


PORTADA

Nombre Alumno / DNI	Pablo Nicolás Soto Irago / 02583827F
Título del Programa	Bachelor Degree in Computer Science & Artificial Intelligence
Nº Unidad y Título	UNIT 20 - APPLIED PROGRAMMING & DESIGN PRINCIPLES
Año académico	2024/2025
Profesor de la unidad	Ángel Bravo
Título del Assignment	Assignment Brief
Día de emisión	10/10/2025
Día de entrega	21/01/2025
Nombre IV y fecha	
Declaración del estudiante	<p>Certifico que la presentación del assignment es completamente mi propio trabajo y entiendo completamente las consecuencias del plagio. Entiendo que hacer una declaración falsa es una forma de mala práctica.</p> <p>Fecha: 21/01/2025</p> <p>Firma del alumno:</p> 

Plagio

El plagio es una forma particular de hacer trampa. El plagio debe evitarse a toda costa y los alumnos que infrinjan las reglas, aunque sea inocentemente, pueden ser sancionados. Es su responsabilidad asegurarse de comprender las prácticas de referencia correctas. Como alumno de nivel universitario, se espera que utilice las referencias adecuadas en todo momento y mantenga notas cuidadosamente detalladas de todas sus fuentes de materiales para el material que ha utilizado en su trabajo, incluido cualquier material descargado de Internet. Consulte al profesor de la unidad correspondiente o al tutor del curso si necesita más consejos.



UNIT 20 - APPLIED PROGRAMMING & DESIGN PRINCIPLES

ENTREGA FINAL



21 DE ENERO DE 2025
PABLO NICOLÁS SOTO IRAGO
Computer Science & AI

Contenido

Marco General del Proyecto	4
Principios SOLID.....	4
S - Single Responsibility Principle (SRP)	4
O - Open/Closed Principle (OCP).....	5
L - Liskov Substitution Principle (LSP).....	5
I - Interface Segregation Principle (ISP)	6
D - Dependency Inversion Principle (DIP)	6
Conclusión	7
Explicación de las decisiones de diseño	7
Modularidad	7
Gestión de datos y memoria.....	7
Interacciones entre clases	8
Estructura de datos	8
Patrones de diseño	8
Patrón Singleton.....	8
Patrón Adapter	9
Patrón Observer	10
Patrón de diseño estructural para pacientes y médicos.....	10
Código Limpio	11
Nombres Significativos	11
Funciones Pequeñas y Enfocadas	11
Uso de Constantes y Parámetros.....	11
Separación de Responsabilidades.....	11
Planificación de pruebas	11
Tipos de pruebas.....	11
Identificar casos de prueba.....	13
Crear datos de prueba	13
Escribir casos de prueba.....	13
Revisión mejora.....	14
Automatizar las pruebas	14
Diagrama general del proyecto	15

Código	15
#include.....	15
citaMedica.h	16
Clase CitaMedica	16
Listado de funciones en la clase CitaMedica	17
Funciones y métodos llamados desde otros archivos	22
gestorArchivos.h.....	22
Listado de funciones en la clase GestorArchivos	22
Funciones y métodos llamados desde otros archivos	24
medico.h.....	25
Clase Medico	25
Listado de funciones en la clase Medico	26
Funciones y métodos llamados desde otros archivos	29
paciente.h.....	29
Clase Paciente	29
Listado de funciones en la clase Paciente	30
reporte.h	32
Listado de funciones en reporte.h.....	32
Funciones privadas	33
Funciones y métodos llamados desde otros archivos	34
main.cpp.....	34
Listado de funciones en main.cpp	34
Funciones y métodos llamados desde otros archivos	35
Archivos.txt	37
pacientes.txt	37
medicos.txt.....	37
citas.txt	37
pacientes_cronicos.txt	38
servicios.txt	38
carpeta backup	38
Algoritmos de Búsqueda y Ordenación	38
Búsqueda lineal.....	38

Búsqueda de pacientes por ID en main.cpp	38
Búsqueda de médicos por ID en main.cpp	39
Búsqueda de citas médicas por fecha o urgencia en CitaMedica	39
Posibles optimizaciones	39
Bubble Short	39
Ordenación de citas por fecha en CitaMedica	39
Ordenación de citas por urgencia en CitaMedica	39
Ordenación de citas en el historial clínico (verHistorialClinico).....	39
Posibles optimizaciones	40
Conclusiones	40
Bibliografía	40
Anexos	41

Marco General del Proyecto

El proyecto consiste en desarrollar un software de gestión hospitalaria, permitiendo la administración de información relacionada con pacientes, médicos, citas y otros servicios. Se utilizarán conceptos de programación orientada a objetos (POO) y manejo eficiente de datos a través de algoritmos de búsqueda y ordenación, así como archivos para el almacenamiento persistente.

Principios SOLID

Los principios SOLID son un conjunto de buenas prácticas de diseño orientado a objetos que ayudan a desarrollar software más escalable, mantenible y flexible. Según el acrónimo SOLID, podemos ver los siguientes principios.

S - Single Responsibility Principle (SRP)

El código sigue el principio SRP, ya que cada clase tiene una responsabilidad específica. Por ejemplo, la clase Paciente se encarga de las operaciones relacionadas con los pacientes.

```
class Paciente {  
public:  
    void mostrarPaciente() ;  
    void editarPaciente(std::vector<Paciente*>& pacientes) ;  
    void registrarPaciente() ;  
    static void verPacientePorFecha(const  
std::vector<Paciente*>& pacientes, const std::string&  
fechaIngreso) ;  
    static void verPacientePorID(const  
std::vector<Paciente*>& pacientes) ;  
    static void verPacientePorNombre(const  
std::vector<Paciente*>& pacientes) ;  
    static void eliminarPaciente(std::vector<Paciente*>&  
pacientes) ;  
};
```

Problema

Se puede decir que main.cpp es el archivo que rompe este principio, ya que solamente debería tener una responsabilidad que sería orquestar la ejecución del programa. El main en main.cpp es completamente inservible en otro proyecto por su falta de modularidad y sus dificultades de evolución y mantenimiento.

Otra parte que viola el principio SRP es la parte de setters y getters ya que hacen que una clase tenga más de una responsabilidad. Además, aplicar esta mecánica hace las clases vulnerables, es decir, que, si una clase tiene getters y setters para sus atributos, cualquier otra clase que utilice la clase Paciente podrá modificar los atributos directamente, y en un sistema de gestión, como este de hospitales, no es adecuado.

Solución

La solución más obvia es aplicar el patrón Facade (página 10), y así delegar toda la lógica y funcionamiento del main a una supuesta clase Hospital, para mantener así el main limpio y modular (página 7), cumpliendo el principio SRP.

Para arreglar el problema de los getters y setters, se recomienda eliminarlos y sustituirlos por métodos que encapsulen bien los atributos de la clase, modificando los atributos de manera controlada.

O - Open/Closed Principle (OCP)

Este programa no sigue el principio OCP, ya que las clases deben estar abiertas para extensión, pero cerrada para modificación. No se valoró desde el principio del desarrollo del código y tampoco según se ha avanzado.

Problema

Las clases Medico y Paciente no está diseñadas para ser extendidas sin modificar su código existente. En gestorArchivos.h, Las funciones no están abiertas para extensión, pero cerradas para modificación.

Solución

Para mejorar la clase Medico y Paciente se podría utilizar herencias para permitir la extensión de funcionalidades sin modificar el código existente. Lo mismo se podría aplicar en gestorArchivos.h. De esta forma se mejoraría la modularidad y escalabilidad del sistema.

Si es necesario se podrían reestructurar desde un inicio estas clases, eliminar los getters y setters, y aplicar una nueva serie de métodos que puedan acceder a los atributos sin intervenir en el principio OCP.

L - Liskov Substitution Principle (LSP)

Este principio no se utiliza ni se ha valorado para realizar el sistema de gestión hospitalaria. Principalmente no se puede aplicar un principio relacionado con clases padres e hijas si directamente no se implementan las herencias entre clases.

Problema

No existe una jerarquía de clases en ninguna parte del proyecto. La estructura de cada clase hace imposible aplicar herencias entre ellas.

Solución

Como solución rápida, se podrían crear subclases de médicos (doctor, enfermera, etc.) pero no solucionaría el problema desde la base. Sería recomendable reestructurar las clases desde el principio. Como solución e implementación del principio LSP, se podría crear una clase Persona que tenga de hijas a Medico y Paciente, las cuales compartirán una serie de atributos y funciones, que evitaría consecuentemente la redundancia y mejorando la escalabilidad del programa.

I - Interface Segregation Principle (ISP)

Lo mismo que con LSP va a ocurrir con el principio ISP. Al no aplicarse los principios SOLID desde un principio, se deja de lado la posibilidad de crear clases con herencias, y sin jerarquía de herencias, no puede existir un principio SOLID relacionado con clases padre e hijas.

Problema

Se puede ver que no existen interfaces con métodos que sean aplicables a todas las clases secundarias, ya que no hay clases secundarias.

Solución

Como se sugería antes, es recomendable dividir las interfaces en otras más pequeñas y específicas para manejar más fácilmente las funciones genéricas. Nos interesa también este principio para mejorar la escalabilidad de un sistema desde un principio.

D - Dependency Inversion Principle (DIP)

Por último, el principio DIP tampoco se utiliza en el sistema. El código no depende de abstracciones, y mucho menos puede desacoplar clases directamente sin problemas o sin necesitar cambios.

Problema

Hay archivos como `gestorArchivos.h` que tienen funciones directamente dependientes de clases de bajo nivel como `std::ifstream` y `std::ofstream`. Además, no se utilizan abstracciones en forma de interfaces o clases abstractas.

Mejora

Introducir abstracciones para las operaciones de archivo y hacer que tanto las funciones de alto nivel como las implementaciones de bajo nivel dependan de estas abstracciones.

Conclusión

Más que unos principios SOLID, se han usado principios KISS. Se puede apreciar en la simplicidad de la nomenclatura o en la estructura de datos. Aun buscando una aplicación de los principios KISS, se ha realizado una mala definición de funciones, donde muchas son muy complejas y poco eficientes. Se puede ver que las clases no siguen ningún principio de diseño directamente, ya que no responden a un flujo de control adecuado, todo por su falta de herencias y abstracciones.

Explicación de las decisiones de diseño

Modularidad

El sistema está diseñado modularmente para distribuir responsabilidades de manera clara y eficiente:

- Clases específicas: Entidades como Paciente, Medico y CitaMedica encapsulan sus respectivas lógicas. Cada funcionalidad (registrar, editar, buscar, etc.) se implementa como métodos independientes, facilitando el mantenimiento y la extensibilidad del sistema.
- Archivo main.cpp: Actúa como controlador principal, delegando la lógica y el manejo de datos a funciones y clases específicas. Esto mantiene el archivo limpio y fácil de leer.
- Gestión de archivos: El archivo gestorArchivos.h centraliza las operaciones de entrada y salida (E/S) de datos, permitiendo guardar y recuperar información sin afectar otras áreas del código.

Aún así, el sistema no puede ser modular si su principal problema reside en la estructuración del proyecto desde un inicio. Los principios SOLID indican que el programa no puede llegar a ser modular porque los archivos no tienen la capacidad de ser modificados sin causar problemas en los demás. Aunque cada uno de los archivos tengan sus responsabilidades, no se pueden relacionar con otras clases por su falta de jerarquía de herencias, y porque los métodos que se utilizan para manejar datos entre clases (getters y setters) no permiten centralizar la modificación de atributos específicos desde una sola clase.

Además, tener toda la lógica del programa en un solo archivo .cpp, puede presentar varios problemas en términos de modularidad ya que no hay separación de responsabilidades, tiene una escalabilidad limitada según crece el programa y si este proyecto fuera realizado entre varias personas, sería insostenible mantener así un código.

Gestión de datos y memoria

Uso de vectores dinámicos: Los datos de pacientes, médicos y citas se gestionan mediante vectores de punteros (std::vector<Paciente*>, std::vector<Medico*>,

etc.), lo que permite un manejo eficiente y dinámico de objetos. Este enfoque facilita la adición, eliminación, búsqueda y modificación de elementos sin duplicar información.

Liberación de memoria: El sistema asegura que la memoria dinámica utilizada para los objetos sea liberada al final de la ejecución, evitando fugas de memoria.

Interacciones entre clases

Relaciones centralizadas: Las clases Paciente, Medico y CitaMedica interactúan eficientemente a través de punteros, lo que asegura coherencia al modificar datos. Por ejemplo, las citas están asociadas directamente a pacientes y médicos, evitando duplicar información.

Delegación: La lógica de negocio relacionada con citas médicas se centraliza en funciones específicas, como la validación de fechas y la gestión de estados de citas.

Estructura de datos

Vectores dinámicos: Los datos se almacenan en estructuras eficientes que soportan búsquedas rápidas, ordenamientos y modificaciones, adaptándose a diferentes criterios de filtrado.

Asociación directa: Las relaciones entre pacientes, médicos y citas se mantienen a través de referencias eficientes, mejorando la integridad de los datos.

Patrones de diseño

Los patrones de diseño son soluciones reutilizables a problemas comunes en el diseño de software. Se dividen en tres categorías principales: creacionales, estructurales y de comportamiento.

Patrón Singleton

El patrón Singleton es un patrón creacional que asegura que una clase tenga una única instancia y proporciona un punto de acceso global a ella. Podríamos usar el patrón en la clase GestorArchivos, para asegurarnos de que solo haya una instancia de esta clase que maneje todas las operaciones de archivo.

```
class GestorArchivos {
private:
    static GestorArchivos* instancia;
    GestorArchivos() {} // Constructor privado

public:
    static GestorArchivos* obtenerInstancia() {
        if (instancia == nullptr) {
```

```

        instancia = new GestorArchivos();
    }
    return instancia;
}
void guardarDatosPacientes(const std::vector<Paciente*>& pacientes){}
void recuperarDatosPacientes(std::vector<Paciente*>& pacientes) {}
// Otras funciones...
};

// Inicialización del puntero estático
GestorArchivos* GestorArchivos::instancia = nullptr;

```

Patrón Adapter

Es un patrón estructural que permite que clases con interfaces incompatibles trabajen juntas. Convierte la interfaz de una clase en otra interfaz que el cliente espera.

Una forma de uso entre `medico.h` y `citaMedica.h` es creando una interfaz común (Habitacion por ejemplo) que se implementará a través de un adaptador:

```

#include "medico.h"
#include "habitacion.h"

class MedicoAdapter : public Habitacion {
private:
    Medico* medico;
public:
    MedicoAdapter(Medico* medico) : medico(medico) {}

    std::string obtenerInformacion() const override {
        std::ostringstream info;
        info << "Dr. " << medico->getNombre() << "\n"
            << "ID: " << medico->getID() << "\n"
            << "Servicio: " << medico->getServicio() << "\n"
            << "Disponibilidad: " << (medico->getDisponibilidad() ?
"Disponible" : "No disponible") << "\n";
        return info.str();
    }
};

-----
#include "citaMedica.h"
#include "Habitacion.h"

```

```

class CitaMedicaAdapter : public Habitacion {

private:
    CitaMedica* cita;

public:
    CitaMedicaAdapter(CitaMedica* cita) : cita(cita) {}

    std::string obtenerInformacion() const override {
        std::ostringstream info;
        info << "Fecha: " << cita->getFecha() << "\n"
            << "Paciente: " << cita->getPaciente()->getNombre() << "
(ID: " << cita->getPaciente()->getID() << ")\n"
            << "Medico: " << cita->getMedico()->getNombre() << " (ID: "
<< cita->getMedico()->getID() << ")\n"
            << "Urgencia: " << cita->getUrgencia() << "\n";
        return info.str();
    }
};

```

Patrón Observer

Es un patrón de comportamiento que define una dependencia de uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Primero se debe definir la interfaz del observador y del sujeto (observado). Después se implementaría el patrón Observer en la clase Paciente, y el observador en Reporte. Por último, se integra el observador con el sujeto.

Patrón de diseño estructural para pacientes y médicos

Para manejar la interacción entre pacientes y médicos, un patrón de diseño estructural adecuado sería el Patrón Facade. Este patrón proporciona una interfaz simplificada para un conjunto de interfaces en un subsistema, haciendo que el subsistema sea más fácil de usar. Indirectamente se utiliza este patrón con la división de archivos, en concreto la designación de main.cpp como “fachada” entre el usuario y los servicios del hospital. Se puede ver gráficamente en el diagrama general del proyecto. Digo que se utiliza indirectamente porque main.cpp no tiene ninguna clase, mientras que el patrón indica que los métodos de las clases (en este caso Medico y Paciente), tienen que ser manejados antes por otra clase (que podría ser Hospital). Esta clase Hospital es la que debería ser llamada en main.cpp.

Código Limpio

Nombres Significativos

Los nombres de las funciones y variables son claros y descriptivos, lo que facilita la comprensión del código.

Por ejemplo, `buscarCitasPorFecha` u `ordenarPorUrgencia`.

Funciones Pequeñas y Enfocadas

No todas las funciones del sistema son pequeñas, y aún menos enfocadas, pero si se busca que las funciones secundarias realicen una única tarea específica. Por ejemplo, `buscarCitasPorFecha` solo busca citas por fecha, y `ordenarPorFecha` solo ordena las citas por fecha.

Uso de Constantes y Parámetros

Se utilizan parámetros y constantes en lugar de valores mágicos (valores literales sin explicación), lo que hace que el código sea más fácil de entender y modificar. Por ejemplo, pasar la fecha como parámetro en `buscarCitasPorFecha`.

Separación de Responsabilidades

Las responsabilidades están claramente separadas en diferentes funciones, lo que sigue el principio de responsabilidad única (SRP). Por ejemplo, la búsqueda y la ordenación de citas están separadas en funciones distintas.

Planificación de pruebas

Para planificar las pruebas y asegurar que el sistema maneje correctamente errores como la falta de datos o el ingreso incorrecto de información, es importante seguir un enfoque estructurado. A continuación, un plan detallado para realizar estas pruebas.

Tipos de pruebas

Según que aspecto o apartado se quiera probar, hay varios tipos de pruebas: unitarias, integración y regresión.

Pruebas unitarias

Sirven para probar componentes individuales del sistema de manera aislada. Utilizaremos Google Test para escribir y ejecutar pruebas unitarias:

```
#include <gtest/gtest.h>
#include "Paciente.h"

TEST(PacienteTest, Constructor) {
    Paciente paciente("Juan Perez", 1, "01-01-2020");
}
```

```

    EXPECT_EQ(paciente.getNombre(), "Juan Perez");
    EXPECT_EQ(paciente.getID(), 1);
    EXPECT_EQ(paciente.getFechaIngreso(), "01-01-2020");
}

TEST(PacienteTest, EditarPaciente) {
    Paciente paciente("Juan Perez", 1, "01-01-2020");
    paciente.setNombre("Juan P.");
    EXPECT_EQ(paciente.getNombre(), "Juan P.");
}

```

Pruebas de integración

Estas pruebas se centran en verificar que los diferentes componentes del sistema funcionan correctamente juntos.

```

#include <gtest/gtest.h>
#include "GestorArchivos.h"
#include "Paciente.h"
#include "Medico.h"
#include "CitaMedica.h"

TEST(GestorArchivosTest, GuardarYRecuperarDatosPacientes) {
    std::vector<Paciente*> pacientes;
    pacientes.push_back(new Paciente("Juan Perez", 1, "01-01-2020"));
    pacientes.push_back(new Paciente("Maria Lopez", 2, "02-02-2020"));

    GestorArchivos gestor;
    gestor.guardarDatosPacientes(pacientes);

    std::vector<Paciente*> pacientesRecuperados;
    gestor.recuperarDatosPacientes(pacientesRecuperados);

    ASSERT_EQ(pacientes.size(), pacientesRecuperados.size());
    for (size_t i = 0; i < pacientes.size(); ++i) {
        EXPECT_EQ(pacientes[i]->getNombre(), pacientesRecuperados[i]-
>getNombre());
        EXPECT_EQ(pacientes[i]->getID(), pacientesRecuperados[i]-
>getID());
        EXPECT_EQ(pacientes[i]->getFechaIngreso(),
pacientesRecuperados[i]->getFechaIngreso());
    }
}

```

Pruebas de regresión

Las pruebas de regresión se centran en verificar que los cambios recientes en el código no hayan introducido nuevos errores en el sistema.

```

#include <gtest/gtest.h>

```

```

#include "GestorArchivos.h"
#include "Paciente.h"
#include "Medico.h"
#include "CitaMedica.h"

TEST(GestorArchivosRegresionTest, GuardarYRecuperarDatosPacientes) {
    std::vector<Paciente*> pacientes;
    pacientes.push_back(new Paciente("Juan Perez", 1, "01-01-2020"));
    pacientes.push_back(new Paciente("Maria Lopez", 2, "02-02-2020"));

    GestorArchivos gestor;
    gestor.guardarDatosPacientes(pacientes);

    std::vector<Paciente*> pacientesRecuperados;
    gestor.recuperarDatosPacientes(pacientesRecuperados);

    ASSERT_EQ(pacientes.size(), pacientesRecuperados.size());
    for (size_t i = 0; i < pacientes.size(); ++i) {
        EXPECT_EQ(pacientes[i]->getNombre(), pacientesRecuperados[i]-
>getNombre());
        EXPECT_EQ(pacientes[i]->getID(), pacientesRecuperados[i]-
>getID());
        EXPECT_EQ(pacientes[i]->getFechaIngreso(),
pacientesRecuperados[i]->getFechaIngreso());
    }
}

```

Identificar casos de prueba

- Archivos de datos faltantes.
- Archivos de datos vacíos.
- Datos mal formateados en los archivos.
- Datos incompletos.
- Datos duplicados.
- Datos con tipos incorrectos (por ejemplo, std::string en lugar de int).

Crear datos de prueba

- Un archivo pacientes.txt vacío.
- Un archivo medicos.txt con datos mal formateados.
- Un archivo citas.txt con datos incompletos.

Escribir casos de prueba

Hay que escribir casos de prueba específicos para cada escenario identificado. Se debería utilizar un marco de pruebas adecuado como Google Test para C++ para la automatización de pruebas de software con ayuda de CMake. Ejemplos:

Si falta un archivo:

```
#include <gtest/gtest.h>
#include "gestorArchivos.h"

TEST(GestorArchivosTest, ArchivoPacientesFaltante) {
    std::vector<Paciente*> pacientes;
    GestorArchivos gestor;
    gestor.recuperarDatosPacientes(pacientes);
    EXPECT_TRUE(pacientes.empty());
}
```

Si un archivo está vacío:

```
TEST(GestorArchivosTest, ArchivoPacientesVacio) {
    std::ofstream archivo("pacientes.txt");
    archivo.close();

    std::vector<Paciente*> pacientes;
    GestorArchivos gestor;
    gestor.recuperarDatosPacientes(pacientes);
    EXPECT_TRUE(pacientes.empty());
}
```

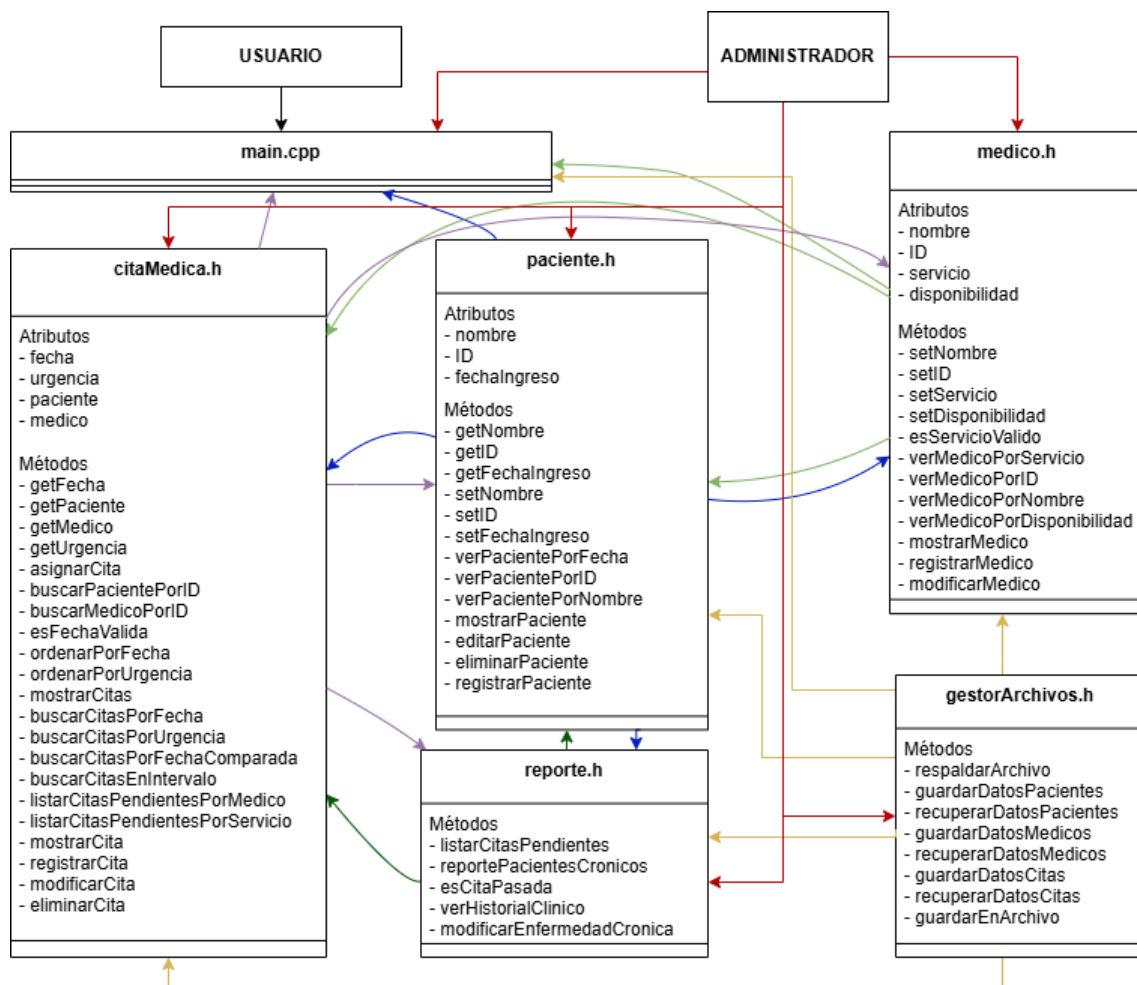
Revisión mejora

Después de ejecutar las pruebas, hay que revisar los resultados de las pruebas y mejorar el código según sea necesario. Es importante que el sistema maneje todos los errores y que los mensajes de error sean claros para el usuario.

Automatizar las pruebas

Como implementación futura interesa automatizar las pruebas para que se ejecuten regularmente como parte de un proceso de integración continua (CI). Esto asegurará que cualquier cambio no introduzca errores antiguos.

Diagrama general del proyecto



Código

A continuación, se presenta la descripción del código completo del AB.

#include

<iostream>: Para operaciones de entrada y salida, como `std::cin` y `std::cout`.

<vector>: Permite el uso de contenedores dinámicos, como `std::vector`, para manejar colecciones de objetos (Pacientes, Médicos, Citas, etc.).

<string>: Proporciona la clase `std::string` para manipular cadenas de texto.

<fstream>: Para operaciones de lectura y escritura de archivos (GestorArchivos).

<ctime>: Utilizado para obtener y manipular fechas y horas actuales (por ejemplo, en Reporte para manejar fechas de citas).

<limits>: Ofrece valores límite para tipos de datos, usado para validar entradas numéricas y manejar errores de entrada.

<set>: Proporciona un contenedor para almacenar elementos únicos, usado para conjuntos ordenados y eliminar duplicados.

<algorithm>: Incluye funciones estándar para manipulación de datos, como ordenamiento, búsqueda y transformaciones en contenedores.

"citaMedica.h": Incluye la definición de la clase CitaMedica, que gestiona la creación, modificación y eliminación de citas médicas.

"paciente.h": Contiene la definición de la clase Paciente, utilizada para manejar información de los pacientes, como nombre, ID y enfermedades.

"medico.h": Define la clase Medico, que gestiona información sobre los médicos, como servicios ofrecidos, ID y disponibilidad.

"gestorArchivos.h": Proporciona funcionalidades para la lectura y escritura de datos en archivos, como pacientes, médicos y citas médicas.

"reporte.h": Incluye la definición de la clase Reporte, que se encarga de generar informes, como historial clínico y pacientes crónicos.

citaMedica.h

El archivo citaMedica.h se centra en definir la clase CitaMedica, que maneja la información de las citas médicas. La clase proporciona métodos para acceder a los datos de las citas y así realizar operaciones relacionadas con las ellas.

Clase CitaMedica

Atributos

- Fecha (string): La fecha de la cita médica.
- Urgencia (int): Nivel de urgencia de la cita médica.
- Paciente (puntero a Paciente): Puntero al objeto Paciente asociado.
- Medico (puntero a Medico): Puntero al objeto Medico asociado.

Getters

getFecha()

Tipo de retorno: std::string

Descripción: Retorna la fecha de la cita médica almacenada en el atributo fecha.

getPaciente()

Tipo de retorno: Paciente*

Descripción: Retorna un puntero al objeto Paciente asociado a la cita médica.

getMedico()

Tipo de retorno: Medico*

Descripción: Retorna un puntero al objeto Medico asociado a la cita médica.

getUrgencia()

Tipo de retorno: int

Descripción: Retorna el nivel de urgencia de la cita médica, representado por un valor entero.

Listado de funciones en la clase CitaMedica

A continuación, se enumeran las funciones definidas en citaMedica.h, específicamente dentro de la clase CitaMedica{

asignarCita

void asignarCita()

Propósito: Guarda una cita médica en el archivo citas.txt con todos sus detalles, como la fecha, el paciente, el médico y la urgencia.

Descripción:

Abre el archivo citas.txt en modo de adición (std::ios::app) para que las nuevas citas se añadan al final del archivo sin modificar los datos existentes.

Escribe la información de la cita: la fecha, el nombre del paciente y médico, el ID del paciente y médico, y el nivel de urgencia.

buscarPacientePorID

static Paciente* buscarPacientePorID

(const std::vector<Paciente*>& pacientes, int id)

Propósito: Buscar un paciente en el sistema mediante su ID.

Descripción:

Recorre el vector de punteros a pacientes (std::vector<Paciente*>) y compara cada ID con el ID proporcionado. Si encuentra un paciente cuyo ID coincide con el proporcionado, devuelve un puntero a ese paciente.

buscarMedicoPorID

static Medico* buscarMedicoPorID

(const std::vector<Medico*>& medicos, int id)

Propósito: Buscar un médico en el sistema mediante su ID.

Descripción:

Recorre el vector de punteros a médicos (`std::vector<Medico*>`) y compara cada ID con el ID proporcionado. Si encuentra un médico cuyo ID coincide con el proporcionado, devuelve un puntero a ese médico.

esFechaValida

```
static bool esFechaValida(const std::string& fecha)
```

Propósito: Validar que una fecha esté en el formato pedido (DD-MM-AAAA) y que sea una válida (con respecto al sistema).

Descripción:

Comprueba que la fecha tenga el formato DD-MM-AAAA verificando los caracteres en las posiciones correctas.

ordenarPorFecha

```
static void ordenarPorFecha(  
std::vector<CitaMedica*>& citas)
```

Propósito: Ordenar las citas médicas por fecha.

Descripción:

Esta función ordena el vector de citas (`std::vector<CitaMedica*>`) en orden ascendente según la fecha. La ordenación permite que las citas se presenten en el orden cronológico. Utiliza una función de comparación que verifica las fechas de las citas para establecer el orden.

ordenarPorUrgencia

```
static void ordenarPorUrgencia  
(std::vector<CitaMedica*>& citas)
```

Propósito: Ordenar las citas médicas por urgencia.

Descripción:

Esta función organiza las citas en el vector (`std::vector<CitaMedica*>`) según el nivel de urgencia, de menor a mayor. Utiliza `std::sort` junto con una función de comparación basada en el valor de urgencia de cada cita.

mostrarCitas

```
static void mostrarCitas  
(const std::vector<CitaMedica*>& citas)
```

Propósito: Mostrar todas las citas médicas.

Descripción:

Imprime por terminal una lista de todas las citas contenidas en el vector citas. Esto proporciona una visión general de todas las citas en el sistema, facilitando la inspección o verificación manual de las citas.

buscarCitasPorFecha

```
static void buscarCitasPorFecha(const
std::vector<CitaMedica*>& citas, const std::string& fecha)
```

Propósito: Buscar citas médicas por una fecha específica.

Descripción:

Filtra y muestra todas las citas que ocurren en una fecha específica dada por el parámetro fecha. Compara la fecha de cada cita con la fecha introducida, y muestra las coincidencias. Este método permite a los usuarios consultar las citas que se han programado para un día específico.

buscarCitasPorUrgencia

```
static void buscarCitasPorUrgencia(const
std::vector<CitaMedica*>& citas, int urgencia)
```

Propósito: Buscar citas médicas según su nivel de urgencia.

Descripción:

Filtra las citas del sistema y muestra aquellas que tienen el nivel de urgencia especificado. El parámetro urgencia debe ser un número entero que representa el nivel de urgencia, y la función mostrará solo aquellas citas que coincidan con ese valor.

buscarCitasPorFechaComparada

```
static void buscarCitasPorFechaComparada(const
std::vector<CitaMedica*>& citas, const std::string& fecha)
```

Propósito: Buscar citas pasadas o futuras comparadas con una fecha dada.

Descripción:

Permite filtrar y mostrar las citas que ocurren antes o después de una fecha. El parámetro fecha es comparado con la fecha de cada cita y se muestran aquellas que ocurren antes o después de esa fecha.

buscarCitasEnIntervalo

```
static std::vector<CitaMedica*>
buscarCitasEnIntervalo(const std::vector<CitaMedica*>&
citas, const std::string& fechaInicio, const std::string&
fechaFin)
```

Propósito: Buscar citas médicas dentro de un intervalo de fechas.

Descripción:

Filtra las citas en el sistema que caen dentro de un rango de fechas especificado por fechaInicio y fechaFin. La función compara las fechas de las citas con el intervalo proporcionado y devuelve un nuevo vector con las citas que ocurren dentro de ese intervalo. Esto facilita la consulta de citas programadas dentro de un período determinado.

listarCitasPendientesPorMedico

```
static void listarCitasPendientesPorMedico(const
std::vector<CitaMedica*>& citas, Medico* medico)
```

Propósito: Lista las citas pendientes para un médico específico.

Descripción:

Filtra y muestra todas las citas que están asignadas a un médico determinado y que aún no han sido atendidas. El parámetro medico es un puntero a un objeto de la clase Medico, y la función muestra las citas que están pendientes para ese médico.

listarCitasPendientesPorServicio

```
static void listarCitasPendientesPorServicio(const
std::vector<CitaMedica*>& citas, const std::string&
servicio)
```

Propósito: Lista las citas pendientes por un servicio específico.

Descripción:

Muestra todas las citas asignadas a un servicio específico que aún no han sido atendidas.

El parámetro servicio es una cadena de texto que representa el nombre del servicio (por ejemplo, "Cardiología"), y la función filtra y muestra las citas pendientes para ese servicio en particular.

mostrarCita

```
void mostrarCita() const
```

Propósito: Mostrar los detalles de una cita médica.

Descripción:

Imprime en pantalla la información completa de la cita médica en un formato legible, incluyendo el servicio del médico, la fecha de la cita, el nombre e ID del paciente, el nombre e ID del médico, y el nivel de urgencia.

La función no modifica el estado del objeto, solo muestra sus atributos, y se marca como const para garantizar que no altere el objeto.

registrarCita

```
static void registrarCita(std::vector<CitaMedica*>& citas,  
    Paciente* paciente, Medico* medico, const std::string&  
        fecha, int urgencia)
```

Propósito: Registra una nueva cita médica en el sistema.

Descripción:

Crea y agrega una nueva cita médica al sistema, utilizando los detalles proporcionados sobre el paciente, médico, fecha y nivel de urgencia. La función agrega la nueva cita al vector citas, permitiendo que se mantenga un registro de todas las citas en el sistema. Después de registrar la cita, se puede acceder a ella mediante las funciones de búsqueda, visualización y modificación.

modificarCita

```
static void modificarCita(std::vector<Paciente*>& pacientes,  
    std::vector<Medico*>& medicos, std::vector<CitaMedica*>& citas)
```

Propósito: Permite modificar los detalles de una cita médica existente.

Descripción:

Solicita al usuario la fecha de la cita que desea modificar. Después de validar la fecha proporcionada, la función busca la cita correspondiente. Permite al usuario modificar varios aspectos de la cita, como el paciente asignado, el médico, la urgencia y la fecha. Una vez que se validan todos los cambios, la cita es actualizada con los nuevos datos.

eliminarCita

```
static void eliminarCita(std::vector<CitaMedica*>& citas)
```

Propósito: Elimina una cita médica del sistema.

Descripción:

Solicita al usuario la fecha de la cita que desea eliminar. Muestra todas las citas correspondientes a esa fecha. Permite seleccionar una cita para eliminar basándose en el ID del paciente. Solicita confirmación antes de proceder con la eliminación de la cita seleccionada. Una vez confirmada la eliminación, la cita es eliminada del sistema.

Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las funciones y métodos llamados desde otros archivos en citaMedica.h, organizados según su origen. Todos se encuentran dentro de la clase CitaMedica{}

Paciente

Paciente::verPacientePorID: Muestra la información de un paciente según su ID.

Paciente::eliminarPaciente: Elimina un paciente del sistema.

Medico

Medico::verMedicoPorID: Muestra la información de un médico según su ID.

Medico::eliminarMedico: Elimina un médico del sistema.

gestorArchivos.h

El archivo gestorArchivos.h define la clase GestorArchivos, encargada de gestionar la lectura y escritura de datos relacionados con pacientes, médicos y citas médicas. Permite cargar y guardar información desde y hacia archivos, facilitando la persistencia de los datos en el sistema.

Listado de funciones en la clase GestorArchivos

respaldarArchivo

```
void respaldarArchivo(const std::string& archivoOriginal)
```

Propósito: Realiza un respaldo de un archivo específico.

Descripción:

La función crea una copia del archivo original en un directorio de respaldo (backup/), generando un nombre de archivo único con la fecha y hora actuales. Si

el directorio de respaldo no existe, lo crea. Se utiliza la librería `std::filesystem` para la manipulación de archivos y directorios.

guardarDatosPacientes

```
void guardarDatosPacientes  
(const std::vector<Paciente*>& pacientes)
```

Propósito: Guarda los datos de los pacientes en un archivo de texto (`pacientes.txt`).

Descripción:

La función guarda la información de cada paciente (nombre, ID, fecha de ingreso) en un archivo de texto. Además, realiza un respaldo del archivo antes de escribir los nuevos datos.

recuperarDatosPacientes

```
void recuperarDatosPacientes (std::vector<Paciente*>&  
pacientes)
```

Propósito: Recupera los datos de los pacientes desde un archivo de texto (`pacientes.txt`).

Descripción:

Lee el archivo `pacientes.txt` y extrae los datos de los pacientes (nombre, ID, fecha de ingreso). Los pacientes recuperados se almacenan en el vector proporcionado.

guardarDatosMedicos

```
void guardarDatosMedicos (const std::vector<Medico*>&  
medicos)
```

Propósito: Guarda los datos de los médicos en un archivo de texto (`medicos.txt`).

Descripción:

Similar a la función `guardarDatosPacientes`, pero para médicos. Guarda el nombre, ID, servicio y disponibilidad de cada médico en el archivo `medicos.txt`.

recuperarDatosMedicos

```
void recuperarDatosMedicos (std::vector<Medico*>& medicos)
```

Propósito: Recupera los datos de los médicos desde un archivo de texto (`medicos.txt`).

Descripción:

Lee el archivo `medicos.txt` y extrae los datos de los médicos (nombre, ID, servicio, disponibilidad). Los médicos recuperados se almacenan en el vector proporcionado.

guardarDatosCitas

```
void guardarDatosCitas(const std::vector<CitaMedica*>&  
                        citas)
```

Propósito: Guarda los datos de las citas médicas en un archivo de texto (citas.txt).

Descripción:

Guarda la fecha, paciente, médico y urgencia de cada cita médica en el archivo citas.txt.

recuperarDatosCitas

```
void recuperarDatosCitas(std::vector<CitaMedica*>& citas,  
                        const std::vector<Paciente*>& pacientes, const  
                        std::vector<Medico*>& medicos)
```

Propósito: Recupera los datos de las citas médicas desde un archivo de texto (citas.txt).

Descripción:

Lee el archivo citas.txt, extrae la información de cada cita (fecha, ID del paciente, ID del médico y urgencia), y busca los pacientes y médicos correspondientes a través de sus ID. Luego, crea las citas médicas y las agrega al vector proporcionado.

guardarEnArchivo

```
void guardarEnArchivo(const std::string& nombrePaciente,  
                      bool enfermedadCronica)
```

Propósito: Guardar la información de un paciente con enfermedad crónica en un archivo.

Descripción:

Esta función primero respalda el archivo de pacientes crónicos, luego abre un archivo para guardar el nombre del paciente y su estado de enfermedad crónica (sí o no). Si se puede abrir, escribe la información del paciente en el archivo y lo cierra.

Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo gestorArchivos.h, organizados según su origen.

Paciente.h

Paciente::buscarPacientePorID: Para buscar un paciente en el sistema mediante su ID.

medico.h

Medico::buscarMedicoPorID: Para buscar un médico en el sistema mediante su ID.

medico.h

El archivo `medico.h` define la clase `Medico`, que gestiona la información relacionada con los médicos en el sistema. Proporciona métodos para registrar, editar, eliminar y consultar los datos de los médicos, como su nombre, ID, servicio y disponibilidad. Esta clase facilita la organización y manejo de los médicos en el sistema, permitiendo asociar a cada médico un servicio o especialidad, y verificar su disponibilidad para atender a los pacientes.

Clase Medico

Atributos

- `nombre (std::string)`: El nombre del médico.
- `ID (int)`: El ID único del médico.
- `servicio (std::string)`: El servicio al que pertenece el médico.
- `disponibilidad (bool)`: Indica si el médico está disponible.

Getters

`getNombre()`

Tipo de retorno: `std::string`

Descripción: Retorna el nombre del médico almacenado en el atributo `nombre`.

`getID()`

Tipo de retorno: `int`

Descripción: Retorna el identificador único del médico almacenado en el atributo `ID`.

`getServicio()`

Tipo de retorno: `std::string`

Descripción: Retorna el servicio o especialidad del médico almacenado en el atributo `servicio`.

`getDisponibilidad()`

Tipo de retorno: `bool`

Descripción: Retorna el estado de disponibilidad del médico almacenado en el atributo `disponibilidad`.

Listado de funciones en la clase Medico

Constructor Medico

```
Medico(std::string& nombre, int ID, const std::string&  
servicio, bool disponibilidad)
```

Propósito: Inicializa un objeto Medico con los valores proporcionados.

Descripción:

Este constructor permite crear un objeto Medico a partir de un nombre, un ID, un servicio y la disponibilidad del médico. Utiliza estos parámetros para configurar las propiedades del médico al momento de su creación.

setNombre

```
void setNombre(const std::string& nombre)
```

Propósito: Establece el nombre del médico.

Descripción:

Este método establece el valor del atributo nombre del médico.

setID

```
void setID(const int& ID)
```

Propósito: Establece el ID del médico.

Descripción:

Este método establece el valor del atributo ID del médico.

setServicio

```
void setServicio(const std::string& servicio)
```

Propósito: Establece el servicio del médico si es válido.

Descripción:

Este método asigna un servicio al médico, pero antes verifica que el servicio esté presente en el archivo servicios.txt. Si el servicio no es válido, se muestra un mensaje de error.

setDisponibilidad

```
void setDisponibilidad(const bool& disponibilidad)
```

Propósito: Establece la disponibilidad del médico.

Descripción:

Este método asigna un valor booleano al atributo disponibilidad, indicando si el médico está disponible o no.

esServicioValido

```
bool esServicioValido(const std::string& servicio)
```

Propósito: Verifica si un servicio es válido.

Descripción:

Esta función abre el archivo servicios.txt y verifica si el servicio proporcionado se encuentra en el archivo, ignorando las diferencias entre mayúsculas y minúsculas.

verMedicoPorServicio

```
static void verMedicoPorServicio  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los médicos disponibles en un servicio específico.

Descripción:

La función solicita al usuario que ingrese un servicio, valida que sea correcto y luego muestra todos los médicos que pertenecen a ese servicio.

verMedicoPorID

```
static void verMedicoPorID  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los detalles de un médico basado en su ID.

Descripción:

Solicita al usuario un ID, valida que sea un número y busca el médico correspondiente. Si se encuentra, muestra su información.

verMedicoPorNombre

```
static void verMedicoPorNombre  
(const std::vector<Medico*>& medicos)
```

Propósito: Muestra los detalles de un médico basado en su nombre.

Descripción:

Permite al usuario buscar médicos por nombre, y muestra sus detalles si se encuentra alguno que coincida.

verMedicoPorDisponibilidad

```
static void verMedicoPorDisponibilidad
```

(const std::vector<Medico*>& medicos)

Propósito: Muestra los médicos según su disponibilidad.

Descripción:

Solicita al usuario la disponibilidad del médico (1 para disponible, 0 para no disponible), y muestra los médicos que coinciden con esa disponibilidad.

mostrarMedico

void mostrarMedico() const

Propósito: Muestra los detalles del médico.

Descripción:

Imprime la información completa del médico, incluyendo su nombre, ID, servicio y disponibilidad, formateada (con setw y setfill) para su correcta presentación en la consola.

registrarMedico

void registrarMedico()

Propósito: Permite registrar un nuevo médico en el sistema.

Descripción:

Solicita al usuario los datos del médico, realiza validaciones y luego lo agrega al sistema. Este método incluye la verificación de la validez del servicio y la disponibilidad.

modificarMedico

void modificarMedico(std::vector<Medico*>& medicos)

Propósito: Permite modificar los datos de un médico existente.

Descripción:

Solicita el ID del médico, busca el médico en la lista y permite modificar sus datos, incluyendo el nombre, el ID, el servicio y la disponibilidad.

eliminarMedico

static void eliminarMedico(std::vector<Medico*>& medicos)

Propósito: Elimina un médico del sistema según su ID.

Descripción:

Solicita el ID del médico a eliminar, verifica si existe y lo elimina de la lista de médicos. Además, pide confirmación antes de realizar la eliminación.

Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo `medico.h`, organizados según su origen.

Paciente.h

Paciente::buscarPacientePorID

Utilizado para buscar un paciente en el sistema mediante su ID.

citaMedica.h

CitaMedica::buscarCitaPorID

Usado para asociar un médico a una cita médica en el sistema.

`paciente.h`

El archivo `paciente.h` define la clase `Paciente`, que gestiona la información de los pacientes en el sistema. Proporciona métodos para registrar, editar, eliminar y consultar los datos de los pacientes, como su nombre e ID. Además, permite manejar aspectos relacionados con las citas médicas y enfermedades crónicas asociadas a cada paciente.

Clase Paciente

Atributos

- `nombre (std::string)`: El nombre completo del paciente.
- `ID (int)`: Identificador único del paciente.
- `fechaIngreso (std::string)`: Fecha de ingreso del paciente al sistema en formato dd-MM-AAAA.

Getters

`getNombre()`

Tipo de retorno: `std::string`

Descripción: Retorna el nombre del paciente almacenado en el atributo `nombre`.

`getID()`

Tipo de retorno: `int`

Descripción: Retorna el identificador único del paciente almacenado en el atributo `ID`.

`getFechaIngreso()`

Tipo de retorno: `std::string`

Descripción: Retorna la fecha de ingreso del paciente almacenada en el atributo `fechaIngreso`.

Listado de funciones en la clase Paciente

esFechaValida

```
static bool esFechaValida(const std::string& fecha)
```

Propósito: Validar que una fecha esté en el formato correcto DD-MM-AAAA y que sea una fecha válida.

Descripción:

Comprueba que la fecha tenga el formato DD-MM-AAAA verificando los caracteres en las posiciones correctas.

Constructor Paciente

```
Paciente(const std::string& nombre, int ID, const  
std::string& fechaIngreso)
```

Propósito: Inicializa un objeto Paciente con los valores proporcionados.

Descripción:

Este constructor asigna un nombre, un ID y una fecha de ingreso al paciente al momento de su creación.

setNombre

```
void setNombre(const std::string& nombre)
```

Propósito: Establece el nombre del paciente.

Descripción:

Este método asigna un valor al atributo Nombre del paciente.

setID

```
void setID(const int& ID)
```

Propósito: Establece el ID del paciente.

Descripción:

Este método asigna un valor al atributo ID del paciente, asegurando que sea único.

setFechaIngreso

```
void setFechaIngreso(const std::string& fechaIngreso)
```

Propósito: Establece la fecha de ingreso del paciente si es válida.

Descripción:

Este método valida y asigna una fecha al atributo FechaIngreso, asegurándose de que siga el formato dd-MM-AAAA.

verPacientePorFecha

```
static void verPacientePorFecha(const
std::vector<Paciente*>& pacientes, const std::string&
fechaIngreso)
```

Propósito: Muestra pacientes ingresados en una fecha específica.

Descripción:

Solicita una fecha y lista los pacientes cuyo atributo FechaIngreso coincida con la fecha proporcionada.

verPacientePorID

```
static void verPacientePorID
(const std::vector<Paciente*>& pacientes)
```

Propósito: Busca y muestra los detalles de un paciente según su ID.

Descripción:

Solicita un ID, valida la entrada y muestra la información del paciente si se encuentra en la lista.

verPacientePorNombre

```
static void verPacientePorNombre(const std::vector<Paciente*>&
pacientes)
```

Propósito: Muestra pacientes basándose en su nombre.

Descripción:

Solicita un nombre al usuario, busca coincidencias y muestra los detalles de los pacientes encontrados.

mostrarPaciente

```
void mostrarPaciente() const
```

Propósito: Muestra los detalles completos del paciente.

Descripción:

Imprime la información del paciente en la terminal, incluyendo nombre, ID y fecha de ingreso.

editarPaciente

```
void editarPaciente(std::vector<Paciente*>& pacientes)
```

Propósito: Permite modificar los datos de un paciente existente.

Descripción:

Solicita el ID de un paciente, busca el registro en el sistema y permite editar su información.

eliminarPaciente

```
static void eliminarPaciente(std::vector<Paciente*>&
                             pacientes)
```

Propósito: Elimina un paciente del sistema.

Descripción:

Solicita el ID del paciente, verifica su existencia y elimina el registro correspondiente.

registrarPaciente

```
void registrarPaciente()
```

Propósito: Permite registrar un nuevo paciente en el sistema.

Descripción:

Solicita al usuario los datos del paciente, valida las entradas y los agrega al sistema.

reporte.h

El archivo reporte.h define la clase Reporte, que centraliza funciones relacionadas con la generación de reportes, consulta de historial clínico y manejo de enfermedades crónicas.

Listado de funciones en reporte.h

listarCitasPendientes

```
void listarCitasPendientes(const std::vector<CitaMedica*>&
                           citas, const std::string& criterio)
```

Propósito: Mostrar las citas pendientes agrupadas por médico o servicio.

Descripción:

Recorre un vector de citas médicas y muestra información de las citas pendientes según el criterio especificado ("medico" o "servicio").

reportePacientesCronicos

```
void reportePacientesCronicos
(const std::vector<Paciente*>& pacientes)
```

Propósito: Generar un reporte de pacientes con enfermedades crónicas.

Descripción:

Recorre el vector de pacientes para identificar aquellos con enfermedades crónicas y muestra sus nombres e identificadores.

esCitaPasada

```
bool esCitaPasada(const std::string& fechaCita, int year,
                  int month, int day)
```

Propósito: Verificar si una cita pertenece al pasado.

Descripción:

Compara la fecha de una cita con la fecha actual y devuelve true si la cita ya ocurrió.

verHistorialClinico

```
void verHistorialClinico(std::vector<CitaMedica*>& citas)
```

Propósito: Consultar el historial clínico de un paciente o de todos los pacientes.

Descripción:

Ordena las citas por fecha y muestra aquellas pasadas, filtrando por el identificador del paciente o mostrando todas si no se especifica un ID.

modificarEnfermedadCronica

```
void modificarEnfermedadCronica
(
    const std::vector<Paciente*>& pacientes
)
```

Propósito: Registrar o modificar el estado de enfermedad crónica de un paciente.

Descripción:

Permite buscar un paciente por nombre y actualizar su estado de enfermedad crónica, mostrando un mensaje de confirmación.

Funciones privadas

tieneEnfermedadCronica

```
bool tieneEnfermedadCronica(Paciente* paciente)
```

Propósito: Determinar si un paciente tiene una enfermedad crónica.

Descripción:

Devuelve un valor booleano indicando si el paciente posee enfermedades crónicas. Esta implementada como una función auxiliar dentro de la clase.

Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo `reporte.h`, organizados según su origen.

Archivo `citaMedica.h`

CitaMedica::getFecha: Obtiene la fecha de una cita médica.

CitaMedica::getPaciente:

Devuelve un puntero al paciente asociado a la cita.

CitaMedica::getMedico: Devuelve un puntero al médico asociado a la cita.

Archivo `paciente.h`

Paciente::getNombre: Obtiene el nombre del paciente.

Paciente::getID: Devuelve el identificador único del paciente.

Archivo `medico.h`

Medico::getNombre: Devuelve el nombre del médico.

Medico::getServicio: Obtiene el servicio asociado al médico.

main.cpp

El archivo `main.cpp` actúa como el controlador principal del programa, donde se gestionan las interacciones del usuario y la ejecución de las funcionalidades del sistema. Contiene el flujo principal de ejecución, incluyendo la carga de datos, la visualización de menús, y la captura de entradas del usuario. Además, delega las tareas específicas a otras clases como `CitaMedica`, `Paciente`, `Medico` y `GestorArchivos`, manteniendo el código organizado y modular.

Listado de funciones en main.cpp

A continuación, se enumeran las funciones definidas en `main.cpp`.

leerEntero

```
int leerEntero (const std::string& mensaje)
```

Propósito: Solicitar al usuario que ingrese un número entero, validando la entrada.

Descripción:

Muestra un mensaje al usuario. Valida que la entrada sea un número entero y solicita reintentos en caso de entrada inválida.

buscarPacientePorID

```
Paciente* buscarPacientePorID(  
    const std::vector<Paciente*>& pacientes, int id)
```

Propósito: Buscar un paciente en el vector de pacientes a partir de su ID.

Descripción:

Recorre el vector de punteros a pacientes (`std::vector<Paciente*>`) y compara cada ID con el ID proporcionado. Si encuentra un paciente cuyo ID coincide con el proporcionado, devuelve un puntero a ese paciente.

buscarMedicoPorID

Medico* buscarMedicoPorID

(const std::vector<Medico*>& medicos, int id)

Propósito: Buscar un médico en el vector de médicos a partir de su ID.

Descripción:

Recorre el vector de punteros a médicos (`std::vector<Medico*>`) y compara cada ID con el ID proporcionado. Si encuentra un médico cuyo ID coincide con el proporcionado, devuelve un puntero a ese médico.

Funciones y métodos llamados desde otros archivos

A continuación, se enumeran las Funciones y métodos llamados desde otros archivos en el archivo `main.cpp`, organizados según su origen.

citaMedica.h

CitaMedica::buscarCitasEnIntervalo:

Busca citas médicas en un intervalo de fechas.

CitaMedica::modificarCita: Permite editar una cita médica.

CitaMedica::eliminarCita: Elimina una cita médica del sistema.

CitaMedica::ordenarPorFecha: Ordena las citas por fecha.

CitaMedica::ordenarPorUrgencia: Ordena las citas por urgencia.

CitaMedica::mostrarCitas: Muestra una lista de citas médicas.

CitaMedica::listarCitasPendientesPorMedico: Lista citas pendientes para un médico específico.

CitaMedica::listarCitasPendientesPorServicio: Lista citas pendientes por servicio.

CitaMedica::buscarCitasPorFecha: Busca citas médicas por fecha específica.

CitaMedica::buscarCitasPorUrgencia: Busca citas médicas según su urgencia.

CitaMedica::buscarCitasPorFechaComparada: Busca citas pasadas o futuras según una fecha.

CitaMedica::registrarCita: Registra una nueva cita médica.

Archivo gestorArchivos.h

GestorArchivos::recuperarDatosPacientes: Carga los datos de los pacientes desde un archivo.

GestorArchivos::recuperarDatosMedicos: Carga los datos de los médicos desde un archivo.

GestorArchivos::recuperarDatosCitas: Carga los datos de las citas médicas desde un archivo.

GestorArchivos::guardarDatosPacientes: Guarda los datos de los pacientes en un archivo.

GestorArchivos::guardarDatosMedicos: Guarda los datos de los médicos en un archivo.

GestorArchivos::guardarDatosCitas: Guarda los datos de las citas médicas en un archivo.

GestorArchivos::guardarEnArchivo: Guarda información adicional, como el estado de enfermedades crónicas.

Archivo paciente.h

Paciente::verPacientePorFecha: Muestra pacientes registrados en una fecha específica.

Paciente::verPacientePorID: Muestra información de un paciente basado en su ID.

Paciente::verPacientePorNombre: Busca pacientes por nombre.

Paciente::eliminarPaciente: Elimina un paciente del sistema.

Paciente::registrarPaciente: Registra un nuevo paciente.

Paciente::editarPaciente: Permite modificar la información de un paciente.

Archivo medico.h

Medico::verMedicoPorServicio: Lista médicos que ofrecen un servicio específico.

Medico::verMedicoPorID: Busca médicos por ID.

Medico::verMedicoPorNombre: Busca médicos por nombre.

Medico::verMedicoPorDisponibilidad: Lista médicos disponibles en un horario.

Medico::eliminarMedico: Elimina un médico del sistema.

Medico::registrarMedico: Registra un nuevo médico.

Medico::modificarMedico: Permite editar la información de un médico.

Archivo reporte.h

Reporte::verHistorialClinico: Muestra el historial clínico de un paciente.

Reporte::modificarEnfermedadCronica: Permite registrar o modificar una enfermedad crónica para un paciente.

Archivos.txt

La carpeta output y los archivos .txt que contiene (excepto servicios.txt) almacenan los datos gestionados por las funciones de gestorArchivos.h. A continuación, se describe el propósito de cada archivo y se proporciona un ejemplo de los datos almacenados.

pacientes.txt

Almacena la información de los pacientes registrados en el sistema.

Ejemplo de guardado:

Nombre: Juan Perez

ID: 1234

Fecha de ingreso: 01-01-2025

medicos.txt

Almacena la información de los médicos registrados en el sistema.

Ejemplo de guardado:

Dr. Ana Garcia

ID: 0001

Servicio: Cardiología

Disponibilidad: 1

citas.txt

Almacena la información de las citas médicas registradas en el sistema.

Ejemplo de guardado:

Fecha: 20-01-2025

Paciente: Juan Perez (ID: 1234)

Medico: Ana Garcia (ID: 0001)

Urgencia: 3

`pacientes_cronicos.txt`

Almacena la información de los pacientes con enfermedades crónicas.

Ejemplo de guardado:

Nombre: Juan Perez

Enfermedad cronica: Si

`servicios.txt`

Este archivo .txt no guardara nuevos datos, sino que tiene unos datos constantes e inmodificables que sirven para verificar si un servicio existe en el sistema.

`carpeta backup`

Almacena archivos de backup en caso de pérdida de datos. Estos archivos backup se generan en `gestorArchivos.h` individualmente por cada función de guardado en un .txt. Por ejemplo, cuando se guardan los datos de un paciente modificado en su .txt, también se genera un archivo backup con la fecha y hora de guardado.

Algoritmos de Búsqueda y Ordenación

Búsqueda lineal

El algoritmo principal utilizado en el proyecto para encontrar elementos en los vectores de pacientes, médicos o citas es la búsqueda lineal. Se recorre el vector y se compara cada elemento con el criterio de búsqueda hasta encontrar el resultado o llegar al final del vector. A continuación, se muestran algunos casos de uso del algoritmo de búsqueda lineal.

Búsqueda de pacientes por ID en `main.cpp`

La función `buscarPacientePorID` recorre el vector de pacientes y compara cada ID con el proporcionado para encontrar al paciente correspondiente. Este es un caso típico de búsqueda lineal, ya que el código no utiliza estructuras de datos optimizadas para búsquedas rápidas.

Búsqueda de médicos por ID en main.cpp

De manera similar a la búsqueda de pacientes, la función `buscarMedicoPorID` recorre el vector de médicos y compara los ID para encontrar el médico correspondiente.

Búsqueda de citas médicas por fecha o urgencia en CitaMedica

Aunque el código no proporciona un algoritmo específico de búsqueda para las citas, se pueden utilizar búsquedas lineales al recorrer el vector de citas y comparar cada cita con un valor específico (como una fecha o urgencia). Esto también puede considerarse una forma de búsqueda lineal.

Posibles optimizaciones

Si el proyecto llegara a crecer y manejar una gran cantidad de datos, se podría considerar el uso de algoritmos de búsqueda más eficientes como búsqueda binaria o estructuras de datos como mapas o árboles de búsqueda binaria, pero por el momento parece que la búsqueda lineal es suficiente.

Bubble Sort

El algoritmo principal utilizado en el proyecto para ordenar las citas médicas es el Bubble Sort. Este algoritmo recorre el vector de citas y compara elementos adyacentes, intercambiándolos si están en el orden incorrecto, hasta que todos los elementos estén ordenados de acuerdo con el criterio definido. A continuación, se muestran algunos casos de uso del algoritmo de ordenación por burbuja.

Ordenación de citas por fecha en CitaMedica

La función `ordenarPorFecha` recorre el vector de citas médicas y compara las fechas de las citas, desglosándolas en día, mes y año para determinar el orden correcto. Si una cita tiene una fecha anterior a la de la cita siguiente, las citas se intercambian.

Ordenación de citas por urgencia en CitaMedica

La función `ordenarPorUrgencia` compara los niveles de urgencia de las citas médicas en el vector y, si una cita tiene un nivel de urgencia menor que la siguiente, las dos citas se intercambian.

Ordenación de citas en el historial clínico (`verHistorialClinico`)

En la función `verHistorialClinico`, también se utiliza el algoritmo de Bubble Sort para ordenar las citas por fecha antes de mostrar el historial de un paciente. Se recorre el vector de citas y se ordenan según la fecha, utilizando el mismo procedimiento de comparación por día, mes y año, garantizando que las citas se muestren en orden cronológico.

Posibles optimizaciones

Aunque el algoritmo de Bubble Sort es sencillo de implementar y entender, no es el más eficiente para grandes volúmenes de datos debido a su complejidad de $O(n^2)$. Si el proyecto llegara a manejar una cantidad considerable de citas médicas, se podrían considerar algoritmos de ordenación más eficientes, como Quick Sort o Merge Sort, que tienen una complejidad promedio de $O(n \log n)$ y son mucho más rápidos para conjuntos de datos grandes. Sin embargo, para un número moderado de citas, Bubble Sort sigue siendo suficiente, manteniendo la simplicidad y claridad del código.

Conclusiones

Como conclusión, C++ es un lenguaje que permite una gestión eficiente de memoria y la implementación de estructuras de datos complejas, lo cual es esencial para un sistema de gestión hospitalaria. Con relación al desarrollo del proyecto, se ha buscado una separación de archivos firme para facilitar la organización del código. Además, desde un principio se planteó el programa entorno al archivo main.cpp, ya que tener un solo archivo ejecutable simplifica la distribución y ejecución del programa. También asegura que todas las funcionalidades estén integradas en un solo punto, facilitando la gestión de versiones y despliegues.

En partes finales de desarrollo, claramente se ve que centralizar el programa en un .cpp dificulta mucho la evolución y el desarrollo del programa. Al analizar en profundidad principios SOLID y patrones de diseño, este proyecto hecho en C++ sería inservible a la hora de aplicar en un caso real, por la poca escalabilidad que ofrece, y la complejidad que supondría arreglar un código sin clases con jerarquía de herencias y métodos sin encapsulamiento.

Como primer proyecto complejo usando C++, he visto lo que supone no organizar y estructurar un código desde el principio de desarrollo. Por todo esto, he aprendido que es necesario seguir desde un principio unas pautas marcadas de diseño para la programación orientada a objetos.

Bibliografía

En este proyecto se ha utilizado IA

Se han usado herramientas, fuentes de información o páginas como:

<https://cplusplus.com/reference/>

<https://en.cppreference.com/w/>

<https://stackoverflow.com/>

<https://www.w3schools.com/>

https://drive.google.com/file/d/1c6xAHslvrUJ4qzpnYuY0ZuBod7v6Yfcn/view?usp=classroom_web&authuser=2

<https://refactoring.guru/es/design-patterns/facade>

luhan omar (2021). Método de búsqueda hash | implementación C++. Disponible en: <https://www.youtube.com/watch?v=XVgJY27UuM4> [Accedido el 22 de octubre de 2024].

SW Team (2023). Algoritmos de ordenación con ejemplos en C++. Disponible en: <https://www.swhosting.com/es/comunidad/manual/algoritmos-de-ordenacion-con-ejemplos-en-c> [Accedido el 24 de octubre de 2024].

RUDE LABS (2024). Hospital Management System With C++ | C++ Project. Disponible en: <https://www.youtube.com/watch?v=SxSR9UoLmIU> [Accedido el 15 de noviembre de 2024].

Torres A. (2016). Búsqueda lineal en vectores. Disponible en: <https://www.youtube.com/watch?v=8DJNa1uIKUc> [Accedido el 20 de enero de 2025].

Huamán R. (2020) STUPID VS SOLID. Disponible en: <https://codideep.com/blogpost/stupid-vs-solid> [Accedido el 21 de enero de 2025].

Anexos

https://github.com/PabloNSI/AB_SOTO_PABLO_UNIT20.git