Universidad de Antioquia
Departamento de Ingenieria Electrónica y de Telecomunicaciones
2598521 - Informatica II

Informe Desafio N°2

Esteban García Lopez

Juan Camilo Agudelo Giraldo

1. Análisis del problema

1.1. Descripción del problema

El problema consiste en controlar la informacíon de las estaciones de servicio o bombas gasolineras en una red nacional mediante un software que permite el acceso y manipulación a la información.

1.2. Objetivo General

Desarrollar un sistema que permita la manipulacion de la información de las estaciones de servicio o bombas gasolineras, utilizando programación orientada a objetos.

1.3. Objetivo específico

Crear clases que definan el problema real y la comunicación que estas mismas tienen, eficazmente.

1.4. Requerimientos funcionales

- Agregar estaciones de servicio
- Eliminar estaciones de servicio de la red naciona sólo si no posee surtidores activos
- Calcular el monto total de las ventas en cada estacion de servicio del país, discriminando por categoría de combustible
- Fijar los precios de combustible para toda la red
- Agregar eliminar un surtidor de una estacion de servicio
- Activar desactivar un surtidor de una estacion de servicio
- Consultar el histórico de transacciones de cada surtidor de la estacion de servicio
- Reportar la cantidad de litros vendida según cada categoría de combustible
- Simular una venta de combustible
- Asignar la capacidad del tanque, con un valor aleatorio entre 100 y 200 litros para cada categoria
- Detectar la existencia de fugas de combustible de cualquier estacion del país
- Simular ventas de combustible

1.5. Requerimientos no funcionales

- Rendimiento optimo para el sistema
- Seguridad en la información
- Diseño intuitivo y experiencia de usuario

1.6. Restricciones

El sistema contiene las siguiente restricciones debido al limite estipulado y a la seleccion de Sqlit como base de datos.

- 2 Terabytes de tamaño maximo para el almacenamiento de datos
- El sistema solo funciona en el entorno local asi como la base de datos
- Uso de tipos de datos con énfasis en el rendimiento

1.7. Análisis de posibles soluciones

Se evaluó el uso de archivos como medio de persistencia para la información versus el uso de bases de datos. La primera opción ofrece una forma menos costosa de guardar información pero sacrifica rendimiento al estar interacturando con el propio sistema operativo asi como su filtro, para el cual se debe recorrer todo el archivo haciendo ineficiente la utilización. La segunda opción es mas costosa en su implementacion por el hecho de utilizar librerias externas, pero aporta una interaccion mucho mas eficiente con los datos asi como una estructura estandarizada.

2. Diagrama de clases

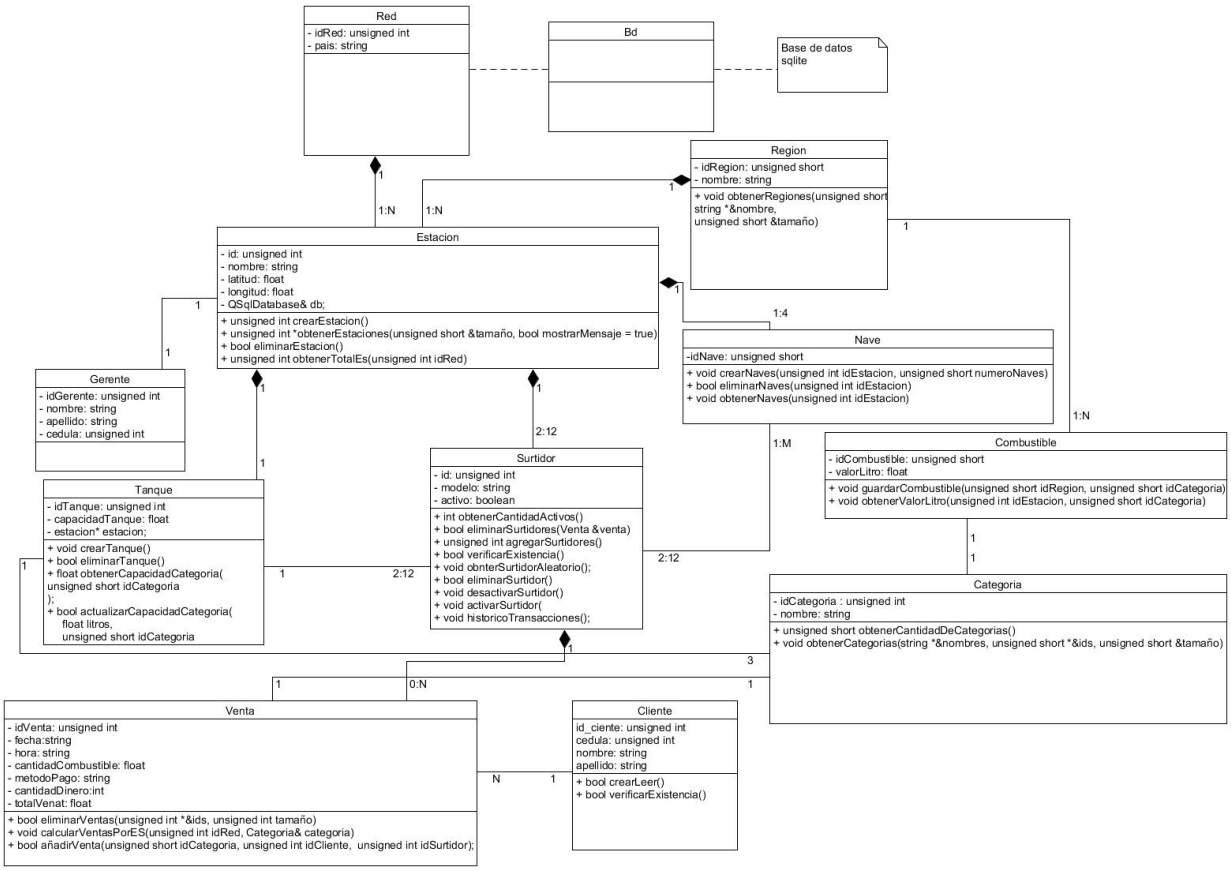

Figura 1: Diagrama de clases (Archivo adjunto en el repositorio)

2.1. Descripción

2.1.1. Estación

- crearEstacion(): método encargado de recoger los datos de la estacion y guardarlo en la base de datos
- *ObtenerEstaciones(): método encargado de consultar todas las estaciones y almacenar sus ids en un array
- obtenerTotalEs(): método encargado de obtener la cantidad de estaciones en una red.
- eliminarEstacion(): método para eliminar una estación de una red.

2.1.2. Tanque

- crearTanque(): método encargado de almacenar los datos del tanque en la base de datos

- eliminarTanque(): método encargado de la validación y eliminación de un tanque de la base de datos
- verificarFugas(): método para verificar fugas en una estación de servicio.
- asignarCapacidadAleatoriaDelTanque(): método para asignar aleatoriamente en un rango entre 100 y 200 litros la capacidad del tanque por categoria.
- obtenerCapacidadCategoria(): método para obtener la capacidad de un tanque según la categoría.
- actualizarCapacidadCategoria(): método para actualizar la capacidad de una categoria de un tanque.

2.1.3. Categoria

- obtenerCantidadDeCategorias(): método para obtener cuantas categorias existen.
- obtenerCategorias(): método para obtener los nombres y ids de las categorias.

2.1.4. Cliente

- verificarExistencia(): método para verificar si un cliente existe de acuerdo a la cédula.
- crearCliente(): método para crear un cliente.

2.1.5. Combustible

- guardarCombustible(): método para insertar o actualizar el valor por litro de un combustible según categoria y región.
- obtenerValorLitro(): método para obtener el valor por litro de un combustible segun la estación y la categoria

2.1.6. Nave

- crearNave(): método para crear naves en una estación.
- obtenerNaves(): método para obtener las naves de una estación.
- eliminarNaves(): método para eliminar naves de una estación.

2.1.7. Region

- obtenerRegiones(): método para obtener las regiones.

2.1.8. Surtidor

- obtenerCantidadActivos(): método que se encarga de obtener todos los surtidores que estan activos para una estacion
- obtenerSurtidores(): método para obtener los surtidores de una estación.
- agregarSurtidor(): método para agregar surtidores a una estación.
- eliminarSurtidores(): método para eliminar surtidores de una estación.
- verificarExistencia(): método que verifica si existen surtidores en una estación.

- obnterSurtidorAleatorio(): método para obtener un surtidor aleatorio de una estación
- eliminarSurtidor(): método para eliminar un surtidor de una estación.
- desactivarSurtidor(): método para desactivar un surtidor de una estación.
- activarSurtidor(): método para activar un surtidor de una estación.
- historicoTransacciones(): método para obtener el historico de transaciones de un surtidor de una estación.

2.1.9. Venta

- añadirVenta(): método para añadir una venta.
- eliminarVentas(): método para eliminar ventas.
- calcularVentasPorES(): método para calcular el monto total de las ventas por E/S del pais por categoria
- litrosDeCombustibleVendidosPorCategoria(): método para obtener los litros de combustible vendidos por categoria

3. Diagrama entidad relación

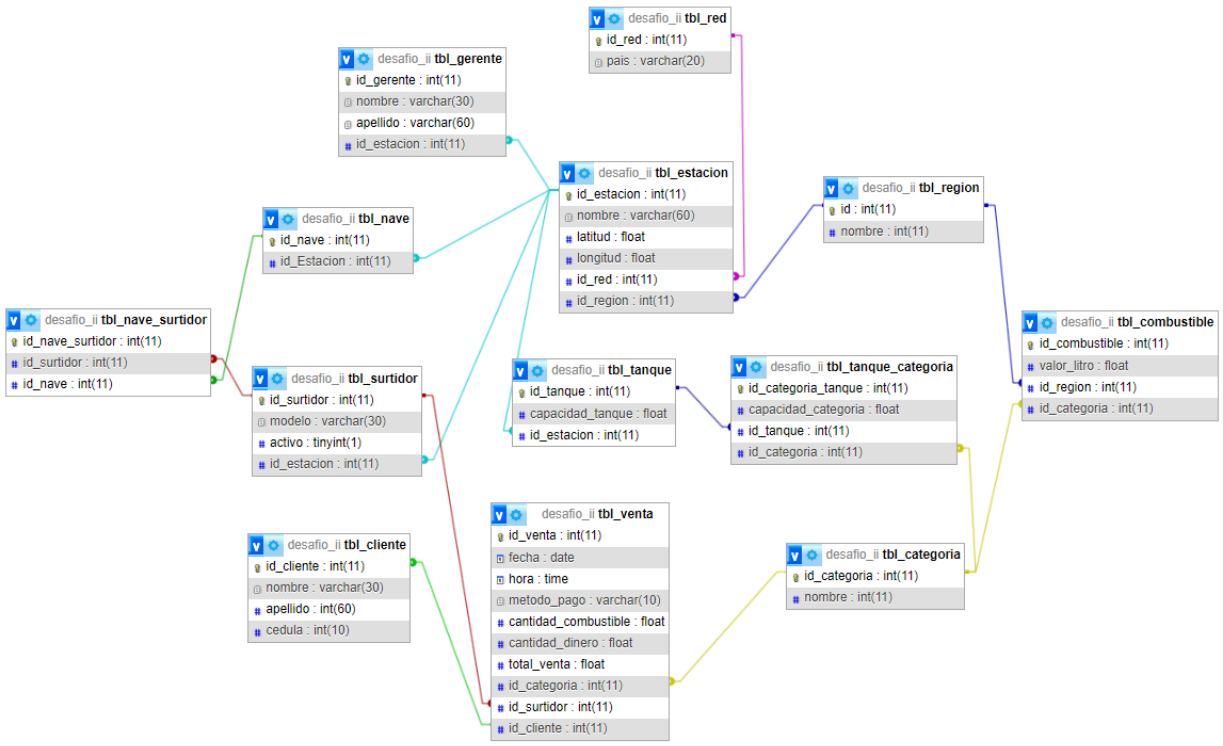

Figura 2: Diagrama entidad relacion (Archivo adjunto en el repositorio)

4. Algoritmos implementados

4.1. Validaciones

```cpp
bool validarCin(){ // validar entrada de datos
    if (cin.fail()){
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Tipo de dato invalido, vuelva a intentar." << endl;
        return false;
    }

    return true;
}

bool validarRango(unsigned short rango1, unsigned short rango2,  unsigned short numero){ // validar un rango de datos

    if (numero < rango1 || numero > rango2){
        cout << "Las opciones permitas son desde " << rango1 << " hasta " << rango2 << ", vuelva a intentar." << endl;
        return false;
    }

    return true;
}

bool validarPositivo(short numero){

    if (numero < 0){
        cout << "No se admiten valores negativos, vuelva a intentar" << endl;
        return false;
    }

    return true;
}
```

Figura 3: Validacion

```cpp
bool validarVarchar(unsigned short longitud, string text){ // sanetizar y validar varchar

    if (text.length() > longitud){
        cout << "La capacidad maxima de caracteres es de " << longitud << ", vuelva a intentar." << endl;
        return false;
    }

    size_t comillaSimple = text.find("'");
    size_t comillasDobles = text.find("\"");

    if (comillaSimple != string::npos){
        cout << "No se permiten comillas simples, vuelve a intentar." << endl;
        return false;
    }

    if (comillasDobles != string::npos){
        cout << "No se permiten comillas dobles, vuelve a intentar." << endl;
        return false;
    }

    return true;
}

bool validarCedula(unsigned int cedula){

    size_t digitos = to_string(cedula).length();

    if (digitos < 8 || digitos > 10){
        cout << "La cedula debe contener de 8 a 10 numeros." << endl;
        return false;
    }

    return true;
}

bool validarNombreYApellido(string nombre){

    regex regexNombre("^[A-Za-zÀ-ÖØ-öø-ÿ'\\s-]+$");

    if (!regex_match(nombre, regexNombre)){
        cout << "El nombre o apellido no es valido" << endl;
        return false;
    }

    return true;
}
```

Figura 4: Validacion

```cpp
template <typename T>
bool validarNumeroEnArreglo(T *&array, T tamañaArray ,T numero){

    for (unsigned int i = 0; i < tamañaArray; i++){
        if (array[i] == numero) return true;
    }

    cout << "Codigo ingresado no existe, vuelva a intentar." << endl;
    cout << "----------------" << endl;

    return false;
}
```

Figura 5: Template de validacion

5. Problemas de desarrollo

Sistema gestor de base de datos: la primera opción para utilizar bases de datos fue MYSQL, pero este mismo no esta nativamente en qt, por lo que la implementación de este era complicada por motivos de instalación de drivers y compatibilidad con la ultima verison de QT.

Abstracción del diagrama de clases: nos encontramos con el problema de no poder interpretar correctamente el diagrama de clases en la realidad.

6. Evolución de la solución

6.1. Modelado UML

Lo primero fue la creación de un **diagrama UML** de clases, lo que permitió visualizar y estructurar las entidades principales del sistema.

6.2. Creación del modelo Entidad-Relación

Posteriormente, se diseñó el **diagrama Entidad-Relación** de la base de datos, lo que permitió estructurar adecuadamente las relaciones entre las entidades y asegurar la integridad de los datos.

6.3. Desarrollo modular

Se comenzó con la implementación de módulos individuales para gestionar cada aspecto del sistema. De esta forma, se pudo aislar las responsabilidades y evitar la duplicación de código.

7. Consideraciónes a tener en cuenta en la implementación

7.1. Persistencia de datos

Se implementó un sistema de almacenamiento basado en una base de datos SQL empleando `QSqlDatabase` para la interacción con la base de datos.

7.2. Validación de entradas de usuario

Se diseñaron varios mecanismos para validar los datos de entrada, garantizando la integridad del sistema.