Desafío 2

Andrés Romero Capachero

Nikolas Geovanny Ortega Suarez

Informática II

Aníbal guerra

Augusto Salazar

Universidad de Antioquia

Medellín, Antioquia

10 de octubre de 2024

1. **Análisis del problema y solución propuesta:**

En una primera vista del desafio 2 encontramos la petición de un cliente, en este caso una empresa petrolifera con un modelo de negocios de estaciones de gasolina a nivel nacional distribuida en las tres regiones del país(norte, sur, centro) con una red de distribución con 3 tipos diferentes de gasolina (regular, premium y ecoExtra), cada estación cuenta con un nombre, un código identificador un gerente y una ubicación geográfica en GPS que permite ubicarla en 1 de las 3 regiones, cada una con un tanque central que almacena los 3 tipos de gasolina de manera no necesariamente uniforme, al tanque se conectan de 2 a 12 surtidores de los cuales se hace la venta directa a los usuarios, a su vez cada surtidor tiene su código, modelo, registran sus ventas individuales diarias, con hora, fecha, cantidad, categoría de combustible, y el método de pago correspondiente (Efectivo, Tdebito, Tcredito) la identificación del cliente, además cada surtidor debe actualizar su disponibilidad con cada venta, el tanque también debe actualizar, con el fin de saber el abastecimiento de la estación.

Dicha problemática lleva al desarrollo e implementación del código que busca ofrecer dichas funcionalidades al cliente mediante el uso del paradigma de POO, para su desarrollo lo principal y a lo que se dedicaron dos días del plazo, es el modelamiento de un diagrama de clases que tenga en cuenta la verdadera y real amplitud de lo solicitado por el cliente, aun así abiertos a errores y cambios a lo largo del desarrollo del proyecto; sin embargo, siempre evitando cambios a futuro y trabajar con un modelo establecido y funcional que no requiera actualizaciones, pero eso es demasiado ideal para el nivel que manejamos, por eso, no aseveramos que este diagrama se mantenga intacto a lo largo del desarrollo por nuestra propia inexperiencia, a lo largo del informe se detallaran las problemáticas, los descubrimientos y los probables cambios que veamos necesarios.

NOTA: primer plano de diagrama de clases para guiarse durante el primer momento de desarrollo (puede tener cambios a lo largo de la implementacion)10/10/2024

los días 8-10 se destinaron a la investigación de la problemática propuesta, profundizar conocimientos sobre clases métodos e instancias y su correcto uso, al final del día 10 se propone el siguiente diagrama

**Clase Abstracta**Entidad

* **Descripción**: Es una clase base que proporciona atributos y métodos comunes para otras clases.
* **Atributos**:
  + codigo: Un identificador único para la entidad.
  + nombre: El nombre de la entidad.
* **Métodos**:
  + getCodigo(): Devuelve el código de la entidad.
  + getNombre(): Devuelve el nombre de la entidad.
  + setCodigo(codigo): Establece el código de la entidad.
  + setNombre(nombre): Establece el nombre de la entidad.

**Clase**RedNacional**(Derivada de**Entidad**)**

* **Descripción**: Gestiona la red de estaciones de servicio en todo el país.
* **Atributos**:
  + estaciones: Lista de todas las estaciones de servicio.
  + precios\_combustible: Precios del combustible por región.
* **Métodos**:
  + agregarEstacion(estacion): Añade una nueva estación a la red.
  + eliminarEstacion(codigo): Elimina una estación de la red si no tiene surtidores activos.
  + calcularVentasTotales(): Calcula el monto total de las ventas en todas las estaciones.
  + fijarPreciosCombustible(region, precios): Establece los precios del combustible para una región específica.
  + buscarEstacion(codigo): Busca y devuelve una estación por su código.

**Clase**EstacionServicio**(Derivada de**Entidad**)**

* **Descripción**: Representa una estación de servicio individual.
* **Atributos**:
  + gerente: El gerente de la estación.
  + region: La región donde se encuentra la estación.
  + ubicacion: Coordenadas GPS de la estación.
  + capacidad\_tanque: Capacidad de almacenamiento de combustible por categoría.
  + surtidores: Lista de surtidores en la estación.
* **Métodos**:
  + agregarSurtidor(surtidor): Añade un surtidor a la estación.
  + eliminarSurtidor(codigo): Elimina un surtidor de la estación.
  + activarSurtidor(codigo): Activa un surtidor para que pueda vender combustible.
  + desactivarSurtidor(codigo): Desactiva un surtidor para que no pueda vender combustible.
  + buscarSurtidor(codigo): Busca y devuelve un surtidor por su código.
  + calcularVentas(): Calcula el monto total de las ventas en la estación.
  + asignarCapacidadTanque(): Asigna una capacidad aleatoria al tanque de la estación para cada categoría de combustible.

**Clase**Surtidor**(Derivada de**Entidad**)**

* **Descripción**: Representa un surtidor de combustible en una estación de servicio.
* **Atributos**:
  + modelo: El modelo del surtidor.
  + ventas: Lista de ventas realizadas por el surtidor.
  + activo: Indica si el surtidor está activo o no.
* **Métodos**:
  + registrarVenta(fecha, hora, cantidad, categoria, metodo\_pago, documento\_cliente, monto): Registra una venta de combustible.
  + activar(): Activa el surtidor.
  + desactivar(): Desactiva el surtidor.

**Instancias**

**Instancia de**RedNacional

* **Nombre:** red
* **Descripción**: Representa la red nacional de estaciones de servicio.
* **Atributos**:
  + estaciones: Lista de estaciones de servicio.
  + precios\_combustible: Diccionario de precios por región.

**Instancia de**EstacionServicio

* **Nombre:** estacion
* **Descripción**: Representa una estación de servicio específica.
* **Atributos**:
  + nombre: “Estación Central”
  + codigo: “001”
  + gerente: “Juan Pérez”
  + region: “Centro”
  + ubicacion: Coordenadas GPS (6.2442, -75.5812)
  + capacidad\_tanque: Capacidad de almacenamiento por categoría {“Regular”: 150, “Premium”: 120, “EcoExtra”: 130}
  + surtidores: Lista de surtidores en la estación.

**Instancia de**Surtidor

* **Nombre:** surtidor1
* **Descripción**: Representa un surtidor específico en una estación de servicio.
* **Atributos**:
  + codigo: “S001”
  + modelo: “ModeloX”
  + ventas: Lista de ventas realizadas por el surtidor.
  + activo: Indica si el surtidor está activo (True).

**Clase Abstracta**Entidad

* **Descripción**: Es una clase base que proporciona atributos y métodos comunes para otras clases.
* **Razón**: Se utiliza para definir atributos y métodos comunes que serán heredados por otras clases. Esto evita la duplicación de código y facilita la gestión de atributos comunes como codigo y nombre.

**Clase**RedNacional**(Derivada de**Entidad**)**

* **Descripción**: Gestiona la red de estaciones de servicio en todo el país.
* **Razón**: Esta clase maneja la colección de todas las estaciones de servicio y sus operaciones a nivel nacional, como agregar o eliminar estaciones, calcular ventas totales y fijar precios de combustible. Hereda de Entidad para reutilizar los atributos comunes.

**Clase**EstacionServicio**(Derivada de**Entidad**)**

* **Descripción**: Representa una estación de servicio individual.
* **Razón**: Cada estación de servicio tiene atributos específicos como gerente, región, ubicación y capacidad del tanque. También maneja sus surtidores y ventas. Hereda de Entidad para reutilizar los atributos comunes.

**Clase**Surtidor**(Derivada de**Entidad**)**

* **Descripción**: Representa un surtidor de combustible en una estación de servicio.
* **Razón**: Cada surtidor tiene atributos específicos como modelo y ventas realizadas. También puede estar activo o inactivo. Hereda de Entidad para reutilizar los atributos comunes.

**Instancias**

**Instancia de**RedNacional

* **Nombre**: red
* **Descripción**: Representa la red nacional de estaciones de servicio.
* **Razón**: Es una instancia de la clase RedNacional que se utiliza para gestionar todas las estaciones de servicio en el país. Contiene la lista de estaciones y los precios del combustible por región.

**Instancia de**EstacionServicio

* **Nombre**: estacion
* **Descripción**: Representa una estación de servicio específica.
* **Razón**: Es una instancia de la clase EstacionServicio que representa una estación de servicio en particular. Contiene información específica de esa estación, como su nombre, código, gerente, región, ubicación, capacidad del tanque y surtidores.

**Instancia de**Surtidor

* **Nombre**: surtidor1
* **Descripción**: Representa un surtidor específico en una estación de servicio.
* **Razón**: Es una instancia de la clase Surtidor que representa un surtidor en particular dentro de una estación de servicio. Contiene información específica del surtidor, como su código, modelo, ventas realizadas y estado (activo o inactivo).

**Resumen**

* **Clases**: Definen la estructura y comportamiento de los objetos. Las clases RedNacional, EstacionServicio y Surtidor heredan de Entidad para compartir atributos comunes.
* **Instancias**: Son objetos concretos creados a partir de las clases. red, estacion y surtidor1 son ejemplos de instancias que representan entidades específicas en el sistema.

El diagrama se rige bajo una clase abstracta “Entidad” la cual contiene atributos y métodos comunes que todas las demás clases heredaran,además usando métodos getters y setters para obtener y establecer valores , dando una misma línea de lógica para la implementación de las clases , de ella salen las clases Surtidor, RedNacional y EstacionServicio siendo dependientes unas de otras lo cual es un reto a la hora de programar ya que es muy fácil errar en nombres e invocación de métodos, cada una de las clases cuenta con sus atributos y métodos tanto públicos como privados en el diagrama se hace una breve descripción de cada uno.

(12/10/2024) para el cuarto día de desarrollo aun no se hacen avances significativos, por decisión del equipo separamos el trabajo para que uno desarrollara las clases RedNacional y EstacionServicio, mientras el otro trabajaba en la clase abstracta(la necesaria para lo demás, ya que de ella heredan las demás clases) y la clase Surtidor, teniendo en cuenta la lógica necesaria para el control de ventas, por cuestiones personales del equipo no se respetó dicho acuerdo en los plazos establecidos, sin embargo, se trabaja la clase de red nacional y estación de servicio durante el día 12 de una manera básica y a grozo modo, buscando poner avances en el git y crear un avance real en el proyecto, se sube un archivo al git el día 13 de octubre con dicho bosquejo a falta de dos días para la entrega el 15 de octubre, con mas trabajo que avances.

(14/10/2024) En vista de la mala organización de trabajo y de tiempo por parte del equipo, comenzamos de nuevo con un criterio de orden mas estructurado separando el código en archivos independientes .h y .cpp para luego ser incluidos como librerías en el main, teniendo en los headers la invocación de métodos y en el source la implementación de cada método, dicho cambio tuvo que ser analizado cuidadosamente ya que era nuevo para ambos sin embargo daba mucho mas orden a la lectura y a la lógica del programa, se desarrollo en primera instancia el archivo entidad.h( por sus métodos simples no requirió un archivo.cpp adicional):

#ifndef ENTIDAD\_H

#define ENTIDAD\_H

#include <string>

using namespace std;

class Entidad {

private:

string codigo; // Identificador único de la unidad

string nombre; // Nombre de la entidad

string ubicacion; // Coordenadas GPS de la entidad

public:

// Constructor por defecto

Entidad() = default;

// Constructor con parámetros: da valores específicos

Entidad(const string& codigo, const string& nombre, const string& ubicacion)

: codigo(codigo), nombre(nombre), ubicacion(ubicacion) {}

// Método getter para obtener el valor de "codigo"

string getCodigo() const {

return codigo;

}

// Método getter para obtener el valor de "nombre"

string getNombre() const {

return nombre;

}

// Método getter para obtener el valor de "ubicacion"

string getUbicacion() const {

return ubicacion;

}

// Método setter para establecer el valor de "codigo"

void setCodigo(const string& codigo) {

this->codigo = codigo;

}

// Método setter para establecer el valor de "nombre"

void setNombre(const string& nombre) {

this->nombre = nombre;

}

// Método setter para establecer el valor de "ubicacion"

void setUbicacion(const string& ubicacion) {

this->ubicacion = ubicacion;

}

// Destructor virtual asegura que las clases derivadas se destruyan correctamente, asegurando una correcta liberación de recursos

virtual ~Entidad() = default;

};

#endif

Luego se crea los archivos RedNacional.h y RedNacional.cpp:

#ifndef REDNACIONAL\_H

#define REDNACIONAL\_H

#include <string>

#include "Entidad.h"

#include "EstacionServicio.h" // Incluimos la cabecera de EstacionServicio

using namespace EstacionServicioNS;

namespace RedNacionalNS {

using namespace std;

class RedNacional : public Entidad {

public:

RedNacional(); // Constructor por defecto

// Métodos para gestionar la red de estaciones de servicio

void agregarEstacion(EstacionServicio\* estacion); // Agregar una nueva estación de servicio a la red

void eliminarEstacion(const string& codigo); // Eliminar una estación de servicio de la red

double calcularVentasTotales(); // Calcular el monto total de las ventas en todas las estaciones de la red

void fijarPreciosCombustible(const string& region, double precios); // Fijar los precios del combustible por región

EstacionServicio\* buscarEstacion(const string& codigo); // Buscar y devolver una estación de servicio por su código

void mostrarInformacion(); // Mostrar información de todas las estaciones en la red

string obtenerRegionPorGPS(const string& ubicacion); // Obtener la región de una estación según su ubicación GPS

private:

EstacionServicio\* estaciones[100]; // Arreglo de punteros a estaciones de servicio (máximo 100 estaciones)

int numEstaciones; // Contador de estaciones en la red

double precios\_combustible[3]; // Arreglo para almacenar los precios del combustible

Su archivo .h muestra lo comentado, además incluye la cabecera de estación de servicio al ser dependiente de ella en algunos parámetros del mismo modo se crea la cabecera Estacion de servicio.h

using namespace std;

using namespace SurtidorNS;

namespace EstacionServicioNS {

class EstacionServicio : public Entidad {

public:

EstacionServicio(const string& codigo, const string& nombre, const string& ubicacion, const string& gerente, const string& region);

// Métodos para gestión de ventas y capacidad

double calcularVentas() const;

double calcularVentasPorCategoria(int categoria) const;

void asignarCapacidadTanque();

void reportarLitrosVendidosPorCategoria() const;

void simularVenta(double litros, int categoria);

// Otros métodos

void consultarHistoricoTransacciones() const;

void mostrarInformacion() const;

private:

string gerente;

string region;

double capacidad\_tanque[3]; // Capacidad de almacenamiento para 3 categorías de combustible

Surtidor\* surtidores[12]; // Máximo de 12 surtidores

int numSurtidores;

double precios\_combustible[3]; // Para manejar precios de combustible por categoría

};

}

#endif // ESTACIONSERVICIO\_H

La cual en un principio incluía métodos para la gestión de surtidores que iban en concordancia con el archivo surtidor.h el cual antes tenía tanto su archivo punto h como su archivo CPP,durante su implementación tuvo muchos problemas con los métodos y sus usos en los diferentes sources, se opto por escribir todo en el header y la clase surtidor no separarla por los fallos que ocasionaba, además perdiendo funcionalidades esenciales en la clase Estacionservicio como agregar surtidor , eliminar surtidor, activar y desactivar surtidor, todo esto hasta el día 17 con problemas de implementación casi hasta el final, el manejo del git fue muy deficiente en este desafio por el equipo y sin animo excusarlo dejamos en este informe plasmada la realidad del desarrollo

A la hora de codificar el main con las clases hechas (a medias algunas) encontramos diferentes problemas que me gustaría detallar en la siguiente lista

1. **Declaración incompleta de constructores**:
   * **Explicación**: Los constructores no estaban completamente definidos en el archivo .cpp, resultando en errores de referencia indefinida. Se debe asegurar que los constructores estén correctamente implementados.
2. **Errores de acceso a miembros**:
   * **Explicación**: Este error ocurre cuando se intenta acceder a un miembro de una clase que no está completamente definido. Asegúrate de que las cabeceras necesarias están incluidas y de que las clases están completamente definidas antes de su uso.
3. **Inclusiones de cabeceras innecesarias**:
   * **Explicación**: Incluir cabeceras que no se usan directamente en un archivo .cpp causa advertencias. Deben incluirse solo las cabeceras necesarias.
4. **Eliminar métodos obsoletos**:
   * **Explicación**: Métodos como agregarSurtidor en EstacionServicio ya no existían, pero seguían siendo referenciados. Se deben eliminar las referencias a métodos obsoletos y ajustar la lógica en consecuencia.
5. **Variables no utilizadas**:
   * **Explicación**: Declaraciones de variables que no se usan en el código, resultando en advertencias. Se deben eliminar estas variables o asegurarse de que se usen correctamente.
6. **Uso restringido de ciertas librerías (como**sstream, vector, etc**)**:
   * **Explicación**: Restricciones en el uso de ciertas librerías impiden su uso en el proyecto. Se debe ajustar el código para evitar el uso de estas librerías y buscar alternativas.
7. **Errores de saltos en**switch:
   * **Explicación**: "Cannot jump from switch statement to this case label" debido a variables inicializadas después de un salto en un switch. Se debe reestructurar el switch para que las variables se inicialicen correctamente sin saltar las inicializaciones.
8. **Métodos vacíos o sin implementar**:
   * **Explicación**: Métodos como calcularVentasTotales en RedNacional estaban vacíos. Se debe implementar la lógica necesaria para cada método según los requerimientos.
9. **Falta de diagrama de clases y descripción en alto nivel**:
   * **Explicación**: No se disponía de un diagrama de clases y una descripción en alto nivel de la lógica. Se deben crear un diagrama de clases y proporcionar descripciones en alto nivel de la lógica de los subprogramas.

Durante la evolución de la solución del proyecto íbamos notando la importancia de un bien detallado y hecho diagrama de clases y a la hora de programar aferrarse al diagrama propuesto, durante nuestra propia implementación pecamos de no darle el valor suficiente a la complejidad del problema dando como resultado una implementación no pobre sino más bien poco estructurada y demasiado básica parte de la evolución y las consideraciones que tenemos sobre el proyecto son la autocrítica y el reconocimiento de prácticas mejorables durante la implementación del proyecto.

(17/10/2024) con la precaria situación del proyectos nos determinamos a cumplir con el trabajo de la mejor manera posible dedicando todo el día del 17 de octubre a la reestructuración del diagrama de clases agregando funcionalidades importantes como el histórico de ventas y la comprobación de fugas, además de un mejor uso de arreglos cambiando varios estáticos por dinámicos y mejor manejo de datos utilizando punteros y puntero de punteros, se logra la entrega (1 hora tarde) del diagrama de clases definitivo y el programa funcional en un archivo zip en el repositorio, excusamos nuestra tardanza, obviando el hecho de que nos exponemos a represalias, con la firme convicción de entregar un trabajo bien hecho antes que uno mediocre e incompleto.