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.

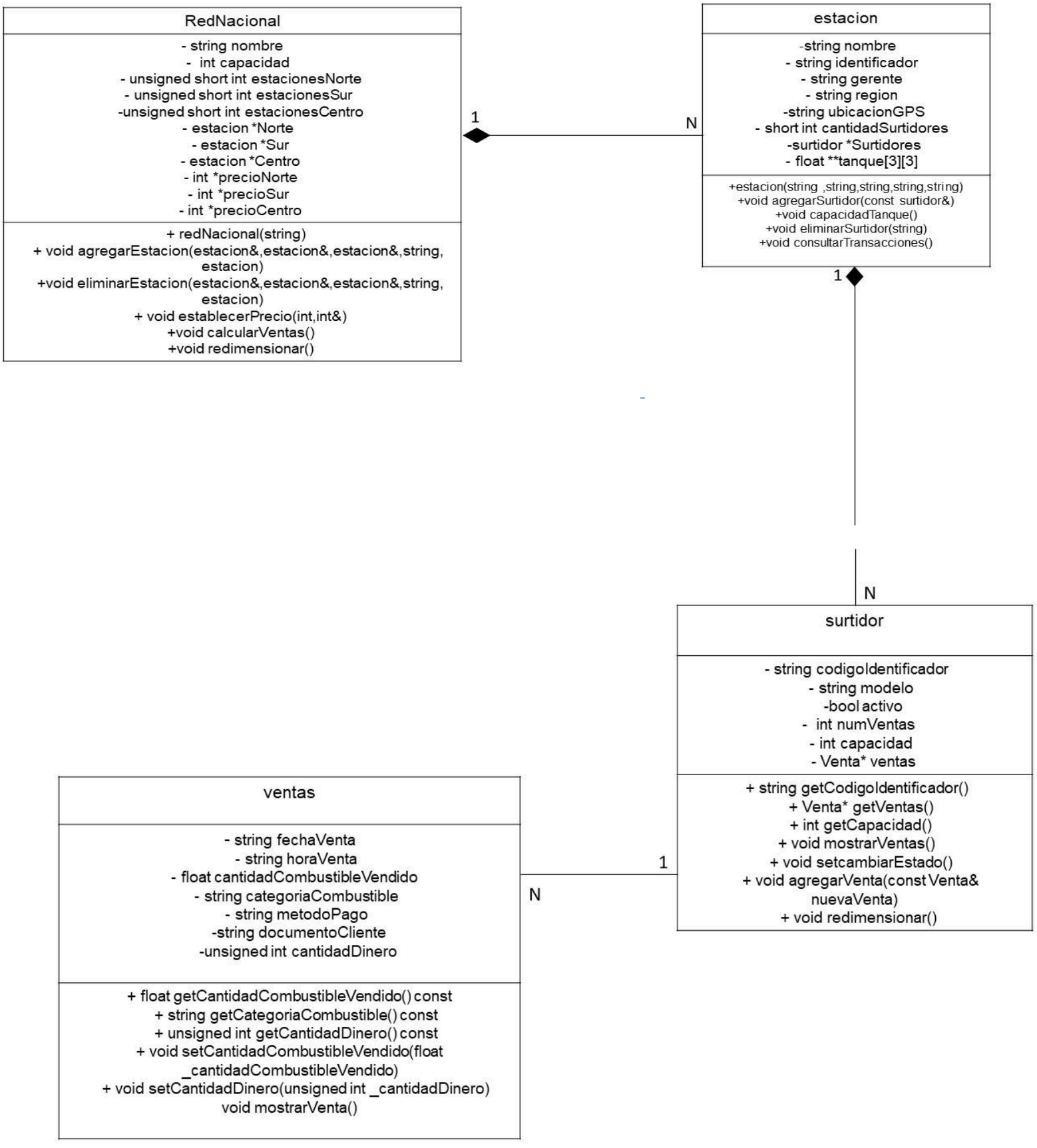
TerMax, una compañía líder en el suministro de combustibles en Colombia, necesita un sistema de gestión para su red de estaciones de servicio. Cada estación tiene un nombre, código identificador, gerente, región y ubicación GPS, y almacena tres tipos de combustible (Regular, Premium y EcoExtra) en un 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. 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 inicial que utilizamos para la planificación y desarrollo de una solución inicial del problema


RedNacional
- string nombre
- int capacidad
- unsigned short int estacionesNorte
- unsigned short int estacionesSur
-unsigned short int estacionesCentro
- estacion *Norte
- estacion *Sur
- estacion *Centro
- int *precioNorte
- int *precioSur
- int *precioCentro
+ redNacional(string)
+ void agregarEstacion(estacion&,estacion&,estacion&,string, estacion)
+void eliminarEstacion(estacion&,estacion&,estacion&,string, estacion)
+ void establecerPrecio(int,int&)
+void calcularVentas()
+void redimensionar()
1
N
estacion
-string nombre
- string identificador
- string gerente
- string region
-string ubicacionGPS
- short int cantidadSurtidores
-surtidor *Surtidores
- float **tanque[3][3]
+estacion(string ,string,string,string,string)
+void agregarSurtidor(const surtidor&)
+void capacidadTanque()
+void eliminarSurtidor(string)
+void consultarTransacciones()
1
N
surtidor
- string codigoIdentificador
- string modelo
-bool activo
- int numVentas
- int capacidad
- Venta* ventas
+ string getCodigoIdentificador()
+ Venta* getVentas()
+ int getCapacidad()
+ void mostrarVentas()
+ void setcambiarEstado()
+ void agregarVenta(const Venta& nuevaVenta)
+ void redimensionar()
1
N
ventas
- string fechaVenta
- string horaVenta
- float cantidadCombustibleVendido
- string categoriaCombustible
- string metodoPago
-string documentoCliente
-unsigned int cantidadDinero
+ float getCantidadCombustibleVendido() const
+ string getCategoriaCombustible() const
+ unsigned int getCantidadDinero() const
+ void setCantidadCombustibleVendido(float _cantidadCombustibleVendido)
+ void setCantidadDinero(unsigned int _cantidadDinero)
void mostrarVenta()

Algoritmos implementados.

En este apartado mostraremos los distintos métodos y funciones fundamentales de la implementación de la solución planteada inicialmente.

```
void surtidor::redimensionar() {

    int nuevaCapacidad = capacidad + 10;
    Venta* nuevoArreglo = new Venta[nuevaCapacidad];

    for (int i = 0; i<numVentas; i++) {
        nuevoArreglo[i] = ventas[i];
    }

    delete[] ventas;

    ventas = nuevoArreglo;
    capacidad = nuevaCapacidad;
}
```

- Este método de la clase surtido llamado redimensionar de la clase surtidor incrementa la capacidad de almacenamiento para las ventas del surtidor. Este método lo que hace es: Calcular la nueva capacidad sumando 10 a la capacidad actual, crea un nuevo arreglo de ventas (nuevoArreglo) con la nueva capacidad usando memoria dinámica, copia todas las ventas del arreglo original (ventas) al nuevo arreglo, libera la memoria ocupada por el arreglo original, asigna el nuevo arreglo (nuevoArreglo) al arreglo original (ventas) y por ultimo actualiza la capacidad del surtidor con la nueva capacidad calculada.

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

- Este método de la clase surtidor llamado agregarVenta se encarga de agregar una nueva venta al surtidor, verificando si hay espacio suficiente. Su funcionalidad se basa en: Comprobar el espacio disponible, si el número actual de ventas (numVentas) ha alcanzado la capacidad máxima del arreglo (capacidad), llama al método redimensionar() para aumentar el espacio disponible. Posteriormente agrega la nueva venta de la siguiente manera: Una vez que se asegura que hay espacio, guarda la nueva venta (nuevaVenta) en el arreglo de ventas en la posición actual (índice numVentas), incrementa el número de ventas (numVentas) en uno, para reflejar que se ha agregado una venta más.

```
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 tiene como objetivo mostrar las ventas registradas en el surtidor, dependiendo de si existen o no ventas. La función realiza los siguientes pasos: En primer lugar, el método verifica si el número de ventas (numVentas) es igual a 0. Si este es el caso, imprime el mensaje "No hay ventas registradas" para indicar que no existen ventas almacenadas en el surtidor. Después, imprime una línea en blanco y el proceso finaliza. Si hay ventas registradas (es decir, numVentas es mayor que 0), el método entra en un bucle for que recorre todas las ventas almacenadas en el arreglo ventas. Para cada venta, se imprime el número correspondiente a esa venta (empezando desde 1) y se llama al método mostrarVenta() de la clase Venta, que es el encargado de mostrar los detalles específicos de cada venta. Luego de cada venta, se imprime una línea vacía para proporcionar una separación visual entre cada registro.

```
void estacion::agregarSurtidor(const surtidor& nuevoSurtidor){

    if(cantidadSurtidores==11){
        cout<<"No se pueden agregar mas surtidores a la estacion "<<nombre<<endl;
    }
    else{
        Surtidores[cantidadSurtidores]= nuevoSurtidor;
        cantidadSurtidores++;
    }
}
```

- El método agregarSurtidor de la clase estación tiene como finalidad añadir un nuevo surtidor a la estación. El procedimiento que sigue este método es el siguiente: Antes de agregar un nuevo surtidor, el método verifica si el número actual de surtidores (cantidadSurtidores) ha alcanzado el límite máximo permitido, que es 12. Si ya hay 12 surtidores, se imprime un mensaje en la consola indicando que no es posible agregar más surtidores a la estación, seguido del nombre de la

estación. Si el número de surtidores es menor a 12, el método procede a agregar el nuevo surtidor al arreglo Surtidores, almacenándolo en la posición correspondiente de acuerdo con el valor de cantidadSurtidores. Posteriormente, incrementa en 1 el contador de surtidores (cantidadSurtidores) para reflejar que se ha añadido un nuevo surtidor.

```
void estacion::eliminarSurtidor(string _codigoIdentificador) {
    if (cantidadSurtidores == 0) {
        cout << "No hay surtidores para eliminar." << endl;
        return;
    }

    surtidor *newSurtidores = new surtidor[12];  // Crear nuevo arreglo dinámico

    bool encontrado = false;  // Para saber si el surtidor fue encontrado

    for (int i = 0, j = 0; i < cantidadSurtidores; i++) {
        if (Surtidores[i].getCodigoIdentificador() == _codigoIdentificador) {
            // No copiar el surtidor que queremos eliminar
            encontrado = true;
        } else {
            newSurtidores[j] = Surtidores[i];  // Copiar los demás surtidores
            j++;
        }
    }

    if (encontrado) {
        cantidadSurtidores--;
        delete[] Surtidores;  // Liberar memoria del arreglo antiguo
        Surtidores = newSurtidores;  // Asignar el nuevo arreglo
        cout << "Surtidor eliminado." << endl;
    } else {
        delete[] newSurtidores;  // Si no se encontró, no necesitamos este nuevo arreglo
        cout << "Surtidor con identificador " << _codigoIdentificador << " no encontrado." << endl;
    }

    cout << endl;
}
```

- El método eliminarSurtidor de la clase estación permite eliminar un surtidor específico de la estación, identificado por su código. A continuación, se detallan los pasos que realiza este método: En primer lugar, el método comprueba si hay surtidores en la estación verificando si cantidadSurtidores es igual a 0. Si no hay surtidores, se muestra el mensaje "No hay surtidores para eliminar.", y el método termina inmediatamente sin realizar ninguna acción adicional. Si hay surtidores, se

crea un nuevo arreglo dinámico de surtidores (new Surtidores) con un tamaño máximo de 12, que será utilizado para almacenar los surtidores restantes luego de eliminar el surtidor solicitado.

El método recorre el arreglo actual de surtidores (Surtidores) y compara el código identificador de cada surtidor con el código que se desea eliminar (_codigoIdentificador). Si se encuentra una coincidencia, el surtidor no es copiado al nuevo arreglo, lo que efectivamente lo elimina. El booleano encontrado se utiliza para registrar si el surtidor fue encontrado o no durante la búsqueda.Si el surtidor fue encontrado y eliminado, el método decrementa la cantidad total de surtidores (cantidadSurtidores), libera la memoria del arreglo original (delete[] Surtidores), y reasigna el nuevo arreglo (new Surtidores) al atributo Surtidores. Se imprime un mensaje confirmando que el surtidor fue eliminado. Si el surtidor no fue encontrado, se libera la memoria del nuevo arreglo y se imprime un mensaje indicando que el surtidor con el código dado no fue encontrado. Después de completar la operación, el método imprime un salto de línea en blanco para dar una separación visual a la salida.

```
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 en la estación. El método hace el siguiente procedimiento: El método utiliza un bucle for para recorrer todos los surtidores almacenados en la estación. La variable de control del bucle (i) recorre desde 0 hasta cantidadSurtidores, que es el número total de surtidores registrados en la estación. Para cada surtidor, el método obtiene y muestra su código identificador llamando al método getCodigoIdentificador(). Este código se imprime junto con el mensaje "Ventas del surtidor con código:", lo que proporciona una referencia clara de a qué surtidor corresponden las transacciones mostradas a continuación. Luego de mostrar el código identificador del surtidor, el método llama al método mostrarVentas() de la clase surtidor, que se encarga de mostrar todas las ventas registradas para ese surtidor en particular. Esto permite visualizar las transacciones realizadas por cada surtidor de forma detallada.

```
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 está diseñado para mostrar los detalles de una venta específica en la consola. A continuación se explica cómo funciona este método: El método comienza mostrando la fecha de la venta (fechaVenta) y la hora en la que ocurrió (horaVenta). Estos datos se presentan en la consola bajo el formato "Fecha:" y "Hora:", respectivamente, lo que permite saber cuándo se

realizó la transacción. Posteriormente se muestra la cantidad de combustible vendido (cantidadCombustibleVendido), junto con la categoría del combustible (categoriaCombustible). Esto proporciona una descripción del producto adquirido durante la transacción. El método también muestra el método de pago utilizado en la venta (metodoPago), indicando si fue pagado en efectivo, con tarjeta de crédito u otros métodos. El documento del cliente que realizó la compra (documentoCliente) es mostrado, permitiendo identificar al cliente que realizó la transacción. Finalmente, se imprime el valor monetario de la venta (cantidadDinero), que representa el monto total pagado por el cliente durante la transacción.

NOTA: Dado que todavía estamos en el proceso de desarrollo estas funciones pueden estar sujetas a cambios, así como también se pueden implementar funciones nuevas.

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 surgen un aborto de ejecución. 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.

Por el momento hasta el dia que se realiza este informe estos han sido los únicos inconvenientes que se nos han presentado, sin embargo no dudamos que más adelante siguiendo con la implementación nos surgirán más inconvenientes que serán descritos y solucionados en el informe final.