Juan pablo rivero garay

1. Análisis del problema y consideraciones para la alternativa de solución propuesta.

Análisis

Queremos hacer una red red nacional de suministro de combustible, la cual tiene las siguientes características:

Estaciones de servicio:

* Cada estación tiene atributos únicos un nombre, código identificador, gerente, región y ubicación geográfica (coordenadas GPS).
* Capacidades de almacenamiento variable: El tanque central en cada estación almacena las tres categorías de combustible, pero la capacidad para cada tipo de combustible no es uniforme entre estaciones(varia)
* Máquinas surtidoras: Cada estación tiene de 2 a 12 surtidores conectados al tanque central. Las ventas se realizan a través de estos surtidores.

Tipos de combustible:

* Regular, Premium y EcoExtra son los tipos de gasolina en cada estación.
* La cantidad disponible de cada tipo de combustible varía entre las estaciones, y dicha cantidad se reduce con cada venta.

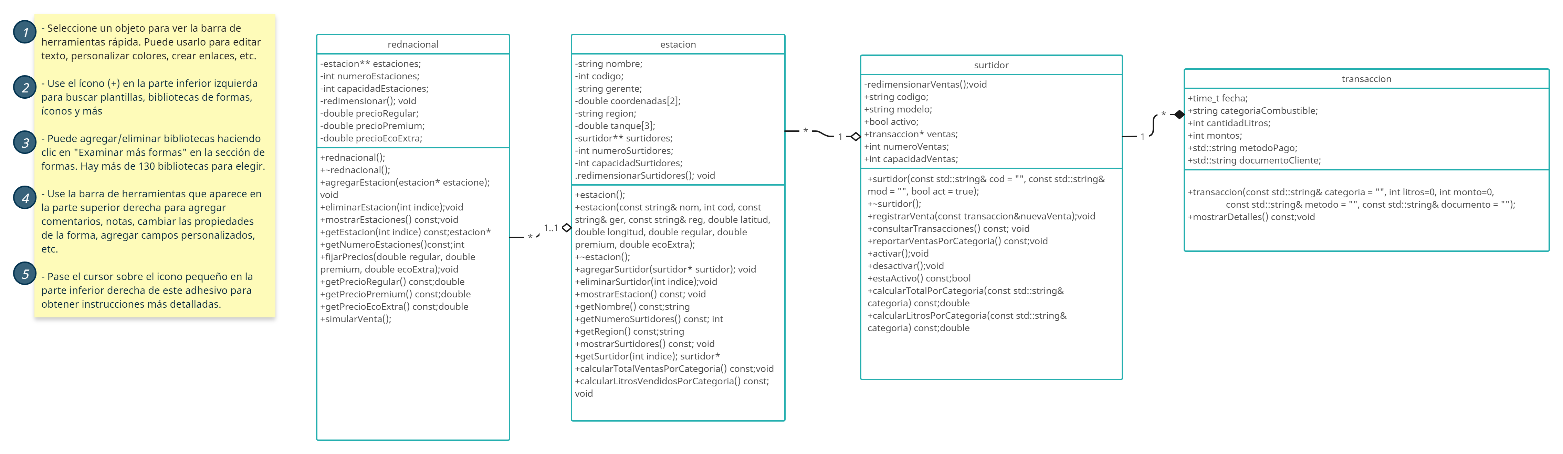
Ventas:

* Registro de ventas diarias: Cada surtidor registra ventas, que incluyen información detallada como la fecha, hora, categoría de combustible, cantidad, método de pago (efectivo, tarjeta de débito, tarjeta de crédito), número de documento del cliente, y el monto total.
* Validación del inventario en tiempo real: En cada venta, se verifica la disponibilidad de combustible en el tanque. Si no hay suficiente combustible para cubrir la venta, solo se cobra la cantidad disponible.
* Actualización automática del inventario: Cada venta reduce la cantidad de combustible disponible en el tanque correspondiente.

Solución

Hacer 4 clases la primera será la clase rednacional la cual gestiona todas las estaciones del servicio (agregar y eliminar estaciones), calcula el total de ventas y gestiona los precios. La idea es que en un arreglo dinámico se almacenes las estaciones. La siguiente es la clase estación la cual representa una estación de servicio y tienes varios surtidores (entre 2 y12) y un tanque donde se almacenan cada combustible, aparte de tener las características de dicha estación como el nombre, ubicación entre otras, está en un arreglo dinámico almacenara los surtidores que esta tenga. La clase surtidor mira un surtidor de una estación, tiene sus características y las ventas se guardaran en un arreglo dinámico de actual surtidor y por último la clase transacción la que representa una venta realizada por cualquier cantidad de combustible vendido y su método de pago

1. Diagrama de clases de la solución planteada. Adicionalmente, describa en alto nivel la lógica de las tareas que usted definió para aquellos subprogramas cuya solución no sea trivial.



Lógicas implementadas

Para la Simulación de ventas en una estación primero este subprograma se encarga de simular una venta de gasolina seleccionando surtidores activos en una estación y procesando una transacción aleatoria de litros y montos(montos es las cantidad a pagar).La lógica utilizada para esto se puede separar en estos procesos:

1 Selecciona aleatoriamente una estación que tenga surtidores activos.

2 en caso no haya estaciones o surtidores activos, notifica al usuario que no se puede realizar la venta.

3 Si hay surtidores activos, selecciona uno de ellos de forma aleatoria.

4 Determina una cantidad aleatoria de litros para la venta (entre 3 y 20 litros).

5 Elige la categoría de combustible según la región de la estación (por ejemplo, "Norte" utiliza EcoExtra).

6 Usa los precios previamente definidos para calcular el monto total.

7 Registra la transacción en el surtidor.

8 Muestra los detalles de la venta, como el tipo de combustible, litros vendidos y el monto total.

Segundo eliminación de una estación de servicio, este subprograma se busca eliminar una estación, pero solo si no tiene surtidores activos. paso:

1 Verifica si la estación tiene surtidores. Si no tiene surtidores, se puede eliminar sin más.

2 Si tiene surtidores, verifica que todos los surtidores estén desactivados.

3 Si todos los surtidores están desactivados, elimina la estación liberando correctamente la memoria dinámica asociada a ella.

4 Si algún surtidor está activo, informa al usuario que la estación no se puede eliminar hasta que todos los surtidores estén desactivados.

Tercero Registrar una venta en un surtidor, este subprograma gestiona la lógica para registrar una transacción en un surtidor específico. Se busco

1Primero, el usuario selecciona una estación de servicio y luego un surtidor en particular.

2Verifica que el surtidor esté activo, si no, no se puede registrar la venta.

3Solicita al usuario la categoría de combustible (Regular, Premium, EcoExtra).

4Solicita la cantidad de litros que desea vender y los datos de la venta (monto, método de pago, documento del cliente).

5Registra la transacción en el surtidor y actualiza el historial de ventas del surtidor.

6Muestra un mensaje de confirmación y los detalles de la venta realizada.

Cuarto la gestión de memoria dinámica para estaciones y surtidores, Estos subprograma se encarga de gestionar el arreglo dinámico de estaciones y surtidores (de forma parecida ocurre lo mismo con rednacional y estacion)

1 El programa usa un arreglo dinámico de punteros para gestionar las estaciones y los surtidores.

2 Al agregar una nueva estación o surtidor, verifica si el arreglo ha alcanzado su capacidad máxima.

3 Si se alcanza la capacidad máxima, se redimensiona el arreglo dinámico duplicando su tamaño, copiando los elementos existentes a un nuevo arreglo más grande, y liberando la memoria del arreglo anterior.

4 Al eliminar una estación o surtidor, desplaza los elementos del arreglo hacia atrás para llenar el espacio vacío, y finalmente, reduce el número de elementos registrados en la estructura.

5 Gestiona correctamente la liberación de memoria para evitar fugas de memoria al eliminar estaciones o surtidores.

Y por el ultimo el quinto fijar los precios del combustible este subprograma permite al usuario establecer los precios de las tres categorías de combustible (Regular, Premium, EcoExtra) para toda la red (se usa en const para evitar variables globales que me puedan perjudicar).

1 Solicita al usuario los precios para cada categoría de combustible.

2 Actualiza los precios de todas las estaciones de la red con estos nuevos valores.

3 Los precios fijos se utilizarán en las ventas posteriores para calcular el monto total de las transacciones.

1. Algoritmos implementados debidamente intra-documentados.

void rednacional::simularVenta() {

if (numeroEstaciones == 0) {

std::cout << "No hay estaciones disponibles para simular ventas.\n";

return;

}

La función primero verifica si hay estaciones disponibles en la red nacional. Si no hay estaciones, la simulación no puede continuar, y por lo tanto, imprime un mensaje de error y sale de la función.

// Seleccionar aleatoriamente una estación que tenga surtidores activos

srand(time(0));

estacion\* est = nullptr;

do {

int indiceEstacion = rand() % numeroEstaciones;

est = estaciones[indiceEstacion];

} while (est->getNumeroSurtidores() == 0);

Aquí se utiliza la función rand() para seleccionar aleatoriamente una estación dentro de la red. Se repite el proceso hasta que se encuentra una estación que tenga al menos un surtidor registrado (si no tiene surtidores, no se puede realizar la venta). Se garantiza que siempre se elige una estación con surtidores.

// Verificar si la estación tiene surtidores activos

bool surtidoresActivos = false;

for (int i = 0; i < est->getNumeroSurtidores(); ++i) {

if (est->getSurtidor(i)->estaActivo()) {

surtidoresActivos = true;

break;

}

}

if (!surtidoresActivos) {

std::cout << "No hay surtidores activos en la estación seleccionada.\n";

return;

}

Una vez seleccionada la estación, la función verifica si esta estación tiene surtidores **activos**. Si no hay surtidores activos, se imprime un mensaje y se sale de la función, ya que no es posible simular una venta sin surtidores funcionales.

// Seleccionar aleatoriamente uno de los surtidores activos

int indiceSurtidor;

do {

indiceSurtidor = rand() % est->getNumeroSurtidores();

} while (!est->getSurtidor(indiceSurtidor)->estaActivo());

surtidor\* surt = est->getSurtidor(indiceSurtidor);

En este punto, la función selecciona aleatoriamente uno de los surtidores **activos** de la estación. Se sigue intentando seleccionar un surtidor hasta que se encuentre uno que esté activado. Una vez encontrado, se guarda el surtidor en la variable surt.

// Generar aleatoriamente entre 3 y 20 litros de combustible

int litrosVendidos = 3 + rand() % 18;

// Determinar la categoría de combustible según la región de la estación

std::string categoria;

double precioPorLitro;

if (est->getRegion() == "Norte") {

categoria = "EcoExtra";

precioPorLitro = getPrecioEcoExtra()\*1.3; // Usar el precio fijo de EcoExtra

} else if (est->getRegion() == "Centro") {

categoria = "Regular";

precioPorLitro = getPrecioRegular()\*1.2; // Usar el precio fijo de Regular

} else {

categoria = "Premium";

precioPorLitro = getPrecioPremium()\*1.4; // Usar el precio fijo de Premium

}

Dependiendo de la **región** de la estación (Norte, Centro o Sur), se selecciona una categoría de combustible. Cada región está asociada a un tipo de combustible y a un precio por litro, que se fija utilizando los precios globales de la red nacional (con factores multiplicadores que varían según la región).

// Definir el precio por litro según la categoría de combustible

int montoTotal = litrosVendidos \* precioPorLitro;

// Crear la transacción y registrarla

transaccion nuevaVenta(categoria, litrosVendidos, montoTotal, "Efectivo", "123456789");

surt->registrarVenta(nuevaVenta);

// Mostrar los detalles de la transacción

std::cout << "\n=== Transaccion registrada ===\n";

nuevaVenta.mostrarDetalles();

}

d. Problemas de desarrollo que afrontó.

Al inicio del desarrollo de este proyecto, consideré trabajar con punteros simples para gestionar las estaciones y surtidores dentro de la red. pero, pronto tuve varios problemas que me obligaron a volver al análisis. Utilizar punteros sencillos me generaba “copias innecesarias” de los objetos, lo que complicaba la gestión de los datos y causaba errores al intentar manipular los objetos.

El principal problema que enfrenté fue que, al trabajar con punteros simples, cada vez que manipulaba un objeto de tipo estacion o surtidor, se creaban copias de esos objetos en lugar de trabajar sobre las instancias originales. Esto provocaba inconsistencias en los datos (por ejemplo -12309), ya que los cambios no se reflejaban en las verdaderas instancias almacenadas en la estructura de datos (perdí 2 días tratando de arreglarlo)

La solución a este problema fue cambiar el enfoque y utilizar dobles punteros para gestionar las relaciones entre las clases. Los dobles punteros permiten gestionar un arreglo dinámico de punteros que apuntan directamente a las instancias reales de las estaciones y surtidores. De esta manera, se evita la creación de copias innecesarias y se asegura que todas las operaciones (agregar, eliminar, modificar) se realicen sobre los objetos correctos.

Además, trabajar con dobles punteros resultó ser más eficiente en cuanto a la gestión de memoria. Con punteros simples, liberar memoria de forma correcta resulto ser más complejo, ya que cada vez que se elimina un objeto, se corre el riesgo de dejar referencias colgantes (fugas). Con los dobles punteros, puedo manejar directamente los punteros a objetos y asegurarme de que cada estación o surtidor esté correctamente eliminado de la memoria sin problemas.

Otra ventaja del uso de dobles punteros es la flexibilidad que ofrecen para gestionar el crecimiento y reducción de los arreglos dinámicos. Cuando una red necesita expandirse y agregar más estaciones o surtidores, puedo simplemente redimensionar el arreglo de punteros, sin necesidad de duplicar datos o mover objetos innecesariamente.

e. Evolución de la solución y consideraciones para tener en cuenta en la implementación.

La **evolución de la solución** durante el desarrollo de este desafío fue una transición desde un diseño inicial “simple”, basado en el uso de punteros simples, hacia un enfoque más sofisticado utilizando **dobles punteros** y mejor gestión de memoria dinámica. Esta evolución fue técnicamente obligada por las dificultades y problemas que surgieron al manejar la complejidad del programa, especialmente cuando se trataba de agregar, eliminar y modificar elementos como **estaciones** y **surtidores**, y asegurar que los **datos no se duplicaran innecesariamente** ni se corrompieran por errores de referencia.

Consideraciones:

1Uso adecuado de memoria: En futuros desarrollos y trabajos, se debe continuar prestando atención a la correcta liberación de memoria dinámica. Especialmente en programas que utilizan arreglos dinámicos de objetos.

2Escalabilidad del sistema: Aunque el uso de dobles punteros ha resuelto los problemas que presentaba antes, es importante considerar la escalabilidad del sistema a medida que aumentan las estaciones y surtidores. Puede ser beneficioso tratar de pensar en otras posibles soluciones.

3Validaciones y excepciones: La implementación actual asegura validaciones básicas, como evitar la eliminación de estaciones con surtidores activos entre otras. A medida que el sistema crece hay que tener en cuenta implementar más controles de errores y excepciones para manejar situaciones inesperadas. Por ejemplo actualmente en el main si escribes norte en vez de Norte seguramente el programa no responderá con normalidad y he de ahí la importancia de las excepciones que tendré en cuenta en el futuro (proyecto final)