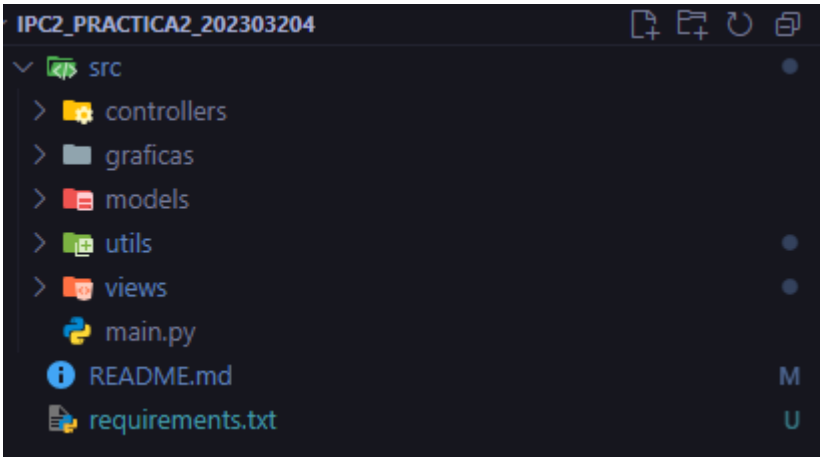
EXPLICACIÓN COMPLETA DEL SISTEMA DE TURNOS MÉDICOS

1. ARQUITECTURA GENERAL DEL PROYECTO

Patrón MVC (Model-View-Controller) o en capas.



¿Por qué MVC?

Separación de responsabilidades: Cada parte tiene una función específica
Mantenibilidad: Fácil de modificar sin afectar otras partes
Escalabilidad: Se puede crecer el proyecto ordenadamente

2. MODELS - ESTRUCTURA DE DATOS

nodo.py - El Elemento Básico

```
You, 3 seconds ago | 1 author (You)
class Nodo:
    def __init__(self, info):
        self.info = info
        self.siguiente = None

    def obtener_info(self):
        return self.info

    def obtener_siguiente(self):
        return self.siguiente

    def establecer_siguiente(self, nuevo_siguiente):
        self.siguiente = nuevo_siguiente
```

¿Qué hace?
Es el elemento básico de la lista enlazada

- Cada nodo contiene:
    - info: La información (un objeto Paciente)
    - siguiente: Referencia al próximo nodo en la cadena

Conceptos aplicados:

- ☑ ~~Estructura de datos dinámica~~
- ☑ ~~Punteros/Referencias~~
- ☑ ~~Encapsulamiento (métodos para acceder a los datos)~~

paciente.py - El modelo de datos principal

```python
class Paciente:
    def __init__(self, nombre, edad, especialidad):
        self.nombre = nombre
        self.edad = edad
        self.especialidad = especialidad
        self.tiempo_atencion = self.TIEMPOS_ESPECIALIDAD.get(especialidad, 10)
        self.tiempo_registro = None
        self.tiempo_espera_estimado = 0

    TIEMPOS_ESPECIALIDAD = {
        "Medicina General": 10,
        "Pediatría": 15,
        "Ginecología": 20,
        "Dermatología": 25
    }
```

¿Qué hace cada parte?
TIEMPOS_ESPECIALIDAD (diccionario de clase):

- Mapea cada especialidad con su tiempo de atención
- Es un atributo de clase (compartido por todas las instancias)

```python
def obtener_tiempo_atencion(self):
    return self.tiempo_atencion
```

- Calcula el tiempo total que un paciente estará en el sistema
- Suma: tiempo esperando + tiempo de atención

establecer_tiempo_espera_estimado():
Se llama desde la Cola cuando se actualiza la posición del paciente

Conceptos aplicados:

- Programación Orientada a Objetos
- Atributos de clase vs instancia
- Métodos de cálculo
- Representación de datos del mundo real

cola.py - Estructura FIFO

```python
class ColaPacientes:
    def __init__(self):
        self.primero = None  # Frente de la cola (próximo a atender)
        self.ultimo = None   # Final de la cola (último en llegar)
```

Operaciones principales:

encolar(paciente) - Agregar al final
desencolar() - Atender al primero

```python
def esta_vacia(self):
    return self.primero is None

def encolar(self, paciente):
    nuevo_nodo = Nodo(paciente)

    if self.esta_vacia():
        self.primero = nuevo_nodo
        self.ultimo = nuevo_nodo
    else:
        self.ultimo.establecer_siguiente(nuevo_nodo)
        self.ultimo = nuevo_nodo

    self._actualizar_tiempos_espera()

def desencolar(self):
    if self.esta_vacia():
        return None

    paciente_atendido = self.primero.obtener_info()
    self.primero = self.primero.obtener_siguiente()

    if self.primero is None:
        self.ultimo = None

    self._actualizar_tiempos_espera()

    return paciente_atendido
```

¿Cómo funciona el encolar?

- Crea un nuevo nodo con el paciente
- Si la cola está vacía: este nodo es primero Y último
- Si NO está vacía: conecta al final y actualiza el puntero último
- Recalcula todos los tiempos de espera

¿Cómo funciona el desencolar?

- Guarda el paciente que está al frente
- Mueve el puntero primero al siguiente nodo
- Si ya no hay nadie, también limpia último
- Recalcula tiempos porque todos avanzaron una posición

_actualizar_tiempos_espera() - El Algoritmo Clave

```
def _actualizar_tiempos_espera(self):
    actual = self.primero
    tiempo_acumulado = 0

    while actual is not None:
        paciente = actual.obtener_info()
        paciente.establecer_tiempo_espera_estimado
        (tiempo_acumulado)
        tiempo_acumulado += paciente.obtener_tiempo_atencion()
        actual = actual.obtener_siguiente()
```

¿Cómo funciona este algoritmo?

1. Empieza desde el primero con tiempo_acumulado = 0
2. El primer paciente espera 0 minutos (lo atienden inmediatamente)
3. El segundo espera = tiempo de atención del primero
4. El tercero espera = tiempo del primero + segundo
5. Y así sucesivamente…

Conceptos aplicados:

- Cola FIFO (First In, First Out)
- Lista enlazada dinámica
- Algoritmos de recorrido
- Complejidad temporal O(1) para encolar/desencolar
- Manejo de punteros

Controllers/turnos.py - Lógica de Negocio

```
You, 3 days ago | 1 author (You)
class ControladorTurnos:
    def __init__(self):
        self.cola = ColaPacientes()
        self.pacientes_atendidos = []  # Lista de pacientes ya atendidos
        self.total_pacientes_atendidos = 0
```

registrar pacientes

¿Qué válida?

- Nombre: No vacío, sin espacios extra
- Edad: Número entero entre 0-120
- Especialidad: Debe estar en el catálogo
- Duplicados: No puede haber dos pacientes con el mismo nombre
- Timestamp: Guarda cuándo se registró

atender pacientes

¿Qué hace internamente?

- Verifica que haya pacientes
- Desencola al primero (automáticamente actualiza tiempos)
- Marca timestamp de atención
- Mueve al historial
- Incrementa contador
- Retorna información formateada

Conceptos aplicados:

- Validación de datos
- Manejo de excepciones
- Lógica de negocio
- Patrón Controller
- Separación de responsabilidades

views/main_page - interfaz con tkinter

```python
class ModernMedicalApp:
    def __init__(self, root):
        self.root = root
        self.setup_window()
        self.setup_style()
        self.controlador = ControladorTurnos()
        self.graphviz = GraphvizGenerator()

        self.current_image = None
        self.stats_image = None
        self.auto_refresh = tk.BooleanVar(value=True)

        self.create_widgets()
        self.setup_layout()
        self.update_display()

        self.auto_update_loop()
```

¿Por qué este orden?

- Configuración base antes de crear widgets
- Instancias de lógica antes de conectar eventos
- Caché de imágenes para optimizar rendimiento
- Variables de control para estado de la aplicación

setup_style() - Sistema de Estilos Personalizado

- ¿Cómo funciona el sistema de estilos?
- ttk.Style(): Maneja estilos de widgets ttk
- theme_use('clam'): Tema base moderno
- configure(): Establece estilos por defecto
- map(): Define estados (normal, hover, pressed)
- Colores consistentes: Paleta unificada para toda la app

create_widgets() - Construcción de la Interfaz

UTILS - GENERACIÓN DE VISUALIZACIONES
generate_queue_graph

¿Cómo funciona el algoritmo?

- Validación: Verifica que Graphviz esté disponible
- Configuración: Establece dirección y estilos del grafo
- Casos especiales: Maneja cola vacía con mensaje apropiado
- Header informativo: Resumen de la cola actual
- Generación de nodos: Un nodo por paciente con toda su info
- Conexiones visuales: Flechas mostrando el orden FIFO
- Renderizado: Convierte a imagen PNG
- Manejo de errores: Try-catch robusto

FLUJO COMPLETO DE LA APLICACIÓN
Inicialización (main.py → main_page.py):

1. main() crea tk.Tk()
2. ModernMedicalApp(root) se inicializa
3. setup_window() configura ventana principal
4. setup_style() define estilos visuales
5. ControladorTurnos() se instancia (crea Cola vacía)
6. GraphvizGenerator() se inicializa
7. create_widgets() construye toda la UI
8. update_display() hace primera visualización
9. auto_update_loop() inicia actualización automática cada 5s
10. root.mainloop() inicia el event loop de Tkinter

Flujo de Registro de Paciente:
USER CLICK [Registrar] → registrar_paciente() (UI)
↓
Validar formulario (UI)
↓
controlador.registrar_paciente() (Controller)
↓
Validar datos de negocio (Controller)
↓
Paciente() crear instancia (Model)
↓
cola.encolar() agregar a cola (Model)
↓
_actualizar_tiempos_espera() (Model)
↓
update_display() actualizar UI (View)
↓
generate_queue_graph() nueva visualización (Utils)
↓
Mostrar mensaje de éxito (UI)

Flujo de Atención de Paciente:
USER CLICK [Atender] → atender_paciente() (UI)
↓
controlador.atender_paciente() (Controller)
↓
cola.desencolar() sacar primero (Model)
↓
_actualizar_tiempos_espera() recalcular (Model)
↓
Agregar a historial (Controller)
↓
update_display() actualizar UI (View)
↓
generate_queue_graph() nueva visualización (Utils)
↓
Mostrar info del paciente atendido (UI)

Flujo de Actualización Automática:
auto_update_loop() ejecuta cada 5 segundos
↓
Si auto_refresh está marcado:
↓
update_display()
↓
update_queue_visualization() → Regenera gráfico Graphviz
↓
update_system_status() → Actualiza barra inferior
↓
root.after(5000, auto_update_loop) → Programa siguiente ejecución

CONCEPTOS TÉCNICOS APLICADOS
Estructuras de Datos:

- Lista Enlazada: Implementación dinámica de la cola
- Cola FIFO: First In, First Out para orden de atención
- Nodos: Elementos con datos y referencias
- Punteros: Referencias para navegación

Patrones de Diseño:

- MVC: Separación Model-View-Controller
- Observer: Actualización automática de UI
- Strategy: Diferentes estrategias de visualización (Graphviz vs texto)
- Factory: Creación de widgets similares

Programación Asíncrona (básica):

- Event Loop: Tkinter maneja eventos de usuario
- Callbacks: Funciones que responden a eventos
- Timer: root.after() para actualización automática
- Non-blocking: UI no se congela durante operaciones

Manejo de Errores:

- Try-Catch: Captura de excepciones
- Validación en capas: UI y Controller
- Graceful degradation: Funciona sin Graphviz
- Mensajes informativos: Feedback claro al usuario

Algoritmos:

- Recorrido de lista: Para calcular tiempos
- Búsqueda lineal: Para encontrar pacientes
- Cálculo de tiempos: Algoritmo de suma acumulativa
- Ordenamiento: Mantenimiento de orden FIFO

¿POR QUÉ TKINTER?

1. Ventajas:
    a. Nativo de Python: No requiere instalación extra
    b. Multiplataforma: Windows, macOS, Linux
    c. Maduro y estable: Años de desarrollo
    d. Documentación extensa: Muchos recursos
    e. Widget variado: Botones, labels, frames, canvas, etc.

2. Widgets Utilizados:
    a. tk.Tk(): Ventana principal
    b. tk.Frame(): Contenedores para organizar
    c. ttk.Button(): Botones con estilo moderno
    d. ttk.Entry(): Campos de texto
    e. ttk.Combobox(): Dropdown de especialidades
    f. ttk.LabelFrame(): Secciones con título
    g. tk.Label(): Textos e imágenes
    h. messagebox: Diálogos de información/error

3. Layout Managers:
    a. pack(): Para disposición lineal
    b. grid(): Para formularios (filas/columnas)
    c. place(): Para posicionamiento absoluto (no usado aquí)

¿POR QUÉ REQUIREMENTS.TXT?

Propósito: (pip install -r requirements.txt # Instala todas las dependencias)

- Reproducibilidad: Mismas versiones en cualquier máquina
- Gestión de dependencias: Instalación automatizada
- Control de versiones: Evita conflictos entre versiones
- Estándar de la industria: Práctica común en Python