INFORME DESAFIO II

JOSE GABRIEL GIRALDO FLOREZ

UNIVERSIDAD DE ANTIOQUIA

INFORMATICA II

ANIBAL JOSE GUERRA SOLER

17/10/2024

**Análisis del problema:**

En la fase inicial del análisis, se enfocó en identificar las clases clave que serían necesarias para el sistema, definiendo sus respectivos atributos y métodos. Además, se consideraron las relaciones entre las clases para guiar el desarrollo y la implementación de la estructura general del sistema.

Para determinar las clases a implementar, se respondieron preguntas clave como: **¿Cuál es el objetivo del sistema?, ¿Cuáles son las entidades principales del sistema?, ¿Cuáles son los atributos y métodos que caracterizan a cada entidad?, ¿Qué relaciones existen entre las entidades?, y ¿Existen restricciones o reglas de negocio que puedan influir en el diseño?** Estas preguntas ayudaron a establecer una base sólida para el diseño de las clases y su interacción.

Posteriormente, se procedió con la creación de un **diagrama de clases UML** utilizando herramientas virtuales como **Lucidchart**, lo cual permitió visualizar y organizar de manera más clara las entidades, relaciones y funcionalidades del sistema. Esto facilitó no solo la planificación, sino también la implementación de cambios y mejoras durante el desarrollo, garantizando una estructura más manejable y coherente.

Con el diagrama inicial definido, se comenzó a establecer los archivos **.h** correspondientes a cada clase, donde se determinaron finalmente los atributos, métodos y las funcionalidades específicas. Este enfoque estructurado permitió avanzar hacia la fase de desarrollo, donde se implementaron las soluciones necesarias para asegurar el correcto funcionamiento del sistema.

Este proceso culminó en la producción de un código más organizado, flexible y preparado para futuras expansiones o modificaciones.

**Diagrama de clase UML:**

Dentro del diagrama, se destaca:

**El método calcularTransacciones de la clase EstacionDeServicio.** La función principal de este método es recorrer los surtidores asociados a una estación de servicio para sumar el monto total de las transacciones registradas.

La lógica de este método se basa en los siguientes pasos:

1) Identificar los surtidores activos de la estación.

2) Para cada surtidor activo, acceder a las transacciones almacenadas.

3) Calcular el valor total de las transacciones de cada surtidor, considerando la cantidad de litros vendidos y el precio por litro previamente establecido en el programa.

4) Finalmente, sumar los totales de todos los surtidores y devolver o mostrar el monto global de las transacciones realizadas en esa estación de servicio.

Este enfoque asegura que se capture con precisión el rendimiento económico de la estación en función de las ventas de combustible realizadas por cada surtidor.

**Método verificarFugas (Tanque&, Surtidor\*, int) de EstacionDeServicio**

Función: Este método verifica si hay discrepancias entre la cantidad de combustible que los surtidores reportan haber dispensado y los niveles de combustible registrados por el tanque, con el objetivo de detectar posibles fugas.

**Lógica:**

1) Recibe como parámetros el tanque y un arreglo de surtidores.

2) Compara la cantidad de combustible dispensado por los surtidores con las cantidades que el tanque ha registrado como consumidas.

3) Si se detecta una discrepancia significativa (es decir, los surtidores han vendido más combustible del que el tanque ha registrado), se señala una posible fuga.

3) En caso de detectar una fuga, se registra y reporta para su revisión oportuna.

**Método reportarTransaccion (int, long int, const string&, const string&, const string&) de Surtidor**

Función: Permite registrar una nueva transacción asociada a un surtidor específico, incluyendo detalles como cantidad de litros vendidos, método de pago, entre otros.

**Lógica:**

1) Recibe como parámetros la cantidad de litros vendidos, el monto total de la transacción, el tipo de combustible, el método de pago y los datos del usuario.

2) Crea una nueva instancia de la clase Transaccion con los datos proporcionados.

3) Almacena la transacción en el surtidor, con un límite de hasta 100 transacciones.

4) Actualiza el contador de transacciones del surtidor.

5) De manera opcional, actualiza la capacidad restante del tanque para reflejar los litros vendidos.

**Método gasolinaVendida (unsigned int, const string&) de Tanque**

Función: Ajusta la capacidad actual del tanque en función de la cantidad de gasolina vendida para un tipo específico de combustible.

**Lógica:**

1) Recibe como parámetros la cantidad de litros vendidos y el tipo de combustible (regular, premium o eco-extra).

2) Según el tipo de combustible, descuenta los litros correspondientes de la capacidad actual del tanque.

3) Devuelve el total de litros vendidos o actualiza un registro interno del tanque con la nueva capacidad.

**Método establecerPrecio (string&, string&, unsigned short int) de RedNacional**

Función: Define el precio del combustible en una región específica y para un tipo particular de combustible.

**Lógica:**

1) Recibe la región, el tipo de combustible y el nuevo precio como parámetros.

2) Accede a la matriz precioPorRegion y actualiza el precio según la combinación de región y tipo de combustible.

3) Almacena el nuevo precio en la estructura correspondiente y actualiza las estaciones de servicio ubicadas en esa región con el precio actualizado.

Estos subprogramas muestran cómo se abordan procesos complejos dentro del sistema, tales como el cálculo de transacciones, la gestión de precios y la detección de problemas operativos, como las fugas de combustible. Cada método requiere una interacción cuidadosa entre varias clases, garantizando así la integridad y la funcionalidad del sistema.

**Pseudocódigo:**

**Algoritmo 1 (CalcularTransacciones)**

**Clase: EstacionDeServicio**

// Algoritmo para calcular el total de transacciones en una estación de servicio

**Función calcularTransacciones()**

totalTransacciones = 0

// Obtener los surtidores activos de la estación

para cada surtidor en surtidoresActivos()

// Obtener las transacciones de cada surtidor y sumar el valor total

para cada transaccion en surtidor.getTransacciones()

totalTransacciones += transaccion.getCantidadDinero()

retornar totalTransacciones

Fin Función.

**Algoritmo 2 (VerificarFugas)**

**Clase: EstacionDeServicio**

// Algoritmo para verificar posibles fugas de combustible en la estación

Función verificarFugas (Tanque tanque, Surtidor[] surtidores, int numSurtidores)

totalCombustibleVendido = 0

totalCombustibleConsumido = tanque.getCombustibleConsumido()

// Recorre todos los surtidores para calcular el total de litros vendidos

para cada surtidor en surtidores

totalCombustibleVendido += surtidor.getLitrosVendidos()

// Compara los litros vendidos con el combustible registrado como consumido

si totalCombustibleVendido > totalCombustibleConsumido entonces

mostrar "Posible fuga detectada"

retornar verdadero

sino

retornar falso

Fin Función

**Algoritmo 3 (ReportarTransaccion)**

**Clase: Surtidor**

// Algoritmo para reportar una nueva transacción en un surtidor

Funcion reportarTransaccion(int litrosVendidos, long int cantidadDinero,

const string& categoriaCombustible,

const string& metodoPago, const string& documentoUsuario)

// Crear una nueva instancia de la clase Transaccion

nuevaTransaccion = nueva Transaccion(litrosVendidos, cantidadDinero, categoriaCombustible, metodoPago, documentoUsuario)

// Agregar la transacción al arreglo de transacciones del surtidor agregarTransaccion(nuevaTransaccion)

// Actualizar el contador de transacciones del surtidor

incrementarContadorTransacciones()

// Opcionalmente, actualizar la capacidad del tanque

tanque.actualizarCapacidad(categoriaCombustible, litrosVendidos)

Fin Funcion

**Algoritmo 4 (gasolinaVendida)**

**Clase: Tanque**

// Algoritmo para ajustar la capacidad del tanque según la gasolina vendida

Funcion gasolinaVendida(unsigned int litrosVendidos, const string& categoriaCombustible)

// Ajustar la capacidad del tanque según el tipo de combustible

si categoriaCombustible == "Regular" entonces

capacidadActualRegular -= litrosVendidos

sino si categoriaCombustible == "Premium" entonces

capacidadActualPremium -= litrosVendidos

sino si categoriaCombustible == "Eco-Extra" entonces

capacidadActualEcoExtra -= litrosVendidos

retornar litrosVendidos

Fin Funcion

**Algoritmo 5 (establecerPrecio)**

**Clase: RedNacional**

// Algoritmo para establecer el precio del combustible en una región específica

Funcion establecerPrecio(string& region, string& tipoCombustible, unsigned short int nuevoPrecio)

// Acceder a la matriz de precios por región

precioPorRegion[region][tipoCombustible] = nuevoPrecio

// Actualizar el precio en las estaciones de servicio de esa región

para cada estacion en estaciones

si estacion.getRegion() == region entonces

estacion.actualizarPrecio(tipoCombustible, nuevoPrecio)

Fin Funcion

**Problemas afrontados:** El programa presentaba un obstáculo significativo en la gestión de los surtidores de las estaciones de servicio debido a una inconsistencia en su implementación. Los surtidores, una vez creados, eran eliminados de la memoria, impidiendo su asociación con la estación y limitando las funcionalidades relacionadas. Esta problemática se originaba en una falla de diseño que comprometía la integridad de los datos y restringía las operaciones posibles. Para resolver este inconveniente, fue necesario reestructurar la lógica de creación y gestión de los surtidores, garantizando su persistencia y permitiendo una interacción más fluida con el resto del sistema, ampliando así las capacidades del programa.

**Evolución y consideraciones:** La evolución de la solución comenzó con un análisis exhaustivo para definir las entidades principales (EstacionDeServicio, Surtidor, Tanque, Transaccion) y sus relaciones. El diseño se centró en la modularidad, asegurando que cada clase tenga responsabilidades claras, facilitando el mantenimiento y la escalabilidad del sistema.

Se consideró la eficiencia en la implementación de algoritmos como calcularTransacciones y verificarFugas, optimizando su complejidad temporal. También se prestó atención a la flexibilidad para futuras expansiones, como la adición de nuevos tipos de combustible o estaciones. El sistema fue diseñado para ser fácilmente extensible sin cambios drásticos.

Se aplicaron buenas prácticas de programación, como una correcta documentación y nombres descriptivos en el código. Además, se realizaron pruebas exhaustivas para asegurar el buen funcionamiento de los algoritmos en diversas situaciones, mejorando así la robustez y confiabilidad del sistema.

