y atributos necesarios.	Crear un esquema detallado de cómo se implementarán las funcionalidades clave del sistema (gestión de estaciones, ventas, verificación de fugas). Definir la interacción entre las clases a nivel de código. Especificar los constructores, getters, setters y funciones necesarias para que el sistema sea funcional.	Esquema de implementación detallado, listo para ser llevado a código.

Fecha	Nombre tarea	Actividades	Resultado
Octubre 12	**Creación del Repositorio en GitHub:** Configurar un espacio en GitHub para gestionar el código y el informe, haciendo commits regulares para documentar el progreso del desarrollo.	Crear el repositorio público en GitHub. Establecer un archivo README que incluya una breve descripción del proyecto y los requisitos del desafío. Hacer el primer commit con la estructura básica del proyecto (carpetas de código y documentos). Realizar commits regulares al menos una vez al día para documentar la evolución de la solución.	Repositorio de GitHub correctamente configurado y actualizado de manera regular.

Fecha	Nombre tarea	Actividades	Resultado
Octubre 13 a 15	**Implementación del sistema:** Convertir el diseño detallado en código funcional en C++, asegurando que todas las funcionalidades solicitadas estén implementadas de manera eficiente y que el sistema funcione de acuerdo a los requisitos.	Implementar la gestión de la red de estaciones (agregar/eliminar estaciones, fijar precios). Programar la administración de surtidores (agregar/eliminar surtidores, registrar ventas). Implementar la simulación de ventas con surtidores asignados aleatoriamente. Programar el sistema de verificación de fugas de combustible. Realizar pruebas unitarias y ajustes necesarios en el código.	Código C++ completo, funcional y documentado, implementando todas las funcionalidades requeridas.

Fecha	Nombre tarea	Actividades	Resultado
Octubre 16 a 17	**Elaboración del Video de Sustentaión:** Crear un video explicativo que resuma la solución planteada, con una presentación clara del análisis realizado, el diseño del sistema, la implementación, y ejemplos de su funcionamiento.	Presentar la estructura general del sistema y el diagrama UML. Explicar las principales funcionalidades implementadas en el sistema. Mostrar ejemplos en tiempo real de cómo funciona el sistema, simulando ventas y verificaciones. Justificar las elecciones técnicas realizadas durante el desarrollo (estructuras de datos, diseño POO, etc.).	Código C++ completo, funcional y documentado, implementando todas las funcionalidades requeridas.

DIAGRAMA DE CLASES



Explicación

Clase RedNacional (en rojo)

Esta clase gestiona una red de estaciones de servicio, permitiendo agregar o eliminar estaciones, así como realizar simulaciones y verificaciones en toda la red.

Atributos privados:

- estaciones: EstacionServicio: Un puntero a un array dinámico de estaciones de servicio. Cada estación está representada por una instancia de la clase EstacionServicio.
- cantidadEstaciones: int`: Número de estaciones de servicio que hay en la red.

Métodos públicos:

- RedNacional(): Constructor de la clase.
- agregarEstacion(estacion: EstacionServicio): Permite agregar una nueva estación de servicio a la red.
- simularVenta(codigoEstacion: std::string): Simula una venta en una estación específica de la red, identificada por su código.
- verificarFugaCombustible(codigoEstacion: std::string): Verifica posibles fugas de combustible en una estación específica.
- mostrarEstaciones(): Muestra todas las estaciones de servicio de la red.
- getEstacionES(): Retorna una estación de servicio específica.
- getCantidadEstaciones(): Retorna el número total de estaciones en la red.
- ~RedNacional(): Destructor de la clase.

Clase EstacionServicio (en rojo)

Esta clase representa una estación de servicio dentro de la red nacional.

Atributos privados:

- codigo: std::string: Código único que identifica a la estación de servicio.
- nombre: std::string: Nombre de la estación de servicio.
- gerente: std::string: Nombre del gerente responsable de la estación.
- region: std::string: Ubicación geográfica de la estación.
- surtidores: Surtidor: Un puntero a un array dinámico de surtidores, donde se almacenan los surtidores disponibles en la estación.
- cantidadSurtidores: int: Número de surtidores en la estación.
- tanques: Combustible[3]: Un array de objetos Combustible, que almacena los diferentes tipos de combustible disponibles en la estación (por ejemplo, gasolina, diesel, etc.).

Métodos públicos:

- EstacionServicio(nombre: std::string, gerente: std::string, region: std::string): Constructor de la clase que inicializa la estación con su nombre, gerente y región.
- agregarSurtidor(surtidor: Surtidor*): Agrega un nuevo surtidor a la estación.
- simularVenta(): Simula una venta de combustible en la estación.
- verificarCapacidad(): Verifica si la capacidad de los tanques es suficiente para realizar una venta.
- obtenerCodigo(): std::string: Retorna el código de la estación.
- obtenerNombre(): std::string: Retorna el nombre de la estación.
- obtenerGerente(): std::string: Retorna el nombre del gerente.
- obtenerRegion(): std::string: Retorna la región de la estación.
- mostrarSurtidores(): Muestra información sobre todos los surtidores de la estación.
- ~EstacionServicio(): Destructor de la clase.

Clase Surtidor (en azul)

Esta clase representa un surtidor de combustible dentro de una estación de servicio.

Atributos privados:

- codigo: int: Código único que identifica al surtidor.
- modelo: std::string: Modelo o tipo del surtidor.
- activo: bool: Estado del surtidor (si está activo o inactivo).
- transacciones: Transaccion**: Un puntero a un array dinámico de transacciones, que almacena las ventas realizadas por el surtidor.
- cantidadTransacciones: int: Número de transacciones realizadas por el surtidor.

Métodos públicos:

- Surtidor(modelo: std::string): Constructor de la clase que inicializa el surtidor con su modelo.
- registrarVenta(transaccion: Transaccion*): Registra una nueva venta o transacción en el surtidor.
- simularVenta(litros: float, tipo: std::string): Simula una venta de combustible en el surtidor.

- calcularMonto(litros: float, tipo: std::string): Calcula el monto total de una venta según el tipo y cantidad de combustible.
- actualizarEstado(estado: bool): Actualiza el estado del surtidor (activo/inactivo).
- obtenerCodigo(): int: Retorna el código del surtidor.
- getModelo(): std::string: Retorna el modelo del surtidor.
- getVentasEstacion(): ventasCategoria: Retorna las ventas realizadas por categoría de combustible.
- getVentasEstacionLitros(): ventasCategoria: Retorna los litros vendidos por categoría de combustible.
- ~Surtidor()`: Destructor de la clase.

Clase Transaccion (en azul)

Esta clase representa una transacción o venta de combustible en un surtidor.

Atributos privados:

- fecha: std::string: Fecha en que se realizó la transacción.
- litros: float: Cantidad de litros de combustible vendidos.
- tipoCombustible: std::string: Tipo de combustible vendido (por ejemplo, gasolina, diesel, etc.).
- metodoPago: std::string: Método de pago utilizado (por ejemplo, efectivo, tarjeta).
- cliente: std::string: Nombre del cliente que realizó la compra.
- monto: float`: Monto total de la transacción.

Métodos públicos:

- Transaccion(fecha: std::string, litros: float, tipo: std::string, metodoPago: std::string, cliente: std::string, monto: float): Constructor que inicializa la transacción con los detalles de la venta.
- obtenerDatos(): Muestra los detalles de la transacción.
- validarTransaccion(): Verifica si la transacción es válida.
- obtenerFecha(): std::string: Retorna la fecha de la transacción.
- getLitros(): float: Retorna la cantidad de litros vendidos.
- getTipoCombustible(): std::string: Retorna el tipo de combustible vendido.

- getMetodoPago(): std::string: Retorna el método de pago.
- getCliente(): std::string: Retorna el nombre del cliente.
- getMonto(): float: Retorna el monto total de la transacción.

Clase Combustible (en verde)

Esta clase representa los tipos de combustible disponibles en una estación de servicio.

Atributos privados:

- tipo: std::string: Tipo de combustible (por ejemplo, gasolina, diesel).
- capacidad: float: Capacidad del tanque de combustible en litros.
- precioPorLitro: float: Precio del combustible por litro.

Métodos públicos:

- Combustible(tipo: std::string, capacidad: float, precio: float): Constructor que inicializa el combustible con su tipo, capacidad y precio.
- ajustarCapacidad(cantidad: float): Ajusta la capacidad del tanque, sumando o restando una cantidad de litros.
- obtenerTipo(): std::string: Retorna el tipo de combustible.
- obtenerCapacidad(): float: Retorna la capacidad disponible en el tanque.
- setPrecio(float p): Ajusta el precio por litro del combustible.

Relaciones entre las clases

- RedNacional contiene y gestiona múltiples EstacionServicio, es decir, tiene una relación de 1 a muchos con la clase EstacionServicio.
- EstacionServicio contiene y gestiona múltiples Surtidor, representando también una relación de 1 a muchos.
- Surtidor puede tener múltiples Transaccion, indicando una relación de 1 a muchos con las transacciones.
- EstacionServicio contiene un array de tres objetos de la clase Combustible, que representan los tipos de combustible disponibles en la estación.

DOCUMENTACIÓN

Introducción

El sistema de gestión de estaciones de servicio está diseñado para facilitar la administración de múltiples estaciones de combustible en una red nacional. Permite a los administradores realizar operaciones como agregar nuevas estaciones, simular ventas de combustible y verificar la capacidad de los tanques de combustible. Este sistema ayuda a mantener un seguimiento de las transacciones y optimiza la gestión del combustible en las estaciones.

Propósitos

- Administrar Múltiples Estaciones: Permitir la creación y gestión de varias estaciones de servicio.
- Simular Ventas de Combustible: Proporcionar la funcionalidad para simular transacciones de venta de combustible.
- Verificar Capacidad: Monitorear la capacidad de los tanques de combustible y detectar posibles fugas.
- Registrar Transacciones: Mantener un registro de todas las ventas realizadas en cada estación.

Estructura del Proyecto

El proyecto está compuesto por varias clases que representan los elementos del sistema. A continuación se describen las clases y su funcionalidad:

Clases Principales

1. Combustible

Descripción: Representa un tipo de combustible disponible en la estación de servicio.

Atributos:

- std::string tipo: El tipo de combustible (e.g., Regular, Premium).
- float capacidad`: La cantidad de combustible disponible en litros.

- float precioPorLitro: El precio por litro del combustible.

Métodos:

- Combustible(std::string tipo, float capacidad, float precio): Constructor que inicializa los atributos.
- void ajustarCapacidad(float cantidad): Ajusta la capacidad del combustible tras una venta.
- std::string obtenerTipo() const: Devuelve el tipo de combustible.
- float obtenerCapacidad() const: Devuelve la capacidad actual del combustible.

2. Transaccion

Descripción: Representa una transacción de venta de combustible en una estación de servicio.

Atributos:

- std::string fecha: La fecha de la transacción.
- float litros: La cantidad de litros vendidos.
- std::string tipoCombustible: El tipo de combustible vendido.
- std::string metodoPago: El método de pago utilizado.
- std::string cliente: El nombre del cliente.
- float monto`: El monto total de la transacción.

Métodos:

- Transaccion(std::string fecha, float litros, std::string tipoCombustible, std::string metodoPago, std::string cliente, float monto): Constructor que inicializa los atributos.
- std::string obtenerDatos() const: Devuelve un string con los detalles de la transacción.
- Getters para acceder a los atributos.

3. Surtidor

Descripción: Representa un surtidor de combustible en una estación de servicio.

Atributos:

- static int generadorCodigo: Contador para generar códigos únicos de surtidores.

- int codigo: Código único del surtidor.
- std::string modelo: Modelo del surtidor.
- bool activo: Estado del surtidor (activo o inactivo).
- Transaccion** transacciones: Puntero a un arreglo dinámico de transacciones.
- int cantidadTransacciones: Número de transacciones registradas.

Métodos:

- Surtidor(std::string modelo): Constructor que inicializa los atributos.
- void registrarVenta(Transaccion* transaccion): Registra una nueva transacción.
- Transaccion simularVenta(float litros, std::string tipo): Simula una venta de combustible.
- float calcularMonto(float litros, std::string tipo): Calcula el monto total de la venta.
- Getters para acceder a la información del surtidor.

4. EstacionServicio

Descripción: Representa una estación de servicio en la red.

Atributos:

- static int generadorCodigoEstacion: Contador para generar códigos únicos de estaciones.
- std::string codigo: Código único de la estación.
- std::string nombre: Nombre de la estación.
- std::string gerente: Nombre del gerente de la estación.
- std::string region: Región donde se encuentra la estación.
- Combustible* tanques[3]: Arreglo de tanques de combustible.
- Surtidor** surtidores: Puntero a un arreglo dinámico de surtidores.
- int cantidadSurtidores: Número de surtidores registrados.

Métodos:

- EstacionServicio(std::string nombre, std::string gerente, std::string region): Constructor que inicializa los atributos.
- void agregarSurtidor(Surtidor* surtidor): Agrega un nuevo surtidor a la estación.
- void simularVenta(): Simula una venta de combustible.

- bool verificarCapacidad() const: Verifica la capacidad total de los tanques de combustible.
- Getters para acceder a la información de la estación.

5. RedNacional

Descripción: Representa la red de estaciones de servicio.

Atributos:

- EstacionServicio** estaciones: Puntero a un arreglo dinámico de estaciones de servicio.
- int cantidadEstaciones: Número de estaciones registradas.

Métodos:

- RedNacional(): Constructor que inicializa los atributos.
- void agregarEstacion(EstacionServicio* estacion): Agrega una nueva estación a la red.
- void simularVenta(const std::string& codigoEstacion): Simula una venta en la estación correspondiente.
- bool verificarFugaCombustible(const std::string& codigoEstacion): Verifica si hay fugas de combustible en la estación.
- void mostrarEstaciones() const: Muestra las estaciones registradas.

Ejecución del Programa

1. Inicialización del Sistema:

Se crea un objeto RedNacional.

2. Menú Interactivo:

Se presenta un menú que permite al usuario elegir entre las siguientes opciones:

- Agregar una nueva estación de servicio.
- Mostrar las estaciones de servicio existentes.
- Simular una venta de combustible en una estación específica.
- Verificar la capacidad de los tanques de combustible en una estación.
- Salir del programa.

3. Operaciones:

Dependiendo de la opción seleccionada, el programa ejecuta la función correspondiente, solicitando al usuario la información necesaria y realizando las operaciones pertinentes.

Ejemplo de Uso

A continuación se muestra un ejemplo de cómo usar el sistema:

1. El usuario elige Agregar estación de servicio e ingresa los detalles.
2. Se simula una venta de combustible en una estación específica.
3. Se verifica si hay fugas de combustible en la estación seleccionada.
4. Finalmente, se puede mostrar todas las estaciones registradas en la red.

Conclusión

Este software proporciona una solución integral para la gestión de estaciones de servicio, facilitando la administración de transacciones de combustible y monitoreo de capacidad. Permite a los administradores de estaciones optimizar la operación y asegurar el control sobre los recursos.

Contacto

Luis Carlos Romero Cardenas
email: luis.romero3@udea.edu.co

Oscar Miguel Lopez Peñata
email: oscar.lopezp@udea.edu.co

Universidad de Antioquia
Asignatura: Informática 2 Teoría
Programa: Ingeniería de Telecomunicaciones

CAPTURAS DE PANTALLA

Menú principal

main.cpp

```
285 */
286 // Funciones del menú
287 void mostrarMenu() {
288     std::cout << "============================\n";
289     std::cout << "       *** MENU PRINCIPAL ***       \n";
290     std::cout << "============================\n";
291     std::cout << "1.  Agregar estación de servicio\n";
292     std::cout << "2.  Mostrar estaciones de servicio\n";
293     std::cout << "3.  Simular venta de combustible\n";
294     std::cout << "4.  Verificar fuga de combustible\n";
295     std::cout << "5.  Agregar surtidor\n";
296     std::cout << "6.  Calcular el monto total de las ventas por categoría a nivel nacional\n";
297     std::cout << "7.  Histórico de transacciones por surtidor\n";
298     std::cout << "8.  Litros vendidos por categoría\n";
299     std::cout << "9.  Cambiar estado de un surtidor\n";
300     std::cout << "10. Mostrar surtidores\n";
301     std::cout << "11. Salir\n";
302     std::cout << "============================\n";
303     std::cout << "Ingrese su opción: ";
304 }
305
```

Clase rednacional

rednacional.cpp

```
#include "rednacional.h"
#include <iostream>

RedNacional::RedNacional() : estaciones(nullptr), cantidadEstaciones(0) {}

void RedNacional::agregarEstacion(EstacionServicio* estacion) {
    EstacionServicio** nuevo = new EstacionServicio*[cantidadEstaciones + 1];
    for (int i = 0; i < cantidadEstaciones; ++i) {
        nuevo[i] = estaciones[i];
    }
    nuevo[cantidadEstaciones] = estacion;
    delete[] estaciones;
    estaciones = nuevo;
    cantidadEstaciones++;
}

void RedNacional::simularVenta(const std::string& codigoEstacion) {
    for (int i = 0; i < cantidadEstaciones; ++i) {
        if (estaciones[i]->obtenerCodigo() == codigoEstacion) {
            estaciones[i]->simularVenta();
            return;
        }
```

Clase estacionservicio

```cpp
#include "EstacionServicio.h"
#include "archivo.h"
#include "utils.h"
#include <cstdlib>
#include <iostream>

int EstacionServicio::generadorCodigoEstacion = 1000;

EstacionServicio::EstacionServicio(std::string nombre, std::string gerente, std::string region)
    : codigo(generarCodigo()), nombre(nombre), gerente(gerente), region(region), surtidores(nullptr), cantidadSurtidores(0) {
    capacidadInicialTanque[0]=generarNumeroAleatorio(100,200);
    capacidadInicialTanque[1]=generarNumeroAleatorio(100,200);
    capacidadInicialTanque[2]=generarNumeroAleatorio(100,200);
    tanques[0] = new Combustible("Regular",capacidadInicialTanque[0] , 150);
    tanques[1] = new Combustible("Premium", capacidadInicialTanque[1], 100);
    tanques[2] = new Combustible("EcoExtra", capacidadInicialTanque[2], 120);
}

std::string EstacionServicio::generarCodigo() {
    return "E-" + std::to_string(++generadorCodigoEstacion);
}

```

Clase surtidor

```cpp
#include "Surtidor.h"
#include "utils.h"
#include <cstdlib>

int Surtidor::generadorCodigo = 0;



Surtidor::Surtidor(std::string modelo)
    : codigo(++generadorCodigo), modelo(modelo), activo(true),
    transacciones(nullptr), cantidadTransacciones(0) {}

void Surtidor::registrarVenta(Transaccion* transaccion) {
    Transaccion** nuevo = new Transaccion*[cantidadTransacciones + 1];
    for (int i = 0; i < cantidadTransacciones; ++i) {
        nuevo[i] = transacciones[i];
    }
    nuevo[cantidadTransacciones] = transaccion;
    delete[] transacciones;
    transacciones = nuevo;
    cantidadTransacciones++;
}
```

Clase transaccion

```cpp
#include "transaccion.h"

Transaccion::Transaccion(std::string fecha, float litros, std::string tipoCombustible,
                         std::string metodoPago, std::string cliente, float monto)
    : fecha(fecha), litros(litros), tipoCombustible(tipoCombustible),
    metodoPago(metodoPago), cliente(cliente), monto(monto) {}

std::string Transaccion::obtenerDatos() const {
    return "Fecha: " + fecha + ", Litros: " + std::to_string(litros) +
           ", Tipo: " + tipoCombustible +", Método de pago: "+metodoPago+ ", Cliente: " + cliente +
           ", Monto: " + std::to_string(monto);
}

// Getters implementation

std::string Transaccion::getFecha() const {
    return fecha;
}

float Transaccion::getLitros() const {
    return litros;
}
```

Clase combustible

```cpp
#include "Combustible.h"

Combustible::Combustible(std::string tipo, float capacidad, float precio)
    : tipo(tipo), capacidad(capacidad), precioPorLitro(precio) {}

void Combustible::ajustarCapacidad(float cantidad) {
    capacidad -= cantidad;
}

std::string Combustible::obtenerTipo() const {
    return tipo;
}

float Combustible::obtenerCapacidad() const {
    return capacidad;
}
```

PROBLEMAS DE DESARROLLO

Verificación de capacidad de combustible: Riesgo de ventas superiores a la cantidad disponible.

Verificación de fugas de combustible: Posibles pérdidas no detectadas si no se monitorea adecuadamente.

Simulación de ventas: Problemas en la asignación aleatoria de surtidores o inconsistencias en la cantidad vendida.

Actualización de precios: Posible inconsistencia en los precios por región si no se gestiona correctamente.

Gestión de códigos de surtidores: Posible duplicación de códigos o errores en la activación/desactivación de surtidores.

Historial de transacciones: Riesgo de sobrecarga del sistema o pérdida de información valiosa si no se maneja adecuadamente.

Asignación de capacidad de tanques: Riesgo de asignaciones ineficientes que afecten la operación de las estaciones.

Cálculo del monto de ventas: Posibilidad de errores financieros si el cálculo no es preciso.

Manejo de memoria dinámica: Problemas de rendimiento o errores de memoria si no se gestiona correctamente.

Estos problemas potenciales requieren especial atención en el diseño y desarrollo del sistema, para asegurar que el sistema de comercialización de combustible funcione de manera eficiente y cumpla con los requerimientos del desafío.