UNIVERSIDAD DE SAN CARLOS DE GUATEMALA FACULTAD DE INGENIERÍA
LAB IPC1
AUX RODRIGO ANTONIO PORÓN DE LEÓN

MANUAL TECNICO

Daniel André Hernández Flores 202300512
Guatemala 5 de marzo del año 2024

Introducción

El presente Manual Técnico constituye una herramienta fundamental para aquellos involucrados en el desarrollo, mantenimiento y comprensión profunda del sistema de software. En él, se detalla minuciosamente la estructura interna del código fuente, así como los principios de diseño, las tecnologías empleadas y las decisiones arquitectónicas que sustentan su funcionamiento. Este documento se destina a desarrolladores, ingenieros de software, y cualquier persona interesada en adentrarse en los entresijos del sistema. Su objetivo es proporcionar una visión global y detallada del funcionamiento interno del software, facilitando la comprensión de su lógica, la interacción entre sus componentes y la forma en que se implementan sus funcionalidades.

A lo largo de este manual, se abordan aspectos cruciales del desarrollo de software, tales como la modularidad, la reutilización de código, la optimización del rendimiento y la escalabilidad. Se describe la arquitectura del sistema, destacando la relación entre los distintos módulos y componentes, así como las dependencias y las interacciones entre ellos. Además, se profundiza en la documentación del código, proporcionando pautas y buenas prácticas para la escritura de comentarios y la estructuración del código fuente. Se ofrecen también recomendaciones para el mantenimiento y la actualización del software, así como para la resolución de problemas y la depuración de errores.

Requisitos Para ejecutar el programa

Requisitos del Sistema:

1. **Sistema Operativo:**
    - Windows 7 o superior.
    - macOS 10.12 Sierra o superior.
    - Distribuciones Linux compatibles.

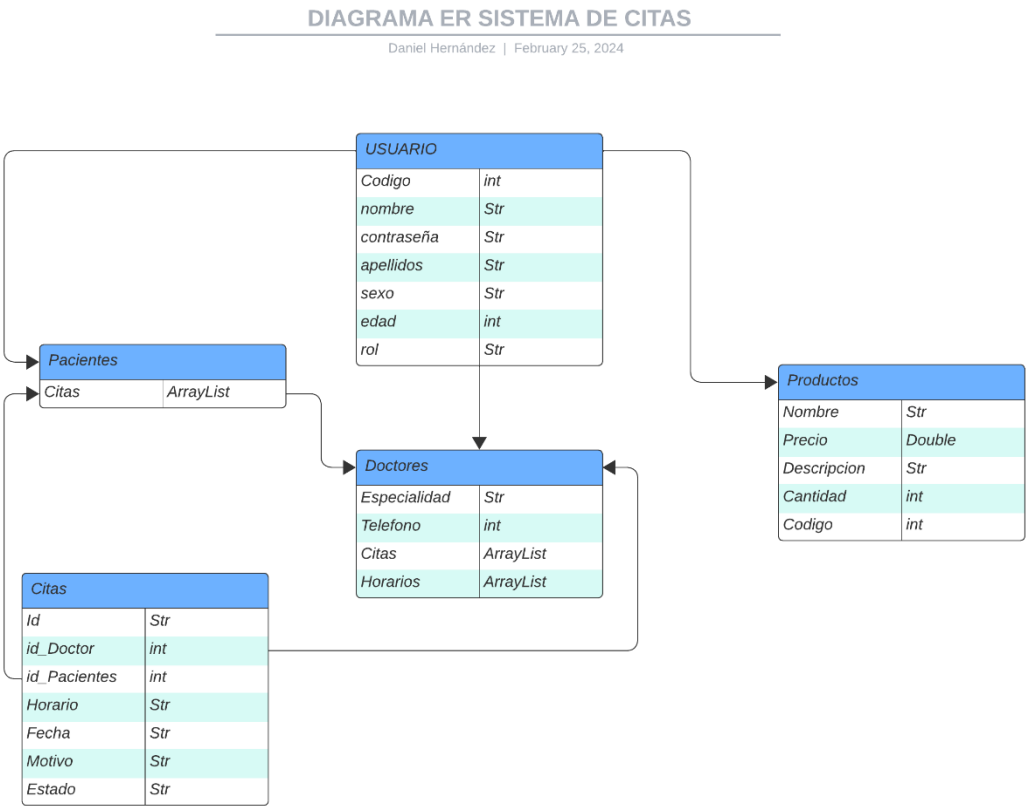
2. **Hardware**:
    - Procesador: Intel Core i3 o equivalente.
    - Memoria RAM: 4 GB de RAM.
    - Almacenamiento: 100 MB de espacio disponible en disco duro.

3. **Software**:
    - Java Development Kit (JDK) 8 o superior.
    - IDE compatible con Java, como Eclipse, IntelliJ IDEA o NetBeans.
    - Conexión a Internet (para descargar dependencias y actualizaciones).

4. **Bibliotecas y Dependencias:**
    - SwingX: Framework de componentes Swing adicionales.
    - JFreeChart: Biblioteca para la creación de gráficos.

Estos requisitos son mínimos y pueden variar dependiendo de la complejidad del sistema y las funcionalidades específicas utilizadas. Es importante tener en cuenta que el rendimiento del programa puede verse afectado por el hardware y la carga de trabajo del sistema en el que se ejecuta.

DIAGRAMA ER DEL PROYECTO



Funciones en clase Persona_Controller

1. Función Login:

La función Login es responsable de autenticar a los usuarios en el sistema. Toma como entrada un código de usuario y una contraseña, luego verifica si coinciden con las credenciales almacenadas en la base de datos del sistema. Si las credenciales son correctas, la función devuelve un panel correspondiente al tipo de usuario que ha iniciado sesión (administrador, paciente o doctor), lo que permite al usuario acceder a las funciones y características específicas de su rol en la aplicación. Si las credenciales son incorrectas, se muestra un mensaje de error. Esta función es fundamental para garantizar la seguridad y el acceso controlado a las distintas áreas y funcionalidades del sistema.

Codigo fuente:

```
public javax.swing.JPanel Login(int codigo, String password){

    if (admin.getid() == codigo && admin.getContraseña().equals(password)){
        id_Persona_Logueada = admin.getid();
        rol_Persona_Logueada = admin.getRol();
        cambioPanel = panel_Control.get_Panel_Menu_Administrador();
        JOptionPane.showMessageDialog(null, "Se ha iniciado Sesion");
        return cambioPanel;
    }

    for (int i = 0; i < lista_Pacientes.size(); i++){
        Persona persona = lista_Pacientes.get(i);
        if (persona.getid() == codigo && persona.getContraseña().equals(password)){
            id_Persona_Logueada = persona.getid();
            rol_Persona_Logueada = persona.getRol();
            cambioPanel = panel_Control.get_Panel_Menu_Paciente();
            JOptionPane.showMessageDialog(null, "Se ha iniciado Sesion");
            return cambioPanel;
        }
    }

    for (int i = 0; i < lista_Doctores.size(); i++){
        Persona persona = lista_Doctores.get(i);
        if (persona.getid() == codigo && persona.getContraseña().equals(password)){
            id_Persona_Logueada = persona.getid();
            rol_Persona_Logueada = persona.getRol();
            JOptionPane.showMessageDialog(null, "Se ha iniciado Sesion");
            cambioPanel = panel_Control.get_Panel_Menu_Doctor();
            return cambioPanel;
        }
    }
    JOptionPane.showMessageDialog(null,"Contraseña o codigo incorrecto");
    return null;
}
```

2. **Función registrarFrecuenciaEspecialidad:**

La función registrarFrecuenciaEspecialidad tiene como objetivo registrar la frecuencia de las especialidades médicas entre los doctores en el sistema. Para ello, utiliza una lista de doctores como entrada y devuelve una lista de objetos Frecuencia, que contiene la especialidad y su frecuencia correspondiente.

1. Se crea una lista vacía llamada frecuencias para almacenar los objetos Frecuencia.
2. Se itera sobre la lista de doctores para procesar cada uno de ellos.
3. Para cada doctor, se busca si la especialidad del doctor ya está registrada en la lista de frecuencias.
    - Si la especialidad ya está en la lista, se incrementa la frecuencia de esa especialidad.
    - Si la especialidad no está en la lista, se agrega un nuevo objeto Frecuencia con la especialidad y una frecuencia inicial de 1.
4. Después de procesar todos los doctores, se ordena la lista de frecuencias en orden descendente según la frecuencia.
5. Finalmente, se devuelve la lista de frecuencias actualizada.

Este proceso garantiza que la función genere una lista de frecuencias que representan la distribución de las especialidades entre los doctores, lo que puede ser útil para análisis y toma de decisiones dentro del sistema.
Código fuente:

```
public ArrayList<Frecuencia> registrarFrecuenciaEspecialidad() {
    // Lista para almacenar las frecuencias de las especialidades
    ArrayList<Frecuencia> frecuencias = new ArrayList<>();
    // Iterar sobre la lista de doctores
    for (Doctor doctor : lista_Doctores) {
        boolean encontrada = false;
        // Buscar si ya existe la especialidad en la lista de frecuencias
        for (Frecuencia frecuencia : frecuencias) {
            if (frecuencia.getDato().equals(anObject: doctor.getEspecialidad())) {
                // Si la especialidad ya está en la lista, incrementar su frecuencia
                frecuencia.incrementarFrecuencia();
                encontrada = true;
                break;
            }
        }
        // Si la especialidad no está en la lista, agregarla con frecuencia 1
        if (!encontrada) {
            frecuencias.add(new Frecuencia(dato: doctor.getEspecialidad(), frecuencia: 1));
        }
    }
    ordenarFrecuenciasDescendente(frecuencias);
    // Devolver la lista de frecuencias
    return frecuencias;
}
```

3. **Funcion ordenarFrecuenciasDescendente:**

Se encarga de ordenar una lista de objetos Frecuencia en orden descendente según el valor de su frecuencia. Aquí está el desglose técnico de la función:

1. Se define un comparador personalizado utilizando la interfaz Comparator. Este comparador se utiliza para comparar dos objetos Frecuencia y determinar su orden relativo en función de la frecuencia.

2. El comparador implementa el método compare, que toma dos objetos Frecuencia, f1 y f2, como parámetros y devuelve un valor entero que indica su orden relativo.

3. Dentro del método compare, se utiliza Integer.compare para comparar las frecuencias de f1 y f2. Se compara el valor de f2 con f1 para lograr un orden descendente.

   - Si f2 tiene una frecuencia mayor que f1, se devuelve un valor negativo.
   - Si f1 tiene una frecuencia mayor que f2, se devuelve un valor positivo.
   - Si las frecuencias son iguales, se devuelve 0.

4. Luego, se utiliza el comparador personalizado para ordenar la lista de frecuencias utilizando el método Collections.sort. Este método ordena la lista en función del comparador proporcionado, lo que resulta en una lista ordenada en orden descendente según la frecuencia.

Esta función permite ordenar una lista de frecuencias en orden descendente, lo que facilita la visualización de las especialidades más frecuentes en primer lugar. Esto es útil para análisis y presentación de datos en el sistema.

Código Fuente:

```
public void ordenarFrecuenciasDescendente(ArrayList<Frecuencia> frecuencias) {
    // Definir un comparador personalizado para ordenar las frecuencias por valor de manera descendente
    Comparator<Frecuencia> comparador = new Comparator<Frecuencia>() {
        @Override
        public int compare(Frecuencia f1, Frecuencia f2) {
            // Ordenar de manera descendente por valor de frecuencia
            return Integer.compare(x: f2.getFrecuencia(), y: f1.getFrecuencia());
        }
    };

    // Ordenar la lista de frecuencias usando el comparador
    Collections.sort(list: frecuencias, c: comparador);
}
```

4. Funcion asignar_Cita:

1. **Propósito**: Esta función se encarga de asignar una cita a un doctor y a un paciente en las respectivas listas de citas de cada uno.

2. **Parámetros**:
   - cita_Nueva: La cita que se va a asignar.

3. **Descripción detallada:**
   - La función itera sobre la lista de doctores para encontrar al doctor correspondiente a la cita nueva.
   - Una vez encontrado el doctor, se agrega la cita nueva a su lista de citas utilizando el método agregarCita.
   - La función realiza un proceso similar para encontrar al paciente correspondiente a la cita y agregar la cita nueva a su lista de citas.
   - Se utiliza un contador para mantener un seguimiento del índice en la lista de doctores y pacientes mientras se realiza la búsqueda.

5. Función cambiar_Estado_Cita:

1. **Propósito**: Esta función se encarga de cambiar el estado de una cita específica para un doctor y un paciente.

2. **Parámetros**:
   - estado_Nuevo: El nuevo estado que se asignará a la cita.
   - id_Doctor: El ID del doctor asociado a la cita.
   - id_Paciente: El ID del paciente asociado a la cita.
   - id_Cita: El ID de la cita que se actualizará.

3. **Descripción detallada**:
   - La función utiliza un controlador de citas (Cita_Controller) para realizar la actualización del estado de la cita en las listas de citas del doctor y del paciente.
   - Itera sobre la lista de doctores para encontrar al doctor correspondiente al ID proporcionado.
   - Una vez encontrado el doctor, utiliza el controlador de citas para actualizar la lista de citas del doctor con el nuevo estado de la cita utilizando el método actualizar_Lista_Citas.
   - La función realiza un proceso similar para encontrar al paciente correspondiente al ID proporcionado y actualizar su lista de citas con el nuevo estado de la cita.
   - Se utiliza un contador para mantener un seguimiento del índice en la lista de doctores y pacientes mientras se realiza la búsqueda.

Código fuente:

```
    public void asignar_Cita(Cita cita_Nueva){
        int contador = 0;
        for( Doctor doctor: lista_Doctores ){
            if(doctor.getid() == cita_Nueva.getId Doctor()){
                lista_Doctores.get(index: contador).agregarCita(cita: cita_Nueva);
                break;
            }
            contador ++;
        }
        contador = 0;
        for( Paciente paciente: lista_Pacientes ){
            if( paciente.getid() == cita_Nueva.getId Paciente() ){
                lista_Pacientes.get(index: contador).agregarCita(cita: cita_Nueva);
                break;
            }
            contador ++;
        }
    }
```

```
public void cambiar Estado Cita(String estado_Nuevo, int id_Doctor, int id_Paciente, int id_Cita){
    Cita_Controller cita_Controller = new Cita_Controller();

    for(Doctor doctor: lista_Doctores){
        if(doctor.getid() == id_Doctor){
            doctor.citas = cita_Controller.actualizar_Lista_Citas(id_Cita, lista_Citas: doctor.getCitas(), estado_Cita: estado_Nuevo, rol:"Doctor");
            break;
        }

    }
```

Funciones en la clase cita_Controller

Propósito:

- La clase Cita_Controller se encarga de manejar la lógica relacionada con las citas en el sistema, como la generación de nuevas citas y la actualización de la lista de citas.

Atributos:

- contador: Un atributo estático que se utiliza para generar identificadores únicos para cada cita. Se inicializa en 1 y se incrementa cada vez que se genera una nueva cita.

Métodos:

- generar_Cita(int id_Doctor, int id_Paciente, String horario, String fecha, String motivo)

Este método se utiliza para generar una nueva cita con los parámetros proporcionados y devuelve la cita creada.

- **Parámetros**:
  - id_Doctor: El ID del doctor asociado a la cita.
  - id_Paciente: El ID del paciente asociado a la cita.
  - horario: El horario de la cita.
  - fecha: La fecha de la cita.
  - motivo: El motivo de la cita.

- **Funcionamiento**:
  - Crea una nueva instancia de Cita con el contador actual como ID y los parámetros proporcionados.
  - Incrementa el contador para el siguiente ID de cita.
  - Retorna la cita recién creada.

actualizar_Lista_Citas(int id_Cita, ArrayList<Cita> lista_Citas, String estado_Cita, String rol):

Este método se utiliza para actualizar el estado de una cita específica en la lista de citas.

- **Parámetros**:
  - id_Cita: El ID de la cita que se actualizará.
  - lista_Citas: La lista de citas en la que se realizará la actualización.
  - estado_Cita: El nuevo estado que se asignará a la cita.
  - rol: El rol del usuario que solicita la actualización (Doctor o Paciente).
- **Funcionamiento**:
  - Itera sobre la lista de citas para encontrar la cita con el ID proporcionado.
  - Si se encuentra la cita:
  - Si el rol es "Doctor", elimina la cita de la lista.
  - Si el rol es "Paciente", actualiza el estado de la cita con el nuevo estado proporcionado y actualiza la cita en la lista.
  - Retorna la lista actualizada de citas.

-

Código fuente:

```
public class Cita_Controller {

    static int contador = 1;

    public Cita generar Cita(int id_Doctor, int id_Paciente, String horario, String fecha, String motivo){
        Cita cita_Nueva = new Cita(id: contador, id_Doctor, id_Paciente, horario, fecha, motivo, estado_Cita: "Pendiente");
        contador++;
        return cita_Nueva;
    }

    public ArrayList<Cita> actualizar_Lista_Citas(int id_Cita, ArrayList<Cita> lista_Citas, String estado_Cita, String rol){
        int contador = 0;
        for( Cita cita: lista_Citas ){
            if( cita.getId() == id_Cita ){
                if(rol.equals(anObject: "Doctor")){
                    lista_Citas.remove(index: contador);
                    return lista_Citas;
                }else{
                    cita.setEstado Cita(estado_Cita);
                    lista_Citas.set(index: contador, element: cita);
                    return lista_Citas;
                }
            }
            contador++;
        }
        return null;
    }

}
```

Funciones en la clase Panel_Controller

La clase Panel_Controller es una clase que proporciona un mecanismo centralizado para obtener instancias de todos los paneles que se utilizan en la aplicación. Su propósito principal es ofrecer un control organizado y dinámico sobre la gestión de paneles en la interfaz gráfica del programa.

Explicación general del uso:

1. **Control de instancias:** La clase Panel_Controller se encarga de crear y proporcionar instancias de diferentes paneles utilizados en la aplicación. Cada método público de esta clase está diseñado para devolver una instancia específica de un panel.

2. **Centralización de la lógica de paneles**: Al centralizar la lógica de creación de instancias de paneles en una sola clase, se mejora la organización y mantenibilidad del código. En lugar de tener la creación de paneles dispersa por todo el código, se concentra en un único lugar, lo que facilita su gestión y modificación.

3. **Actualización dinámica de paneles**: Además de proporcionar instancias de paneles, esta clase también puede incluir lógica para actualizar la información en los paneles según sea necesario. Esto permite una interfaz más dinámica y actualizada para el usuario, ya que los paneles pueden reflejar cambios en los datos o en el estado de la aplicación en tiempo real.

El Panel_Controller actúa como un punto central para el manejo de instancias de paneles, proporcionando una forma ordenada y dinámica de gestionar la interfaz gráfica de la aplicación. Su objetivo es simplificar el acceso a los paneles y facilitar su actualización según las necesidades del programa.

Codigo fuente:

```
22      /*
23          Esta clase me brinda a mi el control de todos los Paneles, para poder
24          controlar las instancias y hacer el cambio en la ventana Base, para
25          tener un mayor control sobre mi programa
26      */
27      
28      Main main = new Main();
29      
30      public Panel_Login get_Panel_Login(){...4 lines }
34      
35      public Ventana_Base get_Ventana_Base(){...3 lines }
38      
39      public Panel_Registrarse get_Panel_Registrarse(){...4 lines }
43      
44      public Panel_Menu_Administrador get_Panel_Menu_Administrador(){...4 lines }
48      
49      public Panel_Control_Doctores get_Panel_Control_Doctores(){...4 lines }
53      
54      public Panel_Reportes get_Panel_Reportes(){...4 lines }
58      
59      public Panel_Control_Pacientes get_Panel_Control_Pacientes(){...4 lines }
63      
64      public Panel_Control_Producto get_Panel_Control_Producto(){...4 lines }
68      
69      public Panel_Menu_Paciente get_Panel_Menu_Paciente(){...4 lines }
73      
74      public Panel_Crear_Citas get_Panel_Crear_Citas(){...4 lines }
78      
79      public Panel_Perfil get_Panel_Perfil(){...4 lines }
83      
84      public Panel_Menu_Doctor get_Panel_Menu_Doctor(){...4 lines }
88      
89      public Panel_Perfil_Doctor get_Panel_Perfil_Doctor(){...4 lines }
93      
94      public Panel_Citas_Doctor get_Panel_Citas_Doc(){...4 lines }
98  }
```