Universidad de Antioquia
Departamento de Telecomunicaciones
2598521 - Informática II.

Informe Final

Informe final Desafío II.

Elvia Marina Gómez Palmar

17 de octubre de 2024

1. Introducción

En el presente informe se detalla la solución del desafío 2, la evolución y los problemas que surgieron a lo largo de la solución y cómo fueron abordados.

2. Detalles

En un principio se planteó crear 3 clases para la resolución del problema: Red Nacional; Estaciones de Servicio y Surtidores. Se escogieron estos elementos como clases basados en la definición de clase y el uso que se le darán a estas durante la ejecución del programa. Como se presenta en eñ diagrama UML, estos fueron los métodos y clases que fueron seleccionados al comienzo del desafío.

Esto representaba una idea muy general basada en las primeras lecturas que se hicieron del documento, sin embargo, la cantidad de métodos se fueron incrementando en mayor proporción que los atributos definidos inicialmente, esto incluyendo a lo métodos setters y getters. También se hizo uso en su mayoría de los arreglos de tipo ¡vector¿, apuntadores para poder acceder a métodos, y métodos propios de c++, por ejemplo ¡limits¿o ¡regex¿.

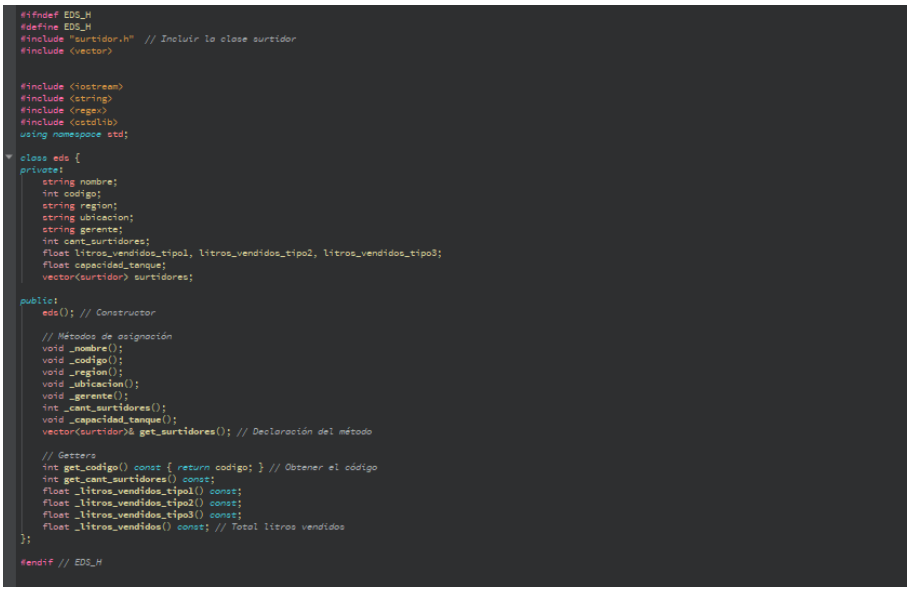
3. Solución de problemas no triviales.

Uno de los retos más grandes encontrados a la hora del desarrollo del programa fue la decidir la manera idónea para gestionar los datos o tener el historial de las transacciones. La solución a esto en cuanto al historial de transacciones fue almacenar en un arreglo [lista] la información aquí contenida y escribirla en un archivo.txt, la información sería agregada permanentemente, y la memoria liberada una vez esto se haya hecho.
Por otra parte también se planteó la incógnita: ¿Cómo activar y desactivar un surtidor sin eliminar sus datos?, ¿Cómo tener esto en cuenta a la hora de realizar una venta? Bien, para esto se pensó en la creación de dos listas de códigos, una para surtidores activos y otra para surtidores inactivos, cuando se deseara desactivar un surtidor solo bastaría con escribir su código, el programa lo buscaría en la lista y lo agregaría a surtidores inactivos. De esta manera, estaba cubierto el hecho de no realizar ventas con surtidores inactivos, además, de ser fundamental para el menú de la red nacional, ya que, si se deseaba eliminar alguna estación de servicio ésta no debía tener surtidores activos.

```cpp
// Eliminar estación de servicio
void red_nacional::eliminar_estacion() {
    int codigo;
    cout << "Ingrese el código de la estación a eliminar: ";
    cin >> codigo;

    // Buscar la estación por código
    for (auto it = estaciones_servicio.begin(); it != estaciones_servicio.end(); ++it) {
        if (it->get_codigo() == codigo) {
            // Verificar si la estación tiene surtidores activos
            vector<surtidor>& surtidores = it->get_surtidores(); // Obtener los surtidores de la estación

            // Aquí asumimos que la variable estado del surtidor indica si está activo
            bool tiene_surtidores_activos = false;
            for (const surtidor& s : surtidores) {
                // Verificar si el surtidor está activo (deberías tener un método que lo determine)
                if (s.esta_activo()) { // Asegúrate de que este método esté implementado en surtidor
                    tiene_surtidores_activos = true;
                    break;
                }
            }

            if (!tiene_surtidores_activos) {
                // Si no tiene surtidores activos, eliminar la estación
                estaciones_servicio.erase(it);
                cout << "Estación eliminada con éxito." << endl;
            } else {
                cout << "Error: No se puede eliminar la estación, tiene surtidores activos." << endl;
            }
            return;
        }
```

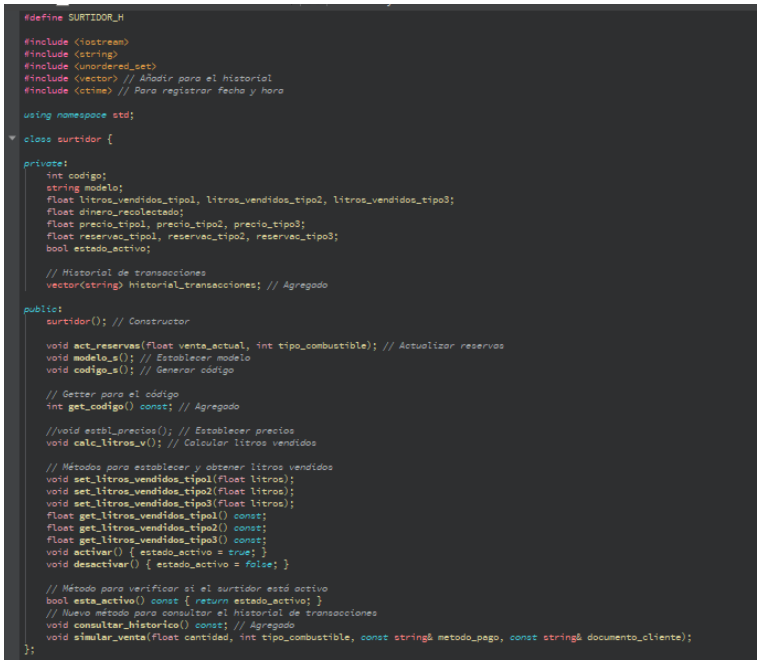
Figura 1: Ejemplo práctico en los métodos de las estaciones de servicios.

A continuación, las clases definidas para la solución.



```cpp
#ifndef EDS_H
#define EDS_H
#include "surtidor.h"  // Incluir la clase surtidor
#include <vector>


#include <iostream>
#include <string>
#include <regex>
#include <cstdlib>
using namespace std;

class eds {
private:
    string nombre;
    int codigo;
    string region;
    string ubicacion;
    string gerente;
    int cant_surtidores;
    float litros_vendidos_tipo1, litros_vendidos_tipo2, litros_vendidos_tipo3;
    float capacidad_tanque;
    vector<surtidor> surtidores;

public:
    eds(); // Constructor

    // Métodos de asignación
    void _nombre();
    void _codigo();
    void _region();
    void _ubicacion();
    void _gerente();
    int _cant_surtidores();
    void _capacidad_tanque();
    vector<surtidor>& get_surtidores(); // Declaración del método

    // Getters
    int get_codigo() const { return codigo; } // Obtener el código
    int get_cant_surtidores() const;
    float _litros_vendidos_tipo1() const;
    float _litros_vendidos_tipo2() const;
    float _litros_vendidos_tipo3() const;
    float _litros_vendidos() const; // Total litros vendidos
};

#endif // EDS_H
```

Figura 2: Clase EDS.



```cpp
#define SURTIDOR_H

#include <iostream>
#include <string>
#include <unordered_set>
#include <vector> // Añadir para el historial
#include <ctime> // Para registrar fecha y hora

using namespace std;

class surtidor {

private:
    int codigo;
    string modelo;
    float litros_vendidos_tipo1, litros_vendidos_tipo2, litros_vendidos_tipo3;
    float dinero_recolectado;
    float precio_tipo1, precio_tipo2, precio_tipo3;
    float reservac_tipo1, reservac_tipo2, reservac_tipo3;
    bool estado_activo;

    // Historial de transacciones
    vector<string> historial_transacciones; // Agregado

public:
    surtidor(); // Constructor

    void act_reservas(float venta_actual, int tipo_combustible); // Actualizar reservas
    void modelo_s(); // Establecer modelo
    void codigo_s(); // Generar código

    // Getter para el código
    int get_codigo() const; // Agregado

    //void estbl_precios(); // Establecer precios
    void calc_litros_v(); // Calcular litros vendidos

    // Métodos para establecer y obtener litros vendidos
    void set_litros_vendidos_tipo1(float litros);
    void set_litros_vendidos_tipo2(float litros);
    void set_litros_vendidos_tipo3(float litros);
    float get_litros_vendidos_tipo1() const;
    float get_litros_vendidos_tipo2() const;
    float get_litros_vendidos_tipo3() const;
    void activar() { estado_activo = true; }
    void desactivar() { estado_activo = false; }

    // Método para verificar si el surtidor está activo
    bool esta_activo() const { return estado_activo; }
    // Nuevo método para consultar el historial de transacciones
    void consultar_historico() const; // Agregado
    void simular_venta(float cantidad, int tipo_combustible, const string& metodo_pago, const string& documento_cliente);
};
```

Figura 3: Clase EDS.

```cpp
#ifndef RED_NACIONAL_H
#define RED_NACIONAL_H

#include "eds.h"
#include <vector>

using namespace std;

class red_nacional {
private:
    string pais;
    float ventas;
    double precio_ctipo1, precio_ctipo2, precio_ctipo3;
    bool estaciones_activa;
    vector<eds> estaciones_servicio; // Vector de estaciones de servicio

public:
    red_nacional(); // Constructor
    vector<double> fijar_precios(); //CON ESTO OBTENGO UNA LISTA CON LOS PRECIOS
    // Métodos para manejar la red nacional
    void calcular_ventas(); // Calcular ventas por tipo de combustible
    void agregar_estacion(); // Agregar estación
    void eliminar_estacion(); // Eliminar estación
    vector<eds>& get_estaciones_servicio() {
        return estaciones_servicio;
    }
};

#endif // RED_NACIONAL_H
```

Figura 4: Clase red nacional.

En cuanto a la simulación de ventas se escogió definir el método dentro de los métodos surtidores, ya que es más fácil acceder al método desde acá.

```cpp
void surtidor::simular_venta(float cantidad, int tipo_combustible, const string& metodo_pago, const string& documento_cliente) {
    float precio = 0.0; // Suponemos que se establecerá el precio correcto basado en el tipo
    switch (tipo_combustible) {
    case 1: precio = precio_tipo1; break;
    case 2: precio = precio_tipo2; break;
    case 3: precio = precio_tipo3; break;
    default: return; // Manejo de error, tipo de combustible no válido
    }

    // Actualizar reservas
    act_reservas(cantidad, tipo_combustible);

    // Calcular el total
    float total = cantidad * precio;

    // Registrar la transacción
    time_t now = time(0);
    char* dt = ctime(&now); // Obtener la fecha y hora actual
    string transaccion = "Fecha: " + string(dt) +
                         ", Cantidad: " + to_string(cantidad) +
                         ", Tipo: " + to_string(tipo_combustible) +
                         ", Pago: " + metodo_pago +
                         ", Documento: " + documento_cliente +
                         ", Total: " + to_string(total);

    historial_transacciones.push_back(transaccion);
    dinero_recolectado += total; // Actualizar el total recolectado
}
```

Figura 5: Simulador de ventas.

Otra parte importante a la hora de implementar las soluciones fue el orden de implementación, ya que, al empezar a definir los métodos para la clase red nacional, era mucho más difícil continuar porque necesitaba de muchos de los datos de las otras clases; y la idea de adaptar todo a una parte del código no parecía lo más eficiente, así que por eso se empezó definiendo los métodos para los surtidores, luego de las EDS y por último la red nacional.

4. Diagrama UML INICIAL.

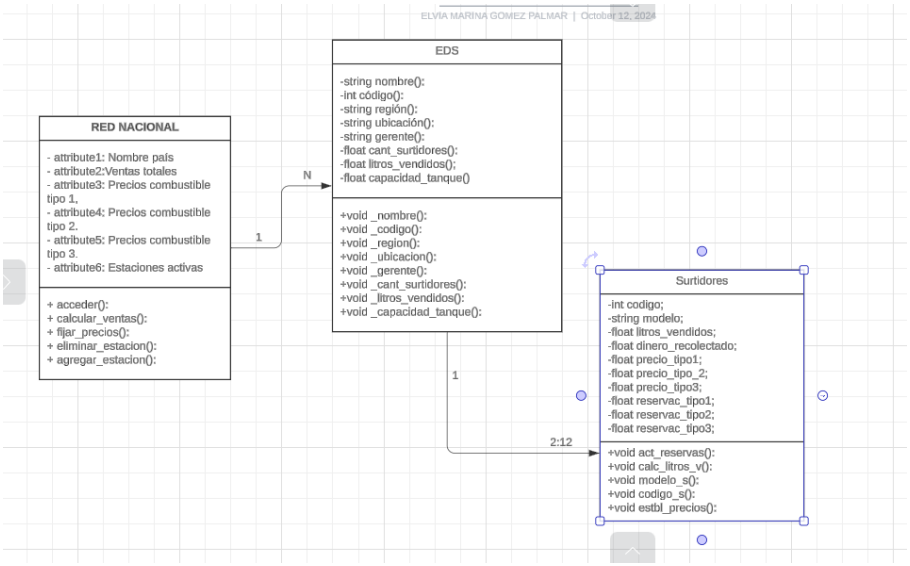


Figura 6: Diagrama de las clases implementadas para la resolución del desafío.

5. Puntos importantes.

Para que el proyecto sea óptimo aún falta integrar el menú en su totalidad con las respectivas funciones, esto, por motivos de tiempo no pudo concretarse, pero se manifiesta en este informe ya que es una parte importante que no debe ser obviada.