Informe del Desafío 2 Informática 2

2024-2

Diego Alejandro Alegría Cano

Felipe Rodas Vásquez

Facultad de Ingeniería, Universidad de Antioquia

Informática II

Dr. Aníbal José Guerra Soler

12 de Octubre

Análisis del problema.

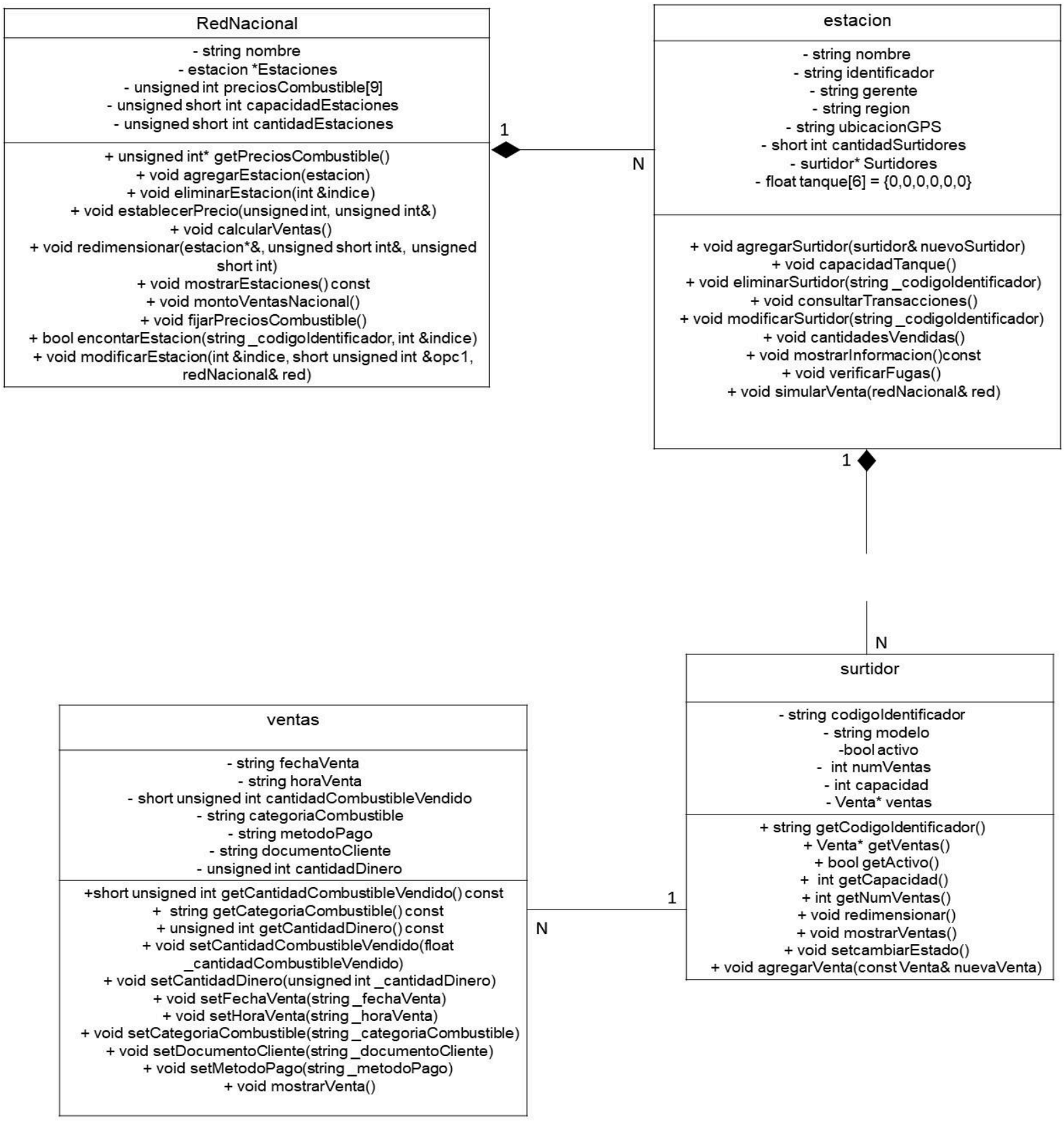
TerMax, una compañía líder en el suministro de combustibles en Colombia, necesita un sistema de gestión para su red de estaciones de servicio. Cada estación tiene un nombre, código identificador, gerente, región y ubicación GPS, y almacena tres tipos de combustible (Regular, Premium y EcoExtra) en un tanque central que abastece de 2 a 12 surtidores. Estos surtidores, agrupados en naves o islas, venden combustible directamente a los clientes, registrando cada venta con detalles como la fecha, hora, cantidad, categoría, método de pago, documento del cliente y monto.

Las ventas impactan la disponibilidad de combustible en el tanque central, debiendo ajustarse automáticamente si la demanda excede la oferta. La gestión del sistema implica el monitoreo y actualización de estos niveles de combustible, la gestión de surtidores y la consolidación de datos de ventas.

La solución planteada para el Desafío es modelar una red nacional de gasolineras mediante el paradigma de la Programación orientada a Objetos, con la cual crearemos diferentes tipos de clases las cuales van a ser: Red nacional (clase en la cual va a existir una sola instancia), estaciones de servicios, surtidores y ventas. Cada una de estas clases contará con sus respectivos atributos y métodos que serán declarados de forma pública o privada. Además también se implementará un sistema de verificación de fugas nacional que servirá para ver en cada una de las estaciones si hay fugas de combustibles.

Diagrama de clases.

A continuación se mostrará el diagrama de clases final que utilizamos para la planificación, desarrollo y ejecución de la solución final del problema.


RedNacional
- string nombre
- estacion *Estaciones
- unsigned int preciosCombustible[9]
- unsigned short int capacidadEstaciones
- unsigned short int cantidadEstaciones
+ unsigned int* getPreciosCombustible()
+ void agregarEstacion(estacion)
+ void eliminarEstacion(int &indice)
+ void establecerPrecio(unsigned int, unsigned int&)
+ void calcularVentas()
+ void redimensionar(estacion*&, unsigned short int&, unsigned short int)
+ void mostrarEstaciones() const
+ void montoVentasNacional()
+ void fijarPreciosCombustible()
+ bool encontarEstacion(string _codigoIdentificador, int &indice)
+ void modificarEstacion(int &indice, short unsigned int &opc1, redNacional& red)
1
N
estacion
- string nombre
- string identificador
- string gerente
- string region
- string ubicacionGPS
- short int cantidadSurtidores
- surtidor* Surtidores
- float tanque[6] = {0,0,0,0,0,0}
+ void agregarSurtidor(surtidor& nuevoSurtidor)
+ void capacidadTanque()
+ void eliminarSurtidor(string _codigoIdentificador)
+ void consultarTransacciones()
+ void modificarSurtidor(string _codigoIdentificador)
+ void cantidadesVendidas()
+ void mostrarInformacion()const
+ void verificarFugas()
+ void simularVenta(redNacional& red)
1
N
surtidor
- string codigoIdentificador
- string modelo
-bool activo
- int numVentas
- int capacidad
- Venta* ventas
+ string getCodigoIdentificador()
+ Venta* getVentas()
+ bool getActivo()
+ int getCapacidad()
+ int getNumVentas()
+ void redimensionar()
+ void mostrarVentas()
+ void setcambiarEstado()
+ void agregarVenta(const Venta& nuevaVenta)
ventas
- string fechaVenta
- string horaVenta
- short unsigned int cantidadCombustibleVendido
- string categoriaCombustible
- string metodoPago
- string documentoCliente
- unsigned int cantidadDinero
+short unsigned int getCantidadCombustibleVendido() const
+ string getCategoriaCombustible() const
+ unsigned int getCantidadDinero() const
+ void setCantidadCombustibleVendido(float _cantidadCombustibleVendido)
+ void setCantidadDinero(unsigned int _cantidadDinero)
+ void setFechaVenta(string _fechaVenta)
+ void setHoraVenta(string _horaVenta)
+ void setCategoriaCombustible(string _categoriaCombustible)
+ void setDocumentoCliente(string _documentoCliente)
+ void setMetodoPago(string _metodoPago)
+ void mostrarVenta()
1
N

Algoritmos implementados.

En este apartado mostraremos los distintos métodos y funciones fundamentales de la implementación de la solución planteada. Estos algoritmos están organizados por clases

- ☐ **Clase redNacional.**

```
void redNacional::agregarEstacion(estacion nuevaEstacion)
{
    bool Duplicada = false;
    for(int i = 0;i < cantidadEstaciones;i++){
        if(Estaciones[i].getUbicacionGPS() == nuevaEstacion.getUbicacionGPS()){
            cout << endl;
            cout << endl;
            cout << endl;
            cout << endl;
            cout<< "Ya encuentra agregada una estacion en esa ubicacion GPS"<< endl;
            cout << endl;
            cout << endl;
            cout << endl;
            cout << endl;
            Duplicada = true;
            break;
        }
        else if(Estaciones[i].getIdentificador() == nuevaEstacion.getIdentificador()){
            cout << endl;
            cout << endl;
            cout << endl;
            cout << endl;
            cout<< "Ya encuentra agregada una estacion con ese identificador"<< endl;
            cout << endl;
            cout << endl;
            cout << endl;
            cout << endl;
            Duplicada = true;
            break;
        }

    }
    if(!Duplicada){
        if (cantidadEstaciones == capacidadEstaciones) {
            redimensionar(Estaciones,capacidadEstaciones,cantidadEstaciones);
        }
        Estaciones[cantidadEstaciones] = nuevaEstacion;
        cantidadEstaciones++;
        cout<< "La estacion fue agregada correctamente."<< endl;

    }
}
```

- El método agregarEstacion pertenece a la clase redNacional y tiene como objetivo agregar una nueva estación, representada por el objeto nuevaEstacion, a la red de

estaciones. Antes de realizar la adición, el método verifica si ya existe una estación con la misma ubicación GPS o identificador dentro de la red. En caso de encontrar duplicados, se notifica al usuario y no se realiza la adición. Si no se detectan duplicados y la capacidad actual de la red es suficiente, se agrega la nueva estación a la lista. En caso de que la capacidad de la red sea insuficiente, se amplía dinámicamente antes de agregar la nueva estación.

```
void redNacional::eliminarEstacion(int &indice){
    bool estadoSurtidor=false;
    short unsigned int cantidadSurtidores = 0;

    cantidadSurtidores = Estaciones[indice].getCantidadSurtidores();

    for(int k = 0;k<cantidadSurtidores;k++){
        estadoSurtidor = Estaciones[indice].getSurtidores()[k].getActivo();
        if(estadoSurtidor == true){
            cout<<"La estacion tiene surtidores activos"<<endl;
                k = cantidadSurtidores;
        }
    }

    if(estadoSurtidor == false){
        for (int j = indice; j < cantidadEstaciones - 1; j++) {
        Estaciones[j] = Estaciones[j + 1];  // Desplaza todas las estaciones una posición a la izquierda
        }
            cantidadEstaciones--;
            cout << "La estacion ha sido eliminada correctamente." << endl;
    }

}
```

- El método eliminarEstacion de la clase redNacional busca eliminar una estación específica de la red verificando primero si tiene surtidores activos. El método recibe la posición en el arreglo de Estaciones de la estación que se quiere eliminar, obtiene la cantidad de surtidores de esa estación. Luego, verifica mediante otro bucle si alguno de los surtidores está activo, utilizando el método getActivo(). Si algún surtidor está activo, el proceso de eliminación se detiene y se muestra un mensaje indicando que no puede eliminarse la estación. Si todos los surtidores están inactivos, el método desplaza las estaciones posteriores una posición hacia la izquierda para ocupar el espacio de la estación eliminada, decrementa la cantidad

total de estaciones, y muestra un mensaje confirmando que la eliminación fue exitosa. Si no se encuentra la estación en todo el proceso, se informa al usuario que la estación no está registrada.

```
void redNacional::mostrarEstaciones() const {

    cout<<"------------------------------" << endl;
    cout << "Numero de estaciones a nivel Nacional: "<< cantidadEstaciones<< endl;
    for (int i = 0; i < cantidadEstaciones; i++) {
        Estaciones[i].mostrarInformacion();
    }
    cout<<"------------------------------" << endl;
}
```

- El método mostrarEstaciones de la clase redNacional tiene como objetivo mostrar la información de todas las estaciones almacenadas en la red. Este método proporciona una vista general del número total de estaciones y detalla los datos específicos de cada estación, presentados en un formato estructurado para facilitar su lectura. El método es const, lo que asegura que no modificará el estado de los atributos de la clase.

```
void redNacional::montoVentasNacional(){
    unsigned long int montoRegular(0),montoPremium(0),montoEcoExtra(0),monto(0);
    string tipoCombustible;
    short int cantidadSurtidores(0);
    int numVentas(0);

    for(int i = 0; i<cantidadEstaciones;i++){
        cantidadSurtidores = Estaciones[i].getCantidadSurtidores();
        for(int k=0;k<cantidadSurtidores;k++){
            numVentas = Estaciones[i].getSurtidores()[k].getNumVentas();
            for(int w = 0;w<numVentas;w++){
                tipoCombustible = Estaciones[i].getSurtidores()[k].getVentas()[w].getCategoriaCombustible();
                monto =  Estaciones[i].getSurtidores()[k].getVentas()[w].getCantidadDinero();

                if(tipoCombustible=="Regular"){
                    montoRegular += monto;
                }

                else if(tipoCombustible=="Premium"){
                    montoPremium += monto;
                }

                else{
                    montoEcoExtra += monto;
                }

            }
        }

    }

    cout<<"Monto total de ventas de combustible Regular: "<<montoRegular<<endl;
    cout<<"Monto total de ventas de combustible Premium: "<<montoPremium<<endl;
    cout<<"Monto total de ventas de combustible EcoExtra: "<<montoEcoExtra<<endl;
    cout<<endl;


}
```

- El método montoVentasNacional de la clase redNacional permite calcular y mostrar el monto total de las ventas de combustible a nivel nacional. El método realiza el siguiente procedimiento: El método utiliza un bucle for para recorrer todas las estaciones almacenadas en el arreglo Estaciones. La variable de control del bucle (i) recorre desde 0 hasta cantidadEstaciones, que es el número total de estaciones registradas en la red nacional. Para cada estación, el método obtiene la cantidad de surtidores con el método getCantidadSurtidores() y la almacena en la variable cantidadSurtidores. Luego, el método entra en un segundo bucle for anidado que recorre cada surtidor de la estación actual. La variable de control de este segundo bucle (k) recorre desde 0 hasta cantidadSurtidores. Para cada surtidor, el método obtiene el número de ventas realizadas utilizando el método getNumVentas(), que

devuelve el número total de transacciones de ese surtidor. Este número se guarda en la variable numVentas. A continuación, el método entra en un tercer bucle for, donde la variable de control (w) recorre desde 0 hasta numVentas. En cada iteración, el método obtiene el tipo de combustible de la venta actual con getCategoriaCombustible() y la cantidad de dinero de la transacción con getCantidadDinero(). El tipo de combustible se guarda en la variable tipoCombustible, y el monto de la venta en la variable monto. Dependiendo del valor de tipoCombustible, el monto se agrega al total correspondiente: Si tipoCombustible es "Regular", el monto se suma a la variable montoRegular. Si tipoCombustible es "Premium", el monto se suma a la variable montoPremium.

Para el otro tipo de combustible, el monto se suma a la variable montoEcoExtra. Una vez recorridas todas las estaciones, surtidores y ventas, el método muestra los montos totales de cada tipo de combustible. Imprime el monto total de ventas de Regular, Premium y EcoExtra utilizando cout y finaliza la ejecución.

```
void redNacional::fijarPreciosCombustible(){
    unsigned int precio = 0;

    cout<<"Ingrese el precio de combustible Regular en la Region Norte: ";
    cin>>precio;
    preciosCombustible[0] = precio;

    cout<<"Ingrese el precio de combustible Premium en la Region Norte: ";
    cin>>precio;
    preciosCombustible[1] = precio;

    cout<<"Ingrese el precio de combustible EcoExtra en la Region Norte: ";
    cin>>precio;
    preciosCombustible[2] = precio;

    cout<<"Ingrese el precio de combustible Regular en la Region Centro: ";
    cin>>precio;
    preciosCombustible[3] = precio;

    cout<<"Ingrese el precio de combustible Premium en la Region Centro: ";
    cin>>precio;
    preciosCombustible[4] = precio;

    cout<<"Ingrese el precio de combustible EcoExtra en la Region Centro: ";
    cin>>precio;
    preciosCombustible[5] = precio;

    cout<<"Ingrese el precio de combustible Regular en la Region Sur: ";
    cin>>precio;
    preciosCombustible[6] = precio;

    cout<<"Ingrese el precio de combustible Premium en la Region Sur: ";
    cin>>precio;
    preciosCombustible[7] = precio;

    cout<<"Ingrese el precio de combustible EcoExtra en la Region Sur: ";
    cin>>precio;
    preciosCombustible[8] = precio;
}
```

- El método fijarPreciosCombustible de la clase redNacional permite al usuario ingresar manualmente los precios de tres tipos de combustibles (Regular, Premium y EcoExtra) para tres diferentes regiones (Norte, Centro y Sur) y almacenarlos en un arreglo llamado preciosCombustible. El método sigue un proceso secuencial en el que, utilizando la función cin, solicita al usuario que ingrese el precio de cada tipo de combustible para cada región en el siguiente orden: primero, los precios para la región Norte (posiciones 0 a 2 del arreglo), luego los precios para la región

Centro (posiciones 3 a 5 del arreglo), y finalmente los precios para la región Sur (posiciones 6 a 8 del arreglo). Cada precio ingresado por el usuario se almacena en la posición correspondiente del arreglo preciosCombustible, con tres posiciones dedicadas a cada región y cada posición representando un tipo de combustible.

```
void redNacional::redimensionar(estacion*& arreglo, unsigned short int& capacidadEstaciones,unsigned short int estaciones)
{

    int nuevaCapacidad = capacidadEstaciones + 50;
    estacion* nuevoArreglo = new estacion[nuevaCapacidad];
    for(int i= 0;i < estaciones;i++)
    {
        nuevoArreglo[i] = arreglo[i];
    }
    delete[] arreglo;
    arreglo = nuevoArreglo;
    capacidadEstaciones = nuevaCapacidad;
}
```

- El método redimensionar de la clase redNacional tiene como finalidad aumentar la capacidad de almacenamiento del arreglo dinámico que contiene las estaciones de la red. Este método sigue el siguiente procedimiento: primero, calcula una nueva capacidad aumentando la actual (capacidadEstaciones) en 50 unidades, y luego crea un nuevo arreglo dinámico de tipo estacion con esta nueva capacidad. Posteriormente, utilizando un bucle for, copia todas las estaciones existentes desde el arreglo original al nuevo arreglo. Después de que todas las estaciones han sido copiadas, se libera la memoria ocupada por el arreglo original utilizando delete[], y se reasigna el puntero arreglo para que apunte al nuevo arreglo creado. Finalmente, se actualiza el valor de capacidadEstaciones para reflejar la nueva capacidad. Este proceso asegura que la red pueda seguir agregando estaciones sin problemas de espacio en el arreglo.

```
bool redNacional::encontarEstacion(string _codigoIdentificador, int &indice)
{
    for(int i = 0;i<cantidadEstaciones;i++){
        if(Estaciones[i].getIdentificador() == _codigoIdentificador){
            indice = i;
            return true;
        }
    }

    cout<<"La estacion ingresada no se encuentra registrada"<<endl;
    return false;
}
```

El método encontarEstacion de la clase redNacional busca una estación registrada en el sistema mediante su código identificador. Recorre el arreglo de estaciones (Estaciones) utilizando un bucle for, comparando el código identificador de cada estación con el valor de _codigoIdentificador proporcionado.Si encuentra una coincidencia, asigna el valor de la posición (índice) de la estación encontrada a la variable indice y devuelve true, indicando que la estación fue encontrada. Si no encuentra ninguna estación con el código proporcionado, imprime un mensaje indicando que la estación no está registrada y retorna false.

☐ **Clase estacion.**

```
void estacion::agregarSurtidor(surtidor& nuevoSurtidor){
    bool Duplicada = false;

    for(int i = 0;i < cantidadSurtidores;i++){
        if(Surtidores[i].getCodigoIdentificador() == nuevoSurtidor.getCodigoIdentificador()){
            cout << endl;
            cout << endl;
            cout<< "Ya se encuentra agregado un surtidor con ese codigo identificador"<<endl;
            cout << endl;
            cout << endl;
            Duplicada = true;
            break;
        }
    }

    if(cantidadSurtidores==12){
        cout<<"No se pueden agregar mas surtidores a la estacion "<<nombre<<endl;
    }

    else if(!Duplicada){
        Surtidores[cantidadSurtidores]= nuevoSurtidor;
        cantidadSurtidores++;
        cout<< "El SURTIDOR se ha agregado correctamente."<< endl;
        cout<<"La cantidad de surtidores es: "<< cantidadSurtidores<< endl;
    }
}
```

- El método agregarSurtidor de la clase estacion permite añadir un nuevo surtidor a la estación, siempre y cuando no exista un surtidor en la estacion con el mismo código identificador y la cantidad máxima de surtidores no haya sido alcanzada. El método sigue este proceso: primero, mediante un bucle for se comparan todos los códigos identificadores de los surtidores actuales con el nuevo surtidor, si encuentra alguno igual imprime un mensaje indicando que el código usado no está disponible y no permite agregar ese surtidor, después verifica si el número actual de surtidores en la estación (cantidadSurtidores) ha alcanzado el límite, que en este caso es 12. Si se ha alcanzado dicho límite, se imprime un mensaje indicando que no es posible agregar más surtidores a la estación, mencionando su nombre (nombre). En caso contrario, el nuevo surtidor (nuevoSurtidor) es agregado en la posición

correspondiente del arreglo Surtidores, utilizando cantidadSurtidores como índice. Después de agregar el surtidor, se incrementa el contador cantidadSurtidores para reflejar la adición y se muestra un mensaje confirmando que el surtidor ha sido agregado correctamente. Este método asegura que no se exceda la capacidad máxima de surtidores en la estación.

```
void estacion::eliminarSurtidor(string _codigoIdentificador){

    string id;
    bool encontrado = false;

    if(cantidadSurtidores>2){
        for (int i = 0; i < cantidadSurtidores; i++) {
            id = Surtidores[i].getCodigoIdentificador();
            if (id == _codigoIdentificador) {
                encontrado = true;
                // Mover los surtidores hacia atrás en el arreglo
                for (int k = i; k < cantidadSurtidores - 1; k++) {
                    Surtidores[k] = Surtidores[k + 1];
                }
                cantidadSurtidores--; // Reducir la cantidad de surtidores
                break; // Salir del bucle una vez encontrado
            }
        }

        if (encontrado) {
            cout << "Surtidor eliminado" << endl;
            cout << endl;
        } else {
            cout << "El codigo ingresado no pertenece a un surtidor de la estacion" << endl;
            cout << endl;
        }
    }
    else{
       cout <<"La estacion tiene el numero minimo de surtidores: 2"<<endl;
    }
}
```

- El método eliminarSurtidor de la clase estacion busca eliminar un surtidor específico de la estación, identificándolo a través de su código identificador. El proceso que sigue este método es el siguiente: primero, se declara una variable (id) que almacenará el código identificador del surtidor en cada iteración y una variable booleana (encontrado) que indica si el surtidor ha sido localizado.Si la cantidad de

surtidores en la estación no es mayor a 2 no permitirá eliminar o verificar si el código del surtidor ingresado. El método recorre todos los surtidores de la estación utilizando un bucle for. En cada iteración, obtiene el código identificador del surtidor actual llamando al método getCodigoIdentificador() y lo compara con el código proporcionado como argumento (_codigoIdentificador). Si encuentra una coincidencia, se establece encontrado en true y se procede a eliminar el surtidor moviendo todos los surtidores posteriores una posición hacia atrás en el arreglo. Para ello, utiliza un segundo bucle for que desplaza los elementos del arreglo Surtidores desde la posición del surtidor eliminado hasta el final. Luego, se decrementa el valor de cantidadSurtidores para reflejar la eliminación. Si se encuentra y elimina el surtidor, se imprime un mensaje confirmando que ha sido eliminado exitosamente. En caso de que no se encuentre ningún surtidor con el código dado, se muestra un mensaje indicando que el código ingresado no corresponde a un surtidor registrado en la estación. Este método asegura que el arreglo de surtidores permanezca correctamente ordenado y sin vacíos tras la eliminación de un surtidor.

```
void estacion::modificarSurtidor(string _codigoIdentificador){
    string id;
    bool encontrado = false,estado= false;

    for(int i=0;i<cantidadSurtidores;i++){
        id = Surtidores[i].getCodigoIdentificador();
        estado = Surtidores[i].getActivo();
        if(id ==_codigoIdentificador){
            encontrado = true;
            if(estado==true){
                Surtidores[i].setcambiarEstado();
                i = cantidadSurtidores;
                cout<<"Surtidor desactivado"<<endl;
                cout<<endl;
            }
            else{
                Surtidores[i].setcambiarEstado();
                i = cantidadSurtidores;
                cout<<"Surtidor activado"<<endl;
                cout<<endl;
            }
        }

    }

    if(!encontrado){
        cout<<"El codigo ingresado no corresponde a un surtidor de la estacion"<<endl;
        cout<<endl;
    }
}
```

- El método modificarSurtidor de la clase estacion permite modificar el estado de un surtidor en la estación identificándolo por su código identificador. El proceso que sigue este método es el siguiente: primero, se declara una variable (id) que almacenará el código identificador de cada surtidor a medida que el bucle for recorre el arreglo de surtidores de la estación. Además, se utiliza una variable booleana (encontrado) para indicar si el surtidor ha sido localizado. El bucle recorre todos los surtidores de la estación, y en cada iteración se obtiene el código identificador del surtidor actual llamando al método getCodigoIdentificador(). Este código se compara con el código proporcionado como parámetro (_codigoIdentificador). Si se encuentra una coincidencia, se establece encontrado

en true y se llama al método setcambiarEstado(), que desactiva el surtidor (probablemente cambia su estado de activo a inactivo). El bucle se detiene asignando el valor de cantidadSurtidores a la variable de control i para evitar más iteraciones innecesarias. Después de la desactivación, se imprime un mensaje confirmando que el surtidor ha sido desactivado. Si al finalizar el bucle no se ha encontrado ningún surtidor con el código dado (es decir, si encontrado permanece en false), se imprime un mensaje indicando que el código ingresado no corresponde a ningún surtidor registrado en la estación. Este método asegura que solo se modifique el surtidor que coincide con el código identificador proporcionado.

```
void estacion::cantidadesVendidas(){
    float regular=0,premium=0,extra=0,vendida=0,numVentas;
    string tipo;

    for(int i=0;i<cantidadSurtidores;i++){
        numVentas = (Surtidores[i].getNumVentas());
        for(int k = 0;k<numVentas;k++){
            vendida = (Surtidores[i].getVentas())[k].getCantidadCombustibleVendido();
            tipo = (Surtidores[i].getVentas())[k].getCategoriaCombustible();
            if(tipo=="Regular"){
                regular += vendida;
            }
            else if(tipo=="Premium"){
                premium += vendida;
            }
            else{
                extra += vendida;
            }
        }
    }

    cout<<"Cantidad de combustible Regular vendido: "<<regular<<endl;
    cout<<"Cantidad de combustible Premium vendido: "<<premium<<endl;
    cout<<"Cantidad de combustible EcoExtra vendido: "<<extra<<endl;
    cout<<endl;
}
```

- El método cantidadesVendidas de la clase estacion se encarga de calcular y mostrar la cantidad total de combustible vendido, clasificado por tipo de combustible (Regular, Premium y EcoExtra), en todos los surtidores de la estación. El procedimiento es el siguiente: primero, se inicializan variables para almacenar las

cantidades vendidas de cada tipo de combustible (regular, premium, extra), así como una variable (vendida) que representa la cantidad vendida en cada transacción individual, y numVentas para almacenar el número de ventas de cada surtidor. El método comienza recorriendo todos los surtidores de la estación mediante un bucle for, utilizando la variable de control i. Para cada surtidor, se obtiene el número de ventas realizadas con getNumVentas(). Luego, un segundo bucle for (controlado por k) itera sobre cada venta registrada en ese surtidor. Dentro de este bucle, se obtiene la cantidad de combustible vendido en la venta actual con getCantidadCombustibleVendido() y la categoría del combustible (Regular, Premium o EcoExtra) con getCategoriaCombustible(). Dependiendo del tipo de combustible, se suma la cantidad vendida a la variable correspondiente (regular, premium o extra). Finalmente, después de recorrer todas las ventas de todos los surtidores, se imprimen los totales acumulados para cada tipo de combustible. De esta forma, el método proporciona una visión detallada de la cantidad total de combustible vendida en la estación, desglosada por categoría.

```
void estacion::consultarTransacciones(){

    for(int i=0;i<cantidadSurtidores;i++){
        cout<<"Ventas del surtidor con codigo: "<<Surtidores[i].getCodigoIdentificador()<<endl;
        Surtidores[i].mostrarVentas();
    }
}
```

- El método consultarTransacciones de la clase estacion permite consultar y mostrar todas las transacciones realizadas por cada surtidor de la estación. El método sigue un enfoque simple pero efectivo: utiliza un bucle for para recorrer todos los surtidores de la estación, con la variable de control " i " recorriendo desde 0 hasta

cantidadSurtidores, que representa el número total de surtidores en la estación. En cada iteración del bucle, primero se imprime el mensaje "Ventas del surtidor con código:" seguido del código identificador del surtidor actual, obtenido mediante el método getCodigoIdentificador(). Esto proporciona una referencia clara de a qué surtidor corresponden las transacciones mostradas a continuación. Una vez mostrado el código del surtidor, el método llama a mostrarVentas() del surtidor correspondiente, lo que desencadena la visualización detallada de todas las ventas registradas para ese surtidor en particular. De esta manera, el método permite visualizar de forma completa y ordenada las transacciones realizadas por cada uno de los surtidores de la estación.

```
void estacion::capacidadTanque(){
    srand(time(NULL));
    for(int i = 0;i<3;i++){
        tanque[i]= rand() % 101 + 100;
        tanque[3+i]=tanque[i];
    }
}
```

- El método capacidadTanque de la clase estacion asigna capacidades aleatorias a los tanques de combustible de la estación, dividiendo las capacidades entre dos grupos. Para lograr esto, primero se inicializa el generador de números aleatorios usando la función srand(time(NULL)). Esto asegura que los números generados en cada ejecución del programa sean diferentes, ya que se basan en la hora actual. Una vez inicializado el generador de números aleatorios, el método entra en un bucle for que itera tres veces. En cada iteración, se asigna un número aleatorio entre 100 y

200 a los primeros tres tanques de la estación. Esto se realiza utilizando la función rand() y aplicando una fórmula que limita los valores aleatorios generados al rango deseado. Después de asignar valores a los primeros tres tanques, el método duplica estas capacidades en los tanques restantes. Es decir, los tres últimos elementos del arreglo de tanques reciben los mismos valores que los primeros tres. Esto garantiza que los últimos tres tanques tengan las mismas capacidades que los primeros, generando simetría en las capacidades de los tanques de almacenamiento.

☐ **Clase surtidor.**

```
void surtidor::mostrarVentas(){
    if(numVentas==0){
        cout<<"No hay ventas registradas"<<endl;
        cout<<endl;
    }

    else{
        for(int i = 0;i<numVentas;i++){
            cout<<"VENTA "<<i+1<<endl;
            ventas[i].mostrarVenta();
            cout<<endl;
        }
    }

}
```

- El método mostrarVentas de la clase surtidor se encarga de mostrar todas las ventas registradas para ese surtidor. En primer lugar, el método verifica si no hay ventas registradas comprobando si el valor de la variable numVentas es igual a 0. Si no se han realizado ventas, muestra un mensaje en pantalla indicando que "No hay ventas registradas" y luego genera un salto de línea para mantener el formato de salida

ordenado. Si el surtidor ha registrado ventas, el método entra en un bucle for que recorre todas las ventas almacenadas. Durante cada iteración, muestra el número de la venta (iniciando en 1 para facilitar la lectura) seguido de una llamada al método mostrarVenta de cada venta en particular. Esto asegura que se impriman los detalles de cada transacción registrada en el surtidor.

- Simular venta es un método que recibe como parámetro una referencia a un objeto de clase redNacional. Este método pide todos los datos necesarios correspondientes de una venta. El surtidor donde se almacenará la venta es elegido al azar, al igual que la cantidad de gasolina vendida. El costo de la transacción se determinará según el tipo de combustible y la cantidad de combustible vendida, este costo se determina de forma automática. Al final si todos los datos fueron ingresados con éxito se almacenará un nuevo objeto clase venta en el surtidor correspondiente. Para poder simular una venta, la estacion debe tener al menos un surtidor activo.

```
void surtidor::agregarVenta(const Venta& nuevaVenta) {
    if (numVentas == capacidad) {
        redimensionar();
    }
    ventas[numVentas] = nuevaVenta;
    numVentas++;
}
```

- El método agregarVenta de la clase surtidor se encarga de añadir una nueva venta al registro de ventas del surtidor, asegurándose de que haya espacio suficiente en el arreglo donde se almacenan estas ventas. El método toma como parámetro una referencia constante a un objeto de la clase Venta, que representa la nueva venta

que se va a agregar. En primer lugar, el método verifica si la cantidad actual de ventas (numVentas) ha alcanzado la capacidad máxima del arreglo de ventas (capacidad). Si es así, llama al método redimensionar() para aumentar el tamaño del arreglo y asegurarse de que haya espacio para la nueva venta. El método redimensionar() probablemente se encargue de crear un arreglo más grande y transferir las ventas existentes al nuevo arreglo. Si hay espacio suficiente o después de redimensionar el arreglo, el método asigna la nueva venta a la posición correspondiente en el arreglo ventas (es decir, la posición indicada por numVentas). Luego, incrementa la variable numVentas, que lleva el conteo de cuántas ventas se han registrado.

- ☐ **Clase venta.**

```
void Venta::mostrarVenta(){
    cout<<"Fecha: "<<fechaVenta<<endl;
    cout<<"Hora: "<<horaVenta<<endl;
    cout<<"Cantidad de combustible vendido: "<<cantidadCombustibleVendido<<endl;
    cout<<"Categoria de combustible vendido: "<<categoriaCombustible<<endl;
    cout<<"Metodo de pago: "<<metodoPago<<endl;
    cout<<"Documento del cliente: "<<documentoCliente<<endl;
    cout<<"Valor de la venta: "<<cantidadDinero<<endl;
}
```

- El método mostrarVenta de la clase Venta se encarga de imprimir en pantalla todos los detalles de una transacción de venta de combustible. Comienza mostrando la fecha de la venta, almacenada en el atributo fechaVenta, seguida de la hora en que se realizó la transacción, que se encuentra en horaVenta. A continuación, imprime la cantidad de combustible vendido, representada por el atributo cantidadCombustibleVendido, y la categoría del combustible (como "Regular", "Premium" o "EcoExtra"), que se encuentra en categoriaCombustible. El método

continúa mostrando el método de pago utilizado para la compra, almacenado en metodoPago, y el documento del cliente, que se encuentra en documentoCliente, para finalizar mostrando el valor total de la venta, almacenado en cantidadDinero. Este método proporciona de manera clara y estructurada toda la información relevante sobre una venta realizada en el surtidor.

Problemas en el desarrollo del Desafío.

Durante el desarrollo del desafío surgieron inconvenientes en el momento de la utilización de la memoria dinámica, ya que al momento de liberar la memoria dinámica con el destructor en las clases creadas que utilizan memoria dinámica el sistema operativo lanzaba un error. Este problema de la liberación de memoria se puede deber a la poca experiencia que tenemos en el paradigma de la Programación Orientada a Objetos.

Al final del desarrollo tuvimos problemas al agregar, eliminar y modificar surtidores dado que los surtidores que se agregaban o eliminaban se estaban almacenando en una copia del objeto de la clase estacion y no en el objeto original, por lo tanto cuando el programa buscaba el surtidor en la estacion, no lo encontraba, dado que no se había agregado al objeto original.

Evolución del desarrollo de la solución del Desafío.

La evolución de la solución comenzó con un análisis del enunciado del problema, a partir del cual realizamos un borrador del diagrama de clases, que incluía cinco clases: redNacional, estacion , surtidor, venta y tanque. A medida que avanzamos en el desarrollo, decidimos que no era necesario tratar a tanque como una clase independiente, sino como un atributo de la clase estacion. Inicialmente, modelamos tanque como una matriz de dos filas y tres columnas, pero luego optamos por un arreglo de seis posiciones: las tres

primeras representan la capacidad máxima de los combustibles (Regular, Premium, EcoExtra), y las tres siguientes, la cantidad disponible de cada combustible en el mismo orden. Esto refleja que nuestra solución no fue estática y fue ajustada conforme avanzábamos en el desarrollo.

Al comenzar a programar, iniciamos creando la clase venta, implementando sus primeros atributos y métodos, como los getters y setters. Luego, continuamos con la clase surtidor, asegurándonos de que las funciones que relacionaban a surtidor y venta funcionaran correctamente. Posteriormente, trabajamos en las clases estacion y redNacional. A medida que implementamos las funcionalidades requeridas, ajustamos ciertos métodos y atributos de las clases anteriores según fuera necesario. Una vez completada la mayoría de los requerimientos, diseñamos una serie de menús que permiten al usuario navegar fácilmente por las acciones del programa. Por ejemplo, el menú principal ofrece opciones para interactuar con la red nacional de estaciones, gestionar una estación específica o verificar fugas en una estación. Según la selección, se despliegan menús adicionales con las funcionalidades específicas, siendo este un menú jerárquico..

Durante el desarrollo de este desafío, descubrimos nuevas formas de resolver problemas mediante programación, lo que nos permitió adentrarnos en el paradigma de la programación orientada a objetos de manera más profunda.