UNIVERSIDAD DE LAS AMÉRICAS PUEBLA

PROGRAMACIÓN ORIENTADA A OBJETOS
O25-LIS1022-3

Juan Manuel Gonzalez Calleros

SERVICIOS MÉDICOS UDLAP

Proyecto final

Equipo 6
Alberto Amador Loza
Brian Carbajal Ortiz
Jeronimo Sanchez Armas Torres
Marcos Miguel Gutierrez Toniz
Ricardo Montiel Bribiesca

4 / Diciembre / 2025

1. Resumen ejecutivo

El proyecto aborda una problemática recurrente dentro del campus de la Universidad de las Américas Puebla (UDLAP): los tiempos prolongados de espera en los servicios médicos universitarios. Diversos estudiantes han manifestado que, aun cuando presentan malestares menores o situaciones que no requieren atención inmediata, deben desplazarse físicamente a la enfermería y esperar durante periodos que, con frecuencia, provocan la pérdida de clases, actividades académicas o tiempos relevantes de estudio. Esta situación no solo genera ineficiencia operativa, sino que también impacta negativamente la experiencia estudiantil y la percepción del servicio médico institucional.

En el contexto mexicano, esta problemática se relaciona con un fenómeno ampliamente documentado: la saturación de los servicios de salud. De acuerdo con los resultados de la ENSANUT Continua 2023, los tiempos de espera en instituciones públicas promedian entre 69 y 85 minutos, lo cual valida ampliamente que más del 40% de los usuarios enfrentan esperas superiores a media hora. Esta ineficiencia ha provocado que un alto porcentaje de usuarios jóvenes migre a servicios privados de bajo costo donde la espera promedia 20 minutos.

Los usuarios finales del sistema desarrollado son principalmente estudiantes de la UDLAP, aunque el diseño contempla también la participación del personal administrativo y médico, quienes requieren herramientas para gestionar citas, verificar disponibilidad y supervisar situaciones como sobrecarga del servicio o falta de recursos médicos. La aplicación integra módulos para cada tipo de usuario, permitiendo agendar citas, consultar horarios disponibles, registrar urgencias, reportar insuficiencia de recursos y visualizar el historial de atenciones registradas.

La solución propuesta consiste en un conjunto de ventanas, desarrollada en Java con almacenamiento en SQLite, que permite optimizar la organización del servicio médico universitario. El sistema facilita la programación de citas sin necesidad de acudir físicamente a la enfermería, integra un flujo para evaluar la urgencia del caso, incorpora un chatbot básico para consultas rápidas, y ofrece al personal médico herramientas para gestionar la disponibilidad de recursos y evitar la saturación del servicio. Con ello, se busca reducir tiempos de espera, mejorar la asignación de horarios, disminuir interrupciones académicas y proporcionar un mecanismo más eficiente y accesible de atención para la comunidad estudiantil.

2. Casos de Uso UML

2.1. Caso de uso 1: Registrar paciente en modo sobrecarga (Ricardo)

Actor primario: Personal de servicios médicos / Administrador

Descripción breve: Permite registrar de forma rápida a pacientes durante periodos de alta demanda (sobrecarga), capturando sus datos básicos, síntomas, prioridad y si serán atendidos en campus y/o referidos a hospital, para que queden almacenados en la base de datos.

Precondiciones:

- El usuario se ha autenticado como administrador o personal autorizado.
- El sistema está operativo y con conexión a la base de datos.

- Se ha determinado que la clínica está en un contexto de alta demanda (sobrecarga) según los criterios internos del servicio médico.

Postcondiciones:
- Los datos del paciente en contexto de sobrecarga quedan registrados en la base de datos.
- La información de prioridad y posible derivación (campus/hospital) queda asociada al registro.

Flujo principal:
1. El actor abre el panel de administración de servicios médicos.
2. El sistema muestra las opciones disponibles para el administrador.
3. El actor selecciona la opción “Modo Sobrecarga”.
4. El sistema despliega la ventana para registrar pacientes en sobrecarga.
5. El actor introduce el nombre del paciente.
6. El actor describe los síntomas principales en el campo correspondiente.
7. El actor selecciona el nivel de prioridad en la lista de opciones (por ejemplo, baja, media, alta).
8. El actor indica si el paciente será atendido en campus y/o se considera su envío a hospital (mediante casillas de selección).
9. El actor confirma el registro.
10. El sistema valida que los campos obligatorios no estén vacíos.
11. El sistema guarda el registro en la base de datos.
12. El sistema muestra un mensaje de confirmación de registro exitoso.

Flujos alternos y excepciones:
A1. Campos incompletos o inválidos
- Si falta el nombre, los síntomas o no se ha seleccionado ninguna acción (campus/hospital), el sistema muestra un mensaje de error.
- El sistema solicita completar la información y no permite guardar hasta corregir los datos
- El flujo continúa en el paso 9.

A2. Error de conexión o de base de datos
- Si ocurre un error al guardar la información, el sistema muestra un mensaje de error al usuario.
- El actor puede intentar guardar de nuevo o cancelar la operación.

2.2. Caso de uso 2: Consultar chatbot de orientación médica (Alberto)

Actor primario: Alumno

Descripción breve: Permite que el alumno reciba orientación básica sobre sus síntomas y dudas generales relacionadas con los servicios médicos mediante un chatbot con opciones predefinidas, que también puede sugerir agendar una cita.

Precondiciones:

- El alumno se ha autenticado en el sistema con su ID y contraseña.
- El alumno ha accedido al panel de servicios médicos como usuario tipo “Alumno”.

Postcondiciones:

- El alumno ha recibido recomendaciones generales según la opción seleccionada.
- Se registra de forma simple (en archivo de texto) una traza de la conversación con el chatbot, si así lo contempla el sistema.

Flujo principal:

1. El alumno inicia sesión en el sistema.
2. El sistema verifica sus credenciales y muestra el panel de alumno.
3. El alumno selecciona la opción “Chatbot”.
4. El sistema abre la ventana del chatbot y muestra un mensaje inicial de bienvenida.
5. El sistema presenta al alumno un conjunto de opciones de temas o síntomas (por ejemplo: fiebre, dolor de cabeza, náuseas, información general, etc.).
6. El alumno selecciona el síntoma o tema que más se aproxima a su situación.
7. El sistema responde con una explicación breve y recomendaciones generales (por ejemplo, reposo, hidratación, observar evolución del síntoma).
8. Si el caso lo amerita, el sistema sugiere al alumno utilizar el módulo de “Agendar Cita” para una valoración presencial.
9. Opcionalmente, el sistema guarda un registro del intercambio en un archivo de texto.
10. El alumno puede volver al menú principal del chatbot o cerrar la ventana.

Flujos alternos y excepciones:

B1. El alumno elige más información

- Si el alumno elige una opción del tipo “Sí, más información”, el sistema proporciona detalles adicionales sobre el síntoma.
- El sistema vuelve a ofrecer la posibilidad de regresar al inicio o cerrar el chatbot.

B2. Cierre del chatbot

- Si el alumno cierra la ventana en cualquier momento, el caso de uso finaliza sin afectar otros módulos del sistema.

2.3. Caso de uso 3: Agendar cita médica en servicios médicos UDLAP (Jeronimo)

Actor primario: Alumno

Descripción breve: Permite al alumno registrar sus datos básicos (nombre, edad, peso) y seleccionar un horario disponible para una cita médica, de forma que se reduzcan esperas innecesarias y se optimice el uso del servicio médico.

Precondiciones:

- El alumno se ha autenticado correctamente en el sistema.
- El módulo de gestión de horarios y la base de datos están disponibles.

- Existen horarios disponibles registrados en el sistema.

Postcondiciones:

- La cita del alumno queda registrada en la base de datos con sus datos personales y el horario seleccionado.
- El horario elegido queda marcado como ocupado para posteriores solicitudes.

Flujo principal:

1. El alumno inicia sesión en el sistema y accede al panel de alumno.
2. El alumno selecciona la opción “Agendar Cita”.
3. El sistema muestra una ventana para capturar el nombre, la edad y el peso del alumno.
4. El alumno introduce su nombre.
5. El alumno introduce su edad.
6. El alumno introduce su peso.
7. El alumno confirma que la información es correcta y continúa.
8. El sistema valida que los campos tengan un formato correcto (que no estén vacíos y que edad/peso sean valores válidos).
9. Si los datos son válidos, el sistema abre la ventana de selección de horario y muestra los horarios disponibles.
10. El alumno selecciona un horario de la lista.
11. El alumno confirma la reserva de la cita.
12. El sistema verifica que el horario siga disponible.
13. El sistema registra la cita en la base de datos, asociando el alumno y el horario.
14. El sistema muestra un mensaje de confirmación indicando que la cita se ha agendado correctamente.

Flujos alternos y excepciones:

C1. Datos personales incompletos o inválidos

- Si algún campo está vacío o tiene un formato incorrecto, el sistema muestra un mensaje de error.
- El sistema solicita corregir los datos y no permite continuar hasta que se capture información válida.
- El flujo regresa al paso 4.

C2. Horario ya ocupado

- Si otro usuario reservó el mismo horario justo antes de la confirmación, el sistema indica que dicho horario ya no está disponible.
- El sistema regresa a la lista de horarios disponibles para que el alumno elija otro.
- El flujo continúa desde el paso 10.

C3. Error de base de datos

- Si ocurre un error al guardar la cita, el sistema muestra un mensaje informando el problema.
- El alumno podrá reintentar la operación o cancelar el proceso de agendar cita.

2.4. Caso de uso 4: Gestionar recursos médicos en falta (Marcos)

Actor primario: Personal de servicios médicos / Administrador

Descripción breve: Permite al personal de servicios médicos consultar los recursos médicos (medicamentos, insumos, equipo) que están por debajo de su nivel mínimo y registrar acciones o solicitudes relacionadas con dichos faltantes.

Precondiciones:

- El usuario se ha autenticado como administrador o personal autorizado.
- Existe un inventario inicial de recursos médicos registrado en el sistema.

Postcondiciones:

- Se genera un registro en la base de datos indicando qué recursos se han identificado como faltantes y las acciones que se tomarán.
- La información queda disponible para seguimiento o futuras consultas.

Flujo principal:

1. El actor accede al panel de administración de servicios médicos.
2. El actor selecciona la opción “Solicitar Recursos”.
3. El actor consulta el inventario de recursos y determina cuáles están por debajo de su cantidad mínima.
4. El sistema muestra una tabla con los recursos, incluyendo tipo, nombre y severidad.
5. El actor selecciona un recurso de la lista.
6. El actor redacta notas o comentarios sobre la situación (por ejemplo, “Solicitar reposición a farmacia central”).
7. El actor marca, si corresponde, una casilla de validación del reporte.
8. El actor guarda la acción registrada.
9. El sistema valida que se haya seleccionado al menos un recurso y, si así se define, que exista alguna nota.
10. El sistema registra en la base de datos la información del recurso en falta y la acción tomada.
11. El sistema muestra un mensaje de confirmación.

Flujos alternos y excepciones:

D1. No se selecciona ningún recurso

- Si el actor intenta guardar sin seleccionar un recurso, el sistema muestra un mensaje de advertencia.
- El sistema solicita seleccionar al menos un recurso.
- El flujo regresa al paso 5.

D2. Error al guardar

- Si ocurre un error al registrar la acción en la base de datos, el sistema informa al actor.
- El actor puede reintentar o cerrar la ventana.

2.5. Caso de uso 5: Acceder a opciones de urgencias (Brian)

Actor primario: Alumno

Descripción breve: Permite al alumno acceder rápidamente a una interfaz que le muestra opciones de urgencias, distinguiendo entre contacto con servicios médicos del campus y servicios de emergencia externa (simulados mediante mensajes).

Precondiciones:

- El alumno ha iniciado sesión correctamente en el sistema.
- El módulo de urgencias está disponible en el panel del alumno.

Postcondiciones:

- El alumno ha recibido información sobre las opciones de contacto para casos de urgencia.
- No se registran datos clínicos ni se genera una cita automáticamente; la funcionalidad se limita a orientar al alumno sobre a quién contactar.

Flujo principal:

1. El alumno inicia sesión en el sistema.
2. El sistema muestra el panel de alumno.
3. El alumno selecciona la opción “Urgencias”.
4. El sistema muestra una ventana con al menos dos opciones:
5. Servicios Médicos UDLAP
6. Emergencia 911
7. El alumno selecciona la opción que considere adecuada.
8. El sistema muestra un mensaje con la acción simulada o la información de contacto correspondiente (por ejemplo, número telefónico o indicación de llamada).
9. El alumno puede cerrar la ventana de urgencias o regresar al panel principal.

Flujos alternos y excepciones:

E1. Cierre de la ventana

- En cualquier momento, si el alumno cierra la ventana, el caso de uso termina sin que se ejecuten acciones adicionales en el sistema.

2.6. Diagramas de Casos de Uso

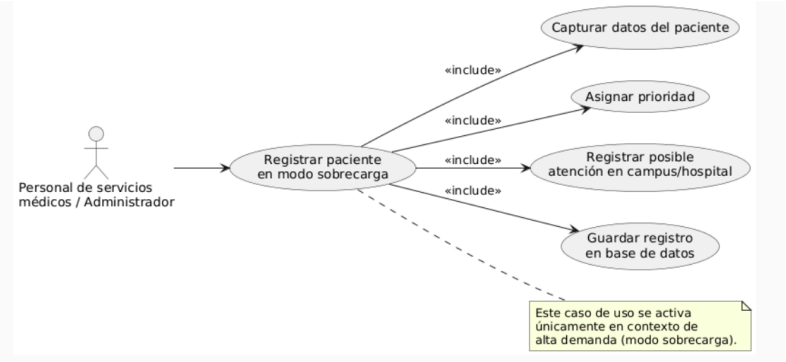


Figura 1. Caso de uso 1: Registrar paciente en modo sobrecarga

Figura 2. Caso de uso 2: Consultar chatbot de orientación médica

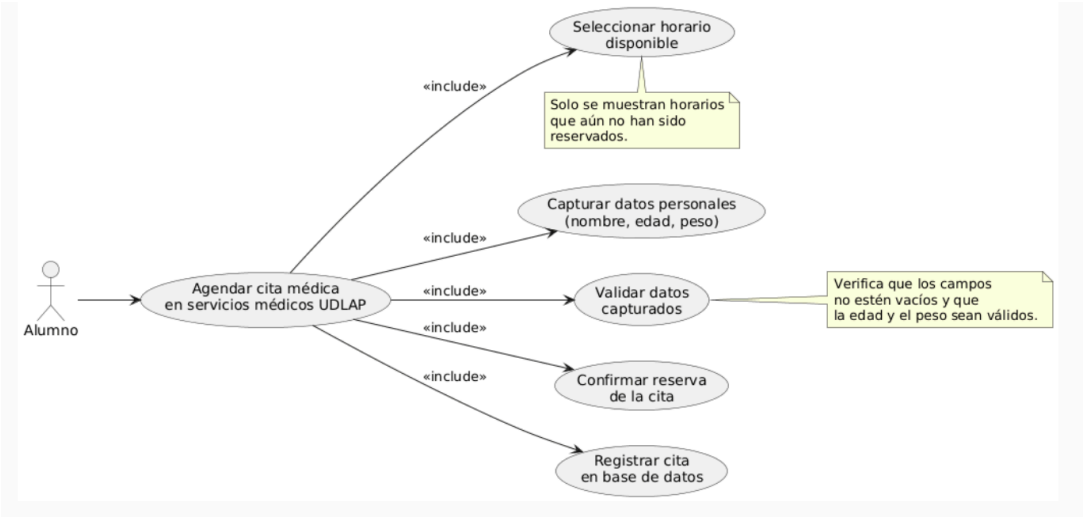


Figura 3. Caso de uso 3: Agendar cita médica en servicios médicos UDLAP

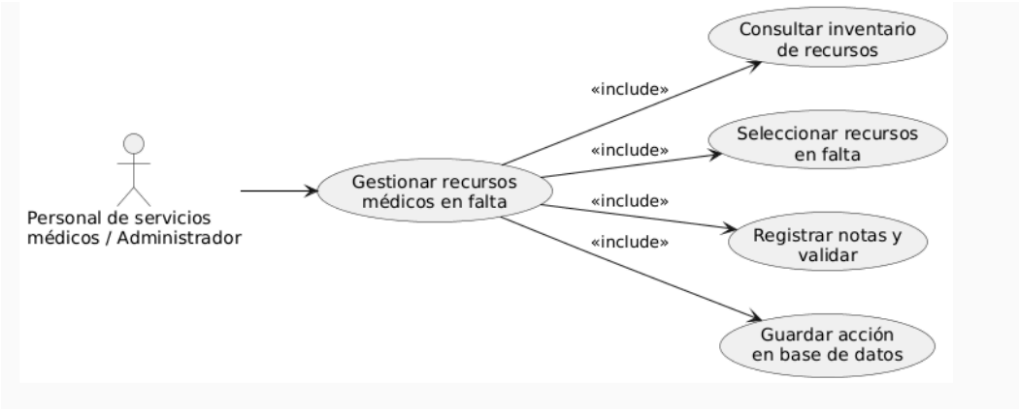


Figura 4. Caso de uso 4: Gestionar recursos médicos en falta

Figura 5. Caso de uso 5: Acceder a opciones de urgencias

3. Diagramas de Actividades

Los siguientes escenarios generados mediante IA se presentan como parte del proceso creativo inicial. Sin embargo, el análisis y los diagramas de actividades que se muestran posteriormente han sido ajustados para reflejar con precisión la implementación final del sistema.

3.1. Caso de uso 1: Registrar paciente en modo sobrecarga

3.1.1 Escenario generado por ChatGPT

Entrevista con el Administrador del Sistema de Salud:

Pregunta: ¿Qué sucede cuando la universidad enfrenta una crisis sanitaria, como una pandemia?

Respuesta: Cuando se activa el "Modo Sobrecarga" en el sistema, el Administrador del sistema inicia un protocolo de emergencia. Primero, el sistema verifica la disponibilidad de recursos, como personal médico y equipos. En este punto, es clave que el sistema maneje la capacidad local, ya que puede haber una falta de recursos en momentos de alta demanda.

Pregunta: ¿Cómo maneja el sistema a los pacientes que llegan al Servicio Médico?

Respuesta: Los pacientes llegan o se registran en el sistema con sus datos y síntomas. El sistema, de manera automatizada, realiza un triage para clasificar la urgencia de cada caso. Basado en esta información, el sistema determina si el paciente puede ser atendido en el campus o si necesita ser derivado a un hospital.

Pregunta: ¿Qué ocurre si hay una falta de personal o de equipos disponibles para la atención?

Respuesta: Si el sistema detecta que no hay suficiente personal o espacio disponible, el paciente es priorizado y, si es necesario, se generan derivaciones a hospitales cercanos. El sistema contacta a los hospitales asociados, confirma la disponibilidad y organiza el traslado del paciente.

Pregunta: ¿Cómo se gestionan los problemas, como fallos del sistema o la falta de insumos médicos?

Respuesta: En caso de falla del sistema, el Administrador del sistema puede activar un modo manual de registro. Si hay desabasto de insumos, el sistema notifica inmediatamente a los responsables para que tomen acciones correctivas. Además, las derivaciones se mantienen registradas y actualizadas en la base de datos.

Pregunta: ¿Cómo se asegura la comunicación entre el sistema y los hospitales externos?

Respuesta: El sistema tiene integración directa con los hospitales asociados. Cada derivación es confirmada por el hospital antes del traslado, y el sistema envía notificaciones al personal médico correspondiente para que puedan seguir el estado del paciente.

3.1.2 Análisis del escenario

Pasos clave:

- Activación del modo sobrecarga.
- Registro de paciente y síntomas.
- Triage automatizado.
- Verificación de recursos.

- Decisión: atención en campus o derivación.
- Asignación y registro.

Decisiones:

- ¿El paciente requiere prioridad alta?
- ¿Hay recursos disponibles?
- ¿Se debe derivar a hospital?

Errores:

- El sistema no realiza triage automatizado.
- No valida disponibilidad de personal o consultorios.
- No está implementada ninguna comunicación real con hospitales.
- No se generan reportes automáticos de sobrecarga.
- El registro sí existe, pero es simple, solo guarda síntomas, prioridad y opciones.
- El escenario menciona múltiples roles que no existen en el sistema.

3.1.3 Diagrama de Actividades

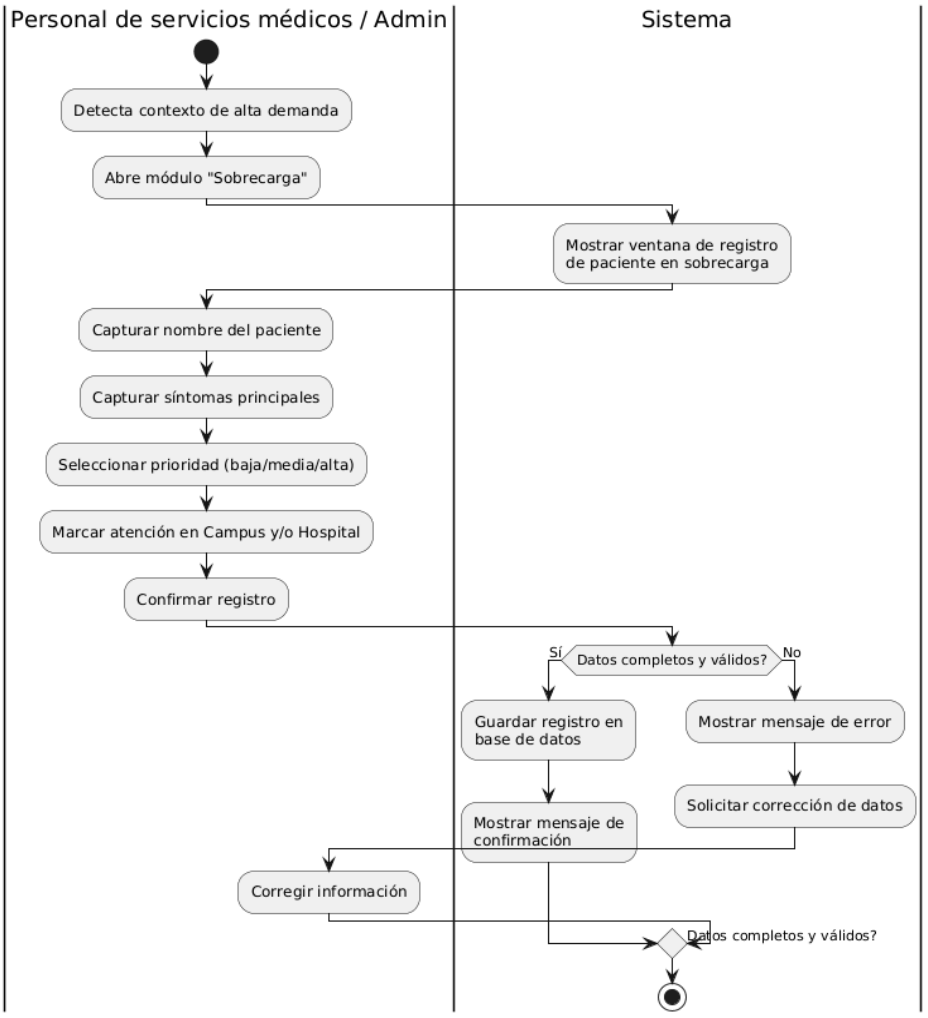


Figura 6. Diagrama de Actividades del Caso de Uso 1

3.2. Caso de uso 2: Consultar chatbot de orientación médica

3.2.1 Escenario generado por ChatGPT

1) El estudiante ingresa con su cuenta institucional y abre “Autoevaluar síntomas”.
2) El sistema pregunta: ¿Cuál es tu síntoma principal? (dolor de garganta). ¿Desde cuándo? (2 días). ¿Intensidad? (3/10). ¿Fiebre medida? (no). ¿Dificultad para respirar? (no). Comorbilidades? (ninguna).
3) El sistema valida respuestas obligatorias y ofrece ayuda contextual (ej.: cómo medir temperatura).
4) Se detectan señales de alarma: ninguna. El sistema clasifica el caso como leve y despliega recomendaciones (líquidos, reposo, paracetamol 500 mg c/8h si no hay alergias, evitar automedicación antibiótica, aislamiento si hay síntomas respiratorios).
5) Ofrece descargar el plan en PDF y programar recordatorios. Pregunta si desea agendar una cita; sugiere no necesaria salvo empeoramiento o persistencia >72 h.
6) Si la red falla o el cuestionario se corta, el sistema guarda progreso y permite retomar sin pérdida.
7) El estudiante finaliza, guarda su historial y cierra sesión

3.2.2 Análisis del escenario

Pasos clave:

- Ingreso al sistema.
- Selección de sección “Autoevaluar síntomas”.
- Contestación de cuestionario médico.
- Evaluación automática del caso.
- Clasificación leve/moderado/grave.
- Recomendaciones y sugerencias.

Decisiones:

- ¿Cuestionario completo?
- ¿Caso grave?
- ¿Se debe recomendar atención médica inmediata?

Errores:

- El chatbot no usa un cuestionario clínico; solo muestra opciones.
- No evalúa gravedad real ni clasifica niveles clínicos.
- No existe análisis médico ni inteligencia clínica.
- No restringe la reserva de cita por no haber completado evaluación.
- No manda alertas automáticas al personal médico.
- El registro es solo un archivo de texto básico.

3.2.3 Diagrama de Actividades

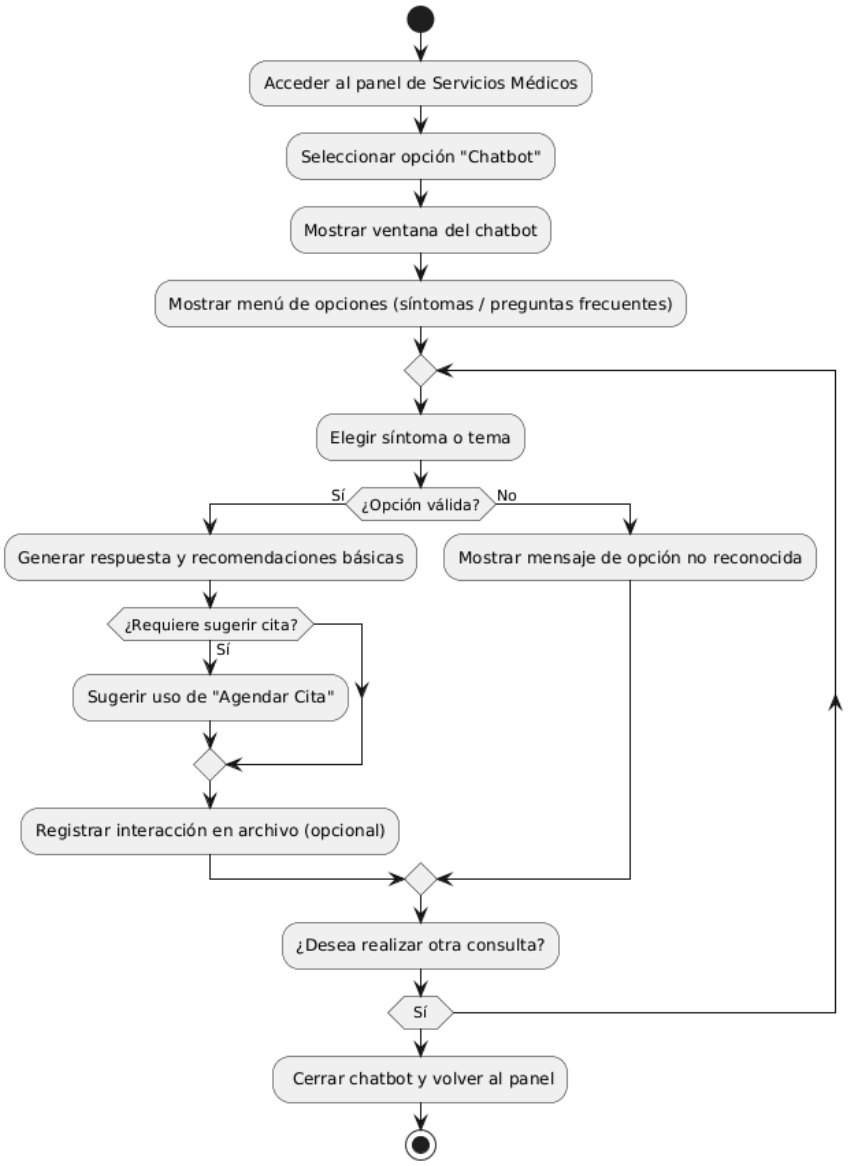


Figura 7. Diagrama de Actividades del Caso de Uso 2

3.3. Caso de uso 3: Agendar cita médica en servicios médicos UDLAP

3.3.1 Escenario generado por ChatGPT

Pregunta: ¿Qué haces primero cuando decides reservar una cita médica en el sistema?
Respuesta: Abro la aplicación desde mi celular e inicio sesión con mi correo institucional y contraseña. El sistema me pide realizar una autoevaluación rápida de síntomas antes de continuar.
Pregunta: ¿Qué sucede después de la autoevaluación?
Respuesta: El sistema me indica si soy candidato para una consulta general. Si no paso la autoevaluación, me da recomendaciones y cierra la solicitud. Si paso, me permite avanzar a la sección de “Reservar cita”.

Pregunta: ¿Cómo eliges tu cita médica?
Respuesta: Selecciono la especialidad de “Medicina General”, después escojo la fecha y la hora que más me conviene. El sistema consulta la disponibilidad de médicos y me muestra varias opciones con nombre del doctor, consultorio y horarios.
Pregunta: ¿Qué pasa si no hay disponibilidad en el horario que quieres?
Respuesta: El sistema me sugiere horarios alternativos o incluso otro médico disponible. Yo elijo una nueva opción y vuelvo a confirmar.
Pregunta: ¿Cómo se confirma la cita y cómo sabes que fue registrada?
Respuesta: El sistema me pide confirmar los datos y aceptar el aviso de privacidad. Una vez que confirmo, se genera un folio con mi ID de paciente y recibo un correo y un SMS con un QR para registrar mi llegada el día de la consulta.
Pregunta: ¿Y si el día de la cita no llegas tú o no llega el doctor?
Respuesta: Si yo no llego, el sistema me ofrece reprogramar para otro día. Si no llega el médico, el sistema también reprograma automáticamente y me notifica de los cambios.

3.3.2 Análisis del escenario

Pasos clave:

- Inicio de sesión.
- Autoevaluación previa.
- Selección de cita.
- Verificación de médico disponible.
- Confirmación.
- Envío de recordatorio.
- Registro de llegada.

Decisiones:

- ¿Rellenado de datos completado?
- ¿Horario disponible?
- ¿Paciente asiste o falta?

Errores:

- No hay selección de médico específico.
- No existe envío de recordatorios.
- No se controla asistencia ni llegada del paciente.

3.3.3 Diagrama de Actividades

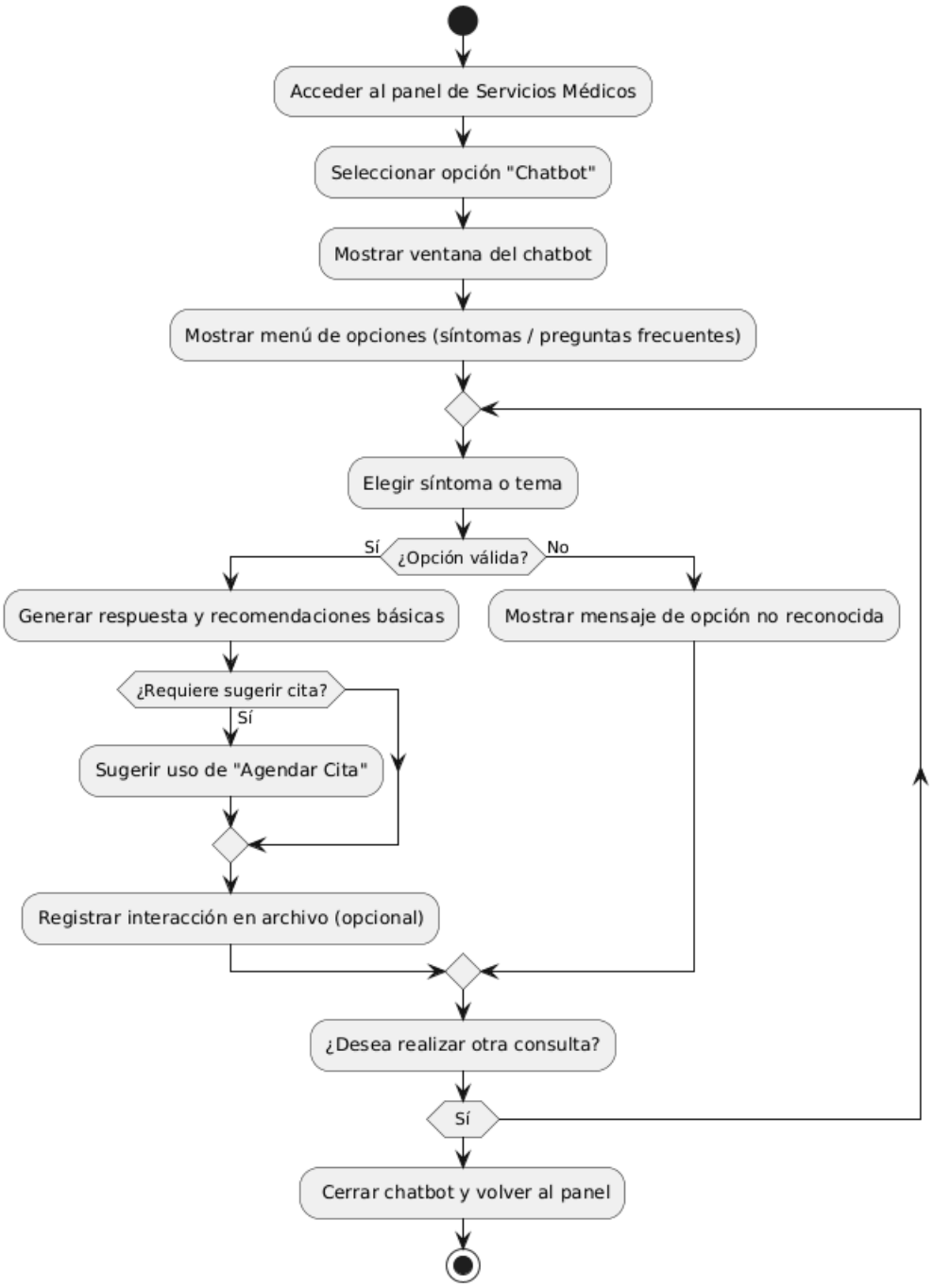


Figura 8. Diagrama de Actividades del Caso de Uso 3

3.4. Caso de uso 4: Gestionar recursos médicos en falta

3.4.1 Escenario generado por ChatGPT

Inicio sesión con mis credenciales y rol de Coordinador(a) de Triage. El sistema abre el Panel Clínico con indicadores: pacientes en espera, ocupación de camillas, disponibilidad de personal por turno y stock crítico de farmacia.

Llega un paciente con dolor abdominal. Registro su Solicitud de Atención (datos personales, motivo, triage preliminar). El sistema ejecuta de inmediato la Verificación de Recursos: (1) medicamentos requeridos por protocolo, (2) camilla en sala de urgencias, (3) médico general disponible.
Caso A – Recursos suficientes: El tablero muestra "disponible". Confirmo la Asignación: camilla #U-12, médico de guardia y set básico de medicamentos. El sistema: (a) bloquea la camilla, (b) descuenta inventario y (c) crea la Orden de Atención. Continúo con la atención; al finalizar, registro evolución, cierro la orden y el sistema actualiza inventario y libera la camilla.
Caso B – Falta de medicamentos: La verificación marca "Amoxicilina 500 mg – No disponible". El sistema muestra alerta y despliega sustitutos autorizados según guía farmacoterapéutica (p. ej., cefalexina). Solicito validación del médico responsable con un clic en "Enviar a validación". El médico aprueba en su bandeja; el sistema registra quién, cuándo y motivo. Se procede con el sustituto; el inventario se descuenta y se etiqueta la atención como "Con sustitución".
Caso C – Falta de camillas: La ocupación está al 100%. El sistema propone derivar al área de observación o a Hospital Asociado A. Reviso tiempos de traslado, confirmo con el Supervisor y genero Orden de Derivación. Se notifica al hospital receptor y al paciente/familiar. El sistema deja trazas: confirmación de recepción y hora estimada.
Caso D – Falta de personal: El sistema alerta "Sin médico disponible en 10 min". Notifica a Supervisión que reconfigura la guardia o activa Protocolo de Contingencia. Mientras, registro al paciente en lista de priorización; el sistema recalcula el tiempo estimado.
Caso E – Falla del sistema de inventario: Aparece "Sin conexión al módulo de inventario". Activo Modo Manual: capturo recursos utilizados (formulario offline con validación mínima), tomo evidencia (foto de etiqueta de lote) y guardo. El sistema marca el caso como "Pendiente de sincronización". Al restablecerse, reconcilia: descuenta stock y cierra la alerta.
Desabasto prolongado: Tras tres incidencias consecutivas en 24 h del mismo medicamento, el sistema genera Reporte de Desabasto a Autoridades Universitarias y Proveedores, adjunta historial, consumo promedio y propone pedido urgente.
Cierre de turno: Reviso Bitácora: alertas atendidas, derivaciones, sustituciones y pendientes de sincronización. Firmo digitalmente el Cierre de Turno; el sistema consolida métricas y envía resumen automático a la jefatura.

3.4.2 Análisis del escenario

Pasos clave:

- Ingreso del personal.
- Verificación de recursos requeridos.
- Estado del inventario.
- Solicitud a proveedores.
- Cierre y registro.

Decisiones:

- ¿Recurso disponible?

- ¿Existe sustituto?
- ¿Está validado?
- ¿Reposición automática o manual?

Errores:

- El sistema no gestiona proveedores reales.
- No existe reposición automática.
- No manda alertas a dirección, rectoría ni seguridad.
- No comunica inventario con hospitales asociados.

3.4.3 Diagrama de Actividades

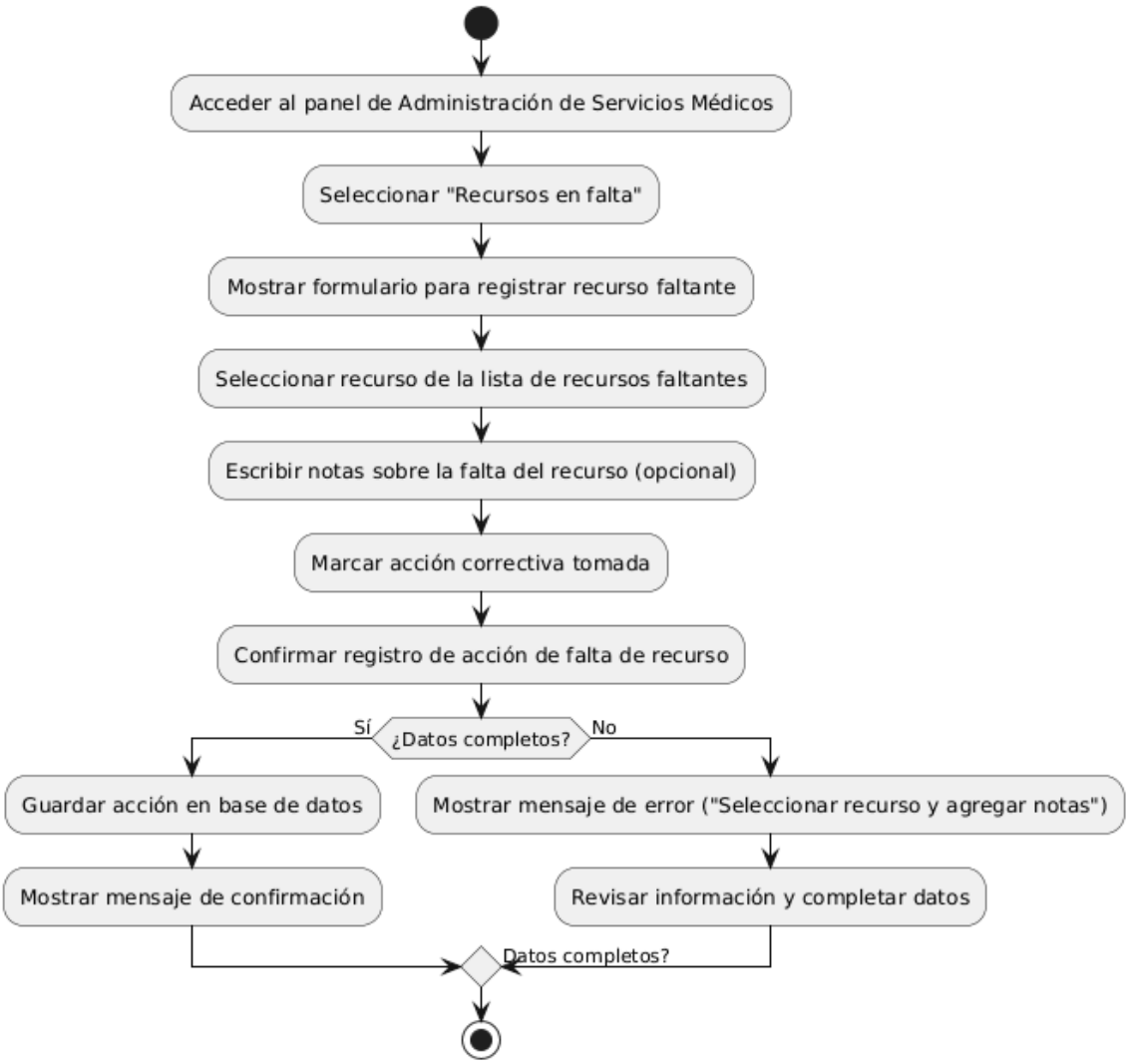


Figura 9. Diagrama de Actividades del Caso de Uso 4

3.5. Caso de uso 5: Acceder a opciones de urgencias

3.5.1 Escenario generado por ChatGPT

1. Durante un partido en el gimnasio, un estudiante sufre mareo y cae al suelo.
2. Un compañero presiona el botón de pánico en la app y Seguridad del campus llama al número 24/7.
3. El CASU registra el incidente, lo clasifica como URGENCIA y geolocaliza el punto exacto.
4. El sistema verifica recursos y asigna la brigada/paramédico más cercano; se notifica a Seguridad y Enfermería.
5. Seguridad abre accesos y guía al equipo por la ruta más rápida.
6. La brigada brinda atención inmediata en sitio y decide trasladar al módulo médico del campus.
7. Si la sala del módulo está ocupada, el CASU coordina ambulancia y traslado a hospital de convenio.
8. Todas las intervenciones y responsables se registran (hora, lugar, paciente, signos, medicamentos).
9. El sistema actualiza en tiempo real la disponibilidad de personal, camillas y salas; se genera reporte para seguimiento

3.5.2 Análisis del escenario

Pasos clave:

- Reporte de emergencia.
- Asignación y despacho.
- Llegada de brigada.
- Atención en sitio.
- Derivación.
- Registro del caso.

Decisiones:

- ¿Hay brigada disponible?
- ¿Es necesaria una ambulancia?
- ¿Espacios libres en módulos médicos?

Errores:

- El sistema no maneja brigadas reales.
- No incluye geolocalización ni monitoreo en tiempo real.
- No llama automáticamente a ninguna instancia.
- No evalúa gravedad del incidente.
- La interfaz solo muestra dos botones: “Servicios Médicos UDLAP” y “911”

3.5.3 Diagrama de Actividades

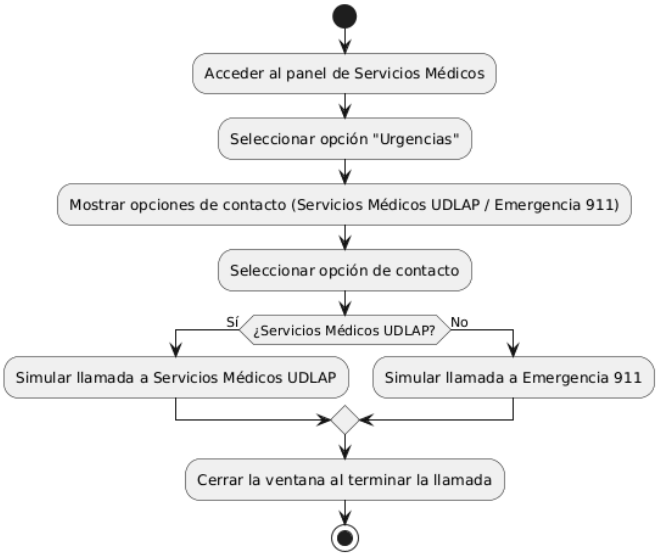


Figura 10. Diagrama de Actividades del Caso de Uso 5

3.6 Diagrama de actividades completo

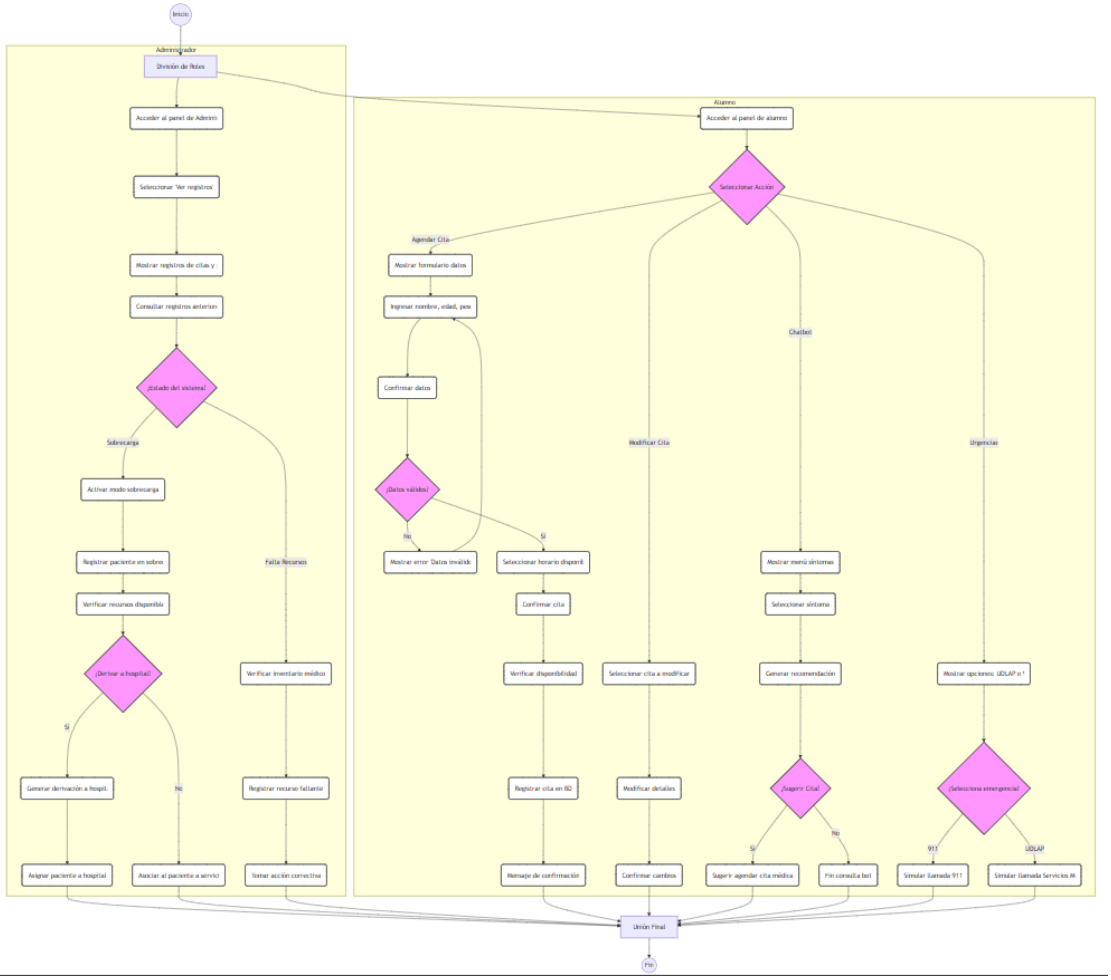


Figura 11. Diagrama de Actividades Completo

3.7 Reflexión sobre el uso de IA

El uso de herramientas de inteligencia artificial, como ChatGPT, resultó fundamental para el diseño y depuración de los diagramas de actividades de nuestro proyecto. En una primera etapa, empleamos la IA para generar escenarios narrativos detallados a partir de entrevistas simuladas con distintos actores (administrador del sistema, estudiantes, personal médico, etc.). Esto permitió visualizar con mayor claridad el contexto de uso del sistema y los puntos críticos de decisión, más allá de lo que normalmente hubiéramos imaginado solo desde la programación.

Posteriormente, utilizamos esos escenarios como base para extraer los pasos clave, las decisiones y las excepciones, y con apoyo de la IA fuimos simplificando y estructurando los flujos en diagramas de actividades con carriles. Este proceso nos ayudó a distinguir entre lo que el sistema realmente implementa y las funcionalidades "ideales" que aparecían en los escenarios generados. Gracias a ello, pudimos ajustar cada diagrama para que estuviera completo, pero al mismo tiempo alineado con el código final de la aplicación.

Desde el punto de vista del aprendizaje, la IA funcionó como un apoyo para organizar ideas, detectar pasos redundantes y proponer alternativas de modelado (por ejemplo, cuándo usar bifurcaciones, cuándo agrupar actividades o cómo representar mejor los roles en swimlanes). Sin embargo, también fue necesario revisar críticamente las propuestas, ya que en ocasiones los escenarios sugeridos por la IA suponían capacidades del sistema que todavía no existen en nuestra implementación.

En conclusión, el uso de IA aceleró el diseño de los flujos, enriqueció la comprensión del problema y nos obligó a tomar decisiones de diseño más informadas. No sustituye el criterio del equipo, pero sí se convierte en una herramienta útil para explorar, comparar y refinar modelos de procesos en el contexto de la Programación Orientada a Objetos.

4. Modelo de Clases

4.1 Interfaces de Navegación y Acceso

Clase: VentanaGeneral			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
txtID	JTextField	private	Campo para ingresar ID institucional.
txtContra	JTextField	private	Campo para ingresar contraseña.
IniciarButton	JButton	private	Botón que dispara la validación de credenciales.
warningLabel	JLabel	private	Muestra errores si el login falla.

Clase: VentanaAlumno			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
alumno	Alumno	public	Objeto con datos del alumno logueado.
AgendarButton	JButton	private	Abre la ventana AgendarCita.
ModificarButton	JButton	private	Abre la ventana ModificarCita.
ChatButton	JButton	private	Abre la ventana Chatbot.
UrgenciaButton	JButton	private	Abre la ventana Urgencias.

Clase: VentanaAdmin			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
RecursosButton	JButton	private	Abre la gestión de FaltaRecursos.
SobrecargaButton	JButton	private	Abre el modo Sobrecarga.
verRegistrosBtn	JButton	private	Abre el visor de registros.

4.2 Módulos Funcionales

Clase: AgendarCita			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
alumno	Alumno	private	Objeto con la información del usuario actual.
txtNombre	JTextField	private	Campo de texto para el nombre.
txtEdad	JTextField	private	Campo de texto para la edad.

txtPeso	JTextField	private	Campo de texto para el peso (kg).
btnContinuar	JButton	private	Botón que valida datos y avanza a la selección de horario.
warningLabel	JLabel	private	Etiqueta oculta para mostrar mensajes de error.

Clase: ModificarCita			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
alumno	Alumno	private	Referencia al usuario actual.
comboNombres	JComboBox	private	Lista desplegable con nombres de citas registradas.
txtEdad	JTextField	private	Campo para editar la edad.
txtPeso	JTextField	private	Campo para editar el peso.
comboHorarios	JComboBox<String>	private	Lista desplegable con los nuevos horarios disponibles.
btnGuardar	JButton	private	Botón para guardar los cambios en la BD.
btnCancelarCita	JButton	private	Botón para eliminar la cita de la BD.

Clase: Chatbot			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
taChat	JTextArea	private	Área donde se muestra el historial de la conversación.
btnReportes	JButton	private	Botón para abrir el archivo de texto con el historial.

panelOpciones	JPanel	private	Contenedor dinámico para los botones de respuesta.
nodes	Map<String, Node>	private	Estructura de datos (Mapa) que contiene el árbol de diálogo.

Clase: Sobrecarga			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
txtNombre	JTextField	private	Campo para el nombre del paciente.
txtSintomas	JTextArea	private	Área de texto para describir los síntomas.
comboPrioridad	JComboBox<String>	private	Selección de nivel urgencia (Alta/Media/Baja).
campusCheck	JCheckBox	private	Casilla para indicar atención en Campus.
hospitalCheck	JCheckBox	private	Casilla para indicar derivación a Hospital.
btnRegistrar	JButton	private	Guarda el registro de sobrecarga en la BD.

Clase: FaltaRecursos			
Atributo	**Tipo de Dato**	**Visibilidad**	**Descripción**
tablaRecursos	JTable	private	Tabla visual con la lista de insumos agotados.
txtNotas	JTextArea	private	Campo para añadir notas o acciones a tomar.
chkValidado	JCheckBox	private	Casilla de validación por autoridad médica.
btnGuardar	JButton	private	Botón para confirmar y guardar la acción.

btnCancelar	JButton	private	Cierra la ventana sin guardar.

5. Interfaces Gráficas

La interfaz de usuario ha sido desarrollada utilizando la biblioteca Java Swing, implementando un diseño basado en ventanas (JFrame) y paneles (JPanel) con gestores de distribución (LayoutManagers) como GridBagLayout y BoxLayout para asegurar una disposición ordenada y adaptable de los elementos. A continuación, se detalla el flujo de navegación, la descripción funcional de cada módulo y las validaciones implementadas.

5.1 Diagrama de Navegación

El siguiente diagrama ilustra el flujo de pantallas del sistema, partiendo desde el inicio de sesión y bifurcándose según el rol del usuario (Alumno o Administrador).

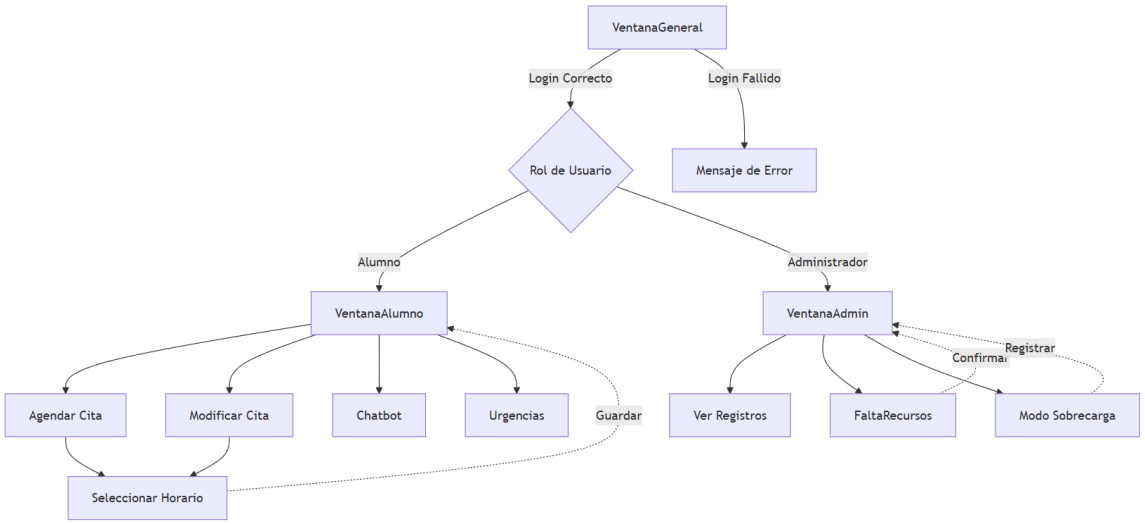


Figura 12. Diagrama de Navegación

5.2 Descripción Funcional y Validaciones

5.2.1 Ventana de Inicio de Sesión

Clase: VentanaGeneral

Descripción: Es el punto de entrada al sistema. Presenta un formulario minimalista solicitando credenciales institucionales (ID y Contraseña). El sistema identifica internamente si el ID corresponde a un alumno o a un administrador para dirigirlo a su panel correspondiente.

Figura 13. Inicio de sesión

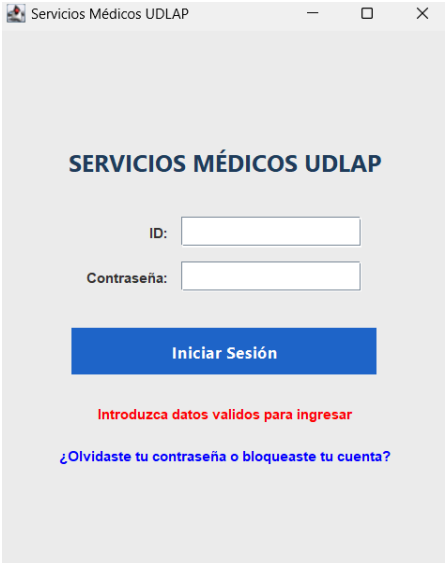


Figura 14. Inicio de sesión con advertencia

Flujos y Validaciones:

- Campos Vacíos: Si se intenta ingresar sin datos, el sistema muestra la etiqueta de advertencia "Introduzca datos válidos para ingresar".
- Credenciales Incorrectas: Si el ID no existe o la contraseña no coincide, se muestra "ID o contraseña incorrectos".
- Éxito: Cierra la ventana actual y abre VentanaAlumno o VentanaAdmin según corresponda.

5.2.2 Menú Principal del Alumno

Clase: VentanaAlumno

Descripción: Actúa como un Hub central para el estudiante. Dispone de cuatro botones grandes y claros para facilitar el acceso rápido a las funciones principales: Agendar, Modificar, Chatbot y Urgencias.

Figura 15. Menú Alumno

5.2.3 Módulo de Agendar Cita

Clase: AgendarCita y SeleccionarHorario

Descripción: Este proceso se divide en dos pasos. Primero, una ventana de registro captura los datos antropométricos del paciente (Nombre, Edad, Peso). Al validar estos datos, se despliega una segunda ventana para la selección del horario disponible.

Figura 16. Formulario de datos

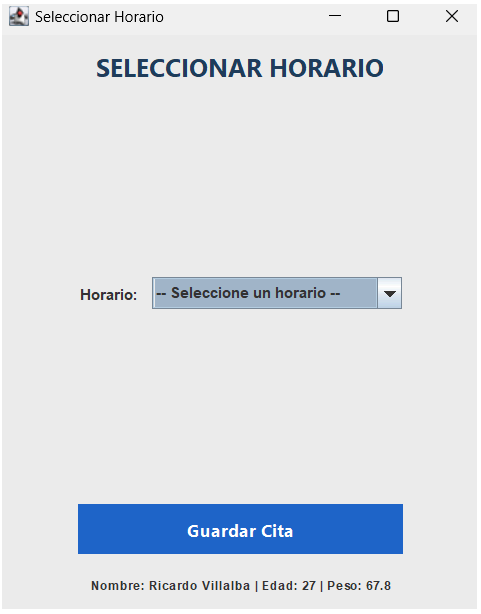


Figura 17. Selección de horario

Figura 18. Advertencia en Formulario

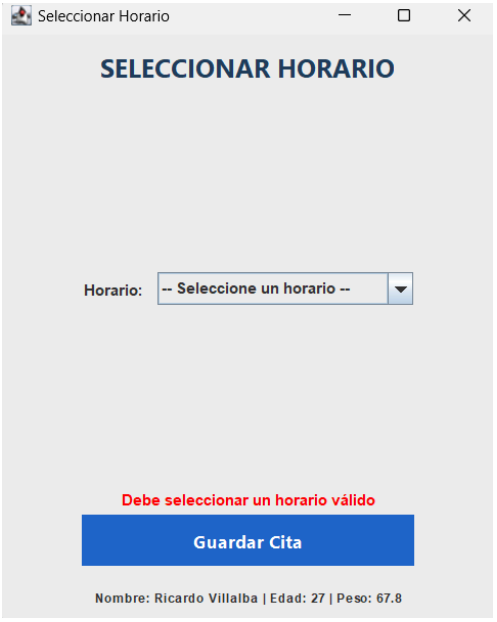


Figura 19. Advertencia en Selección de horario

Flujos y Validaciones:

- Tipos de Dato: La edad debe ser un número entero y el peso un número decimal (double). Si se ingresan letras, el sistema captura la excepción NumberFormatException y alerta al usuario.

- Rangos Lógicos: Se valida que la edad esté entre 0 y 120 años, y el peso entre 1 y 300 kg.
- Campos Obligatorios: El botón "Continuar" no avanza si hay campos vacíos.
- Disponibilidad: El combo de horarios filtra o valida que la hora seleccionada no esté ocupada en la base de datos.
- Cierra Agendar Cita y abre Seleccionar horario

5.2.4 Módulo de Modificar/Cancelar Cita

Clase: ModificarCita

Descripción: Permite al alumno gestionar una cita existente. Incluye un menú desplegable (JComboBox) para buscar la cita por nombre. Al seleccionarla, los campos se autocompletan con la información actual, permitiendo editar edad, peso u horario.

Figura 20. Ventana Modificar Cita

Figura 21. Advertencia en la ventana Modificar Cita

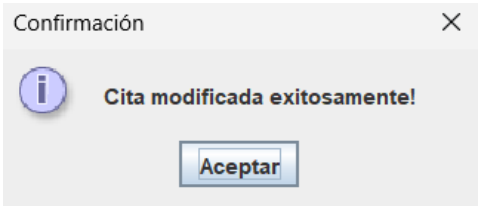


Figura 22. Confirmación de modificación

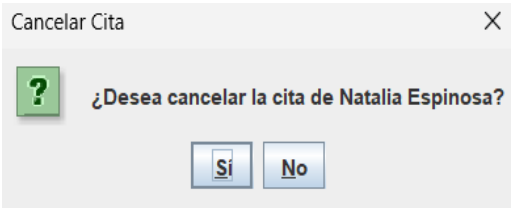


Figura 23. Cuestionar al Cancelar Cita

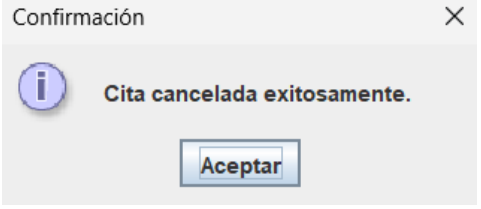


Figura 24. Confirmación de cancelación

Flujos y Validaciones:

- Actualización: El usuario cambia un dato y presiona "Guardar". El sistema verifica si el nuevo horario está libre antes de sobrescribir el registro.
- Cancelación: El botón rojo "Cancelar Cita" despliega un cuadro de diálogo (JOptionPane) de confirmación: "¿Desea cancelar la cita de …?". Si se confirma, el registro se elimina de SQLite.

5.2.5 Chatbot de Orientación

Clase: Chatbot

Descripción: Interfaz conversacional basada en botones. No requiere que el usuario escriba texto libre, sino que lo guía a través de un árbol de decisiones (Síntomas -> Fiebre -> Recomendación).

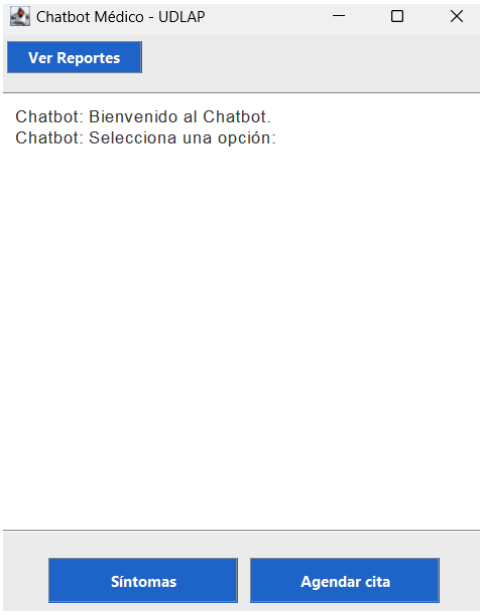


Figura 25. Ventana Chatbot

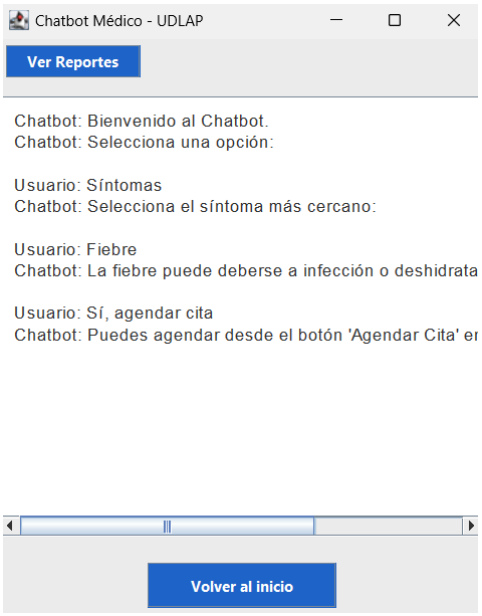


Figura 26. Interacción con el Chatbot

Flujos y Validaciones:

- Historial: Un área de texto (JTextArea) inhabilitada muestra la traza de la conversación.
- Persistencia: Al finalizar, se genera un archivo de texto (reportes.txt) con el historial de la sesión para futura referencia médica.

5.2.6 Módulo de Urgencias

Clase: Urgencias

Descripción: Pantalla de acceso rápido diseñada para situaciones críticas. Contiene dos botones de gran tamaño para evitar errores de precisión en momentos de estrés.

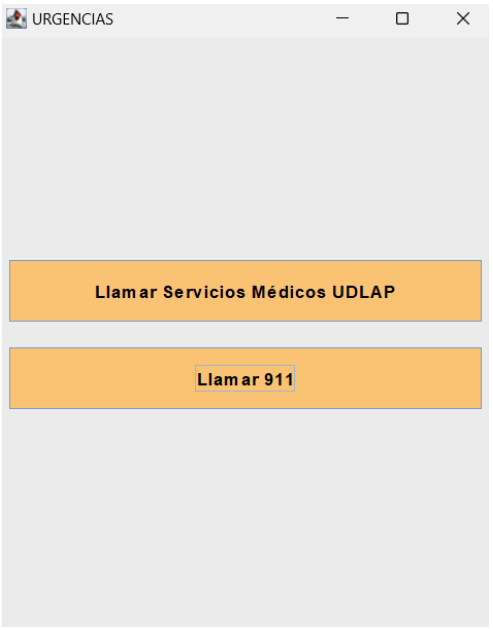


Figura 27. Ventana Urgencias

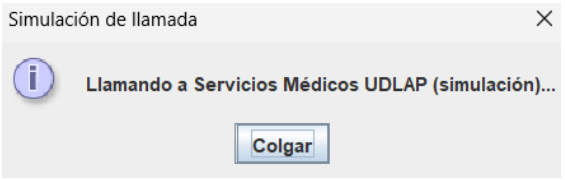


Figura 28. Simulación de llamada 1

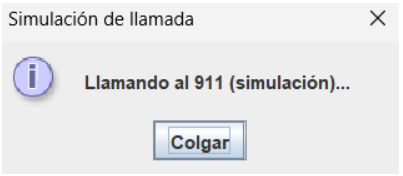


Figura 29. SImulación de llamada 2

Flujos y Validaciones:

- Botón Servicios Médicos UDLAP: Simula una llamada directa a la enfermería del campus.
- Botón 911: Simula el contacto con servicios de emergencia externos. Ambos despliegan alertas visuales confirmando la acción.

5.2.7 Menú Principal del Administrador

Clase: VentanaAdmin

Descripción: Funciona como el centro de control operativo para el personal médico y administrativo. A diferencia del menú del alumno, este panel se enfoca en la gestión del servicio y no en la atención personal. Su diseño es limpio y directo, presentando tres módulos clave para la toma de decisiones.

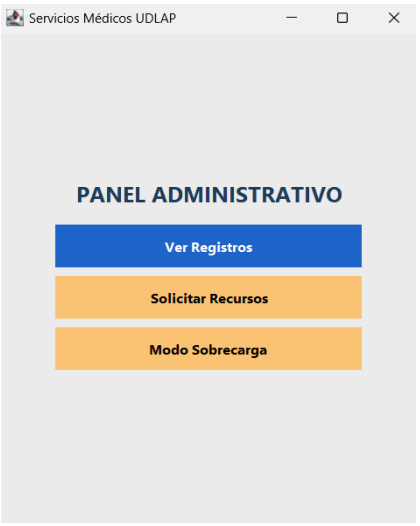


Figura 30. Menú Administrador

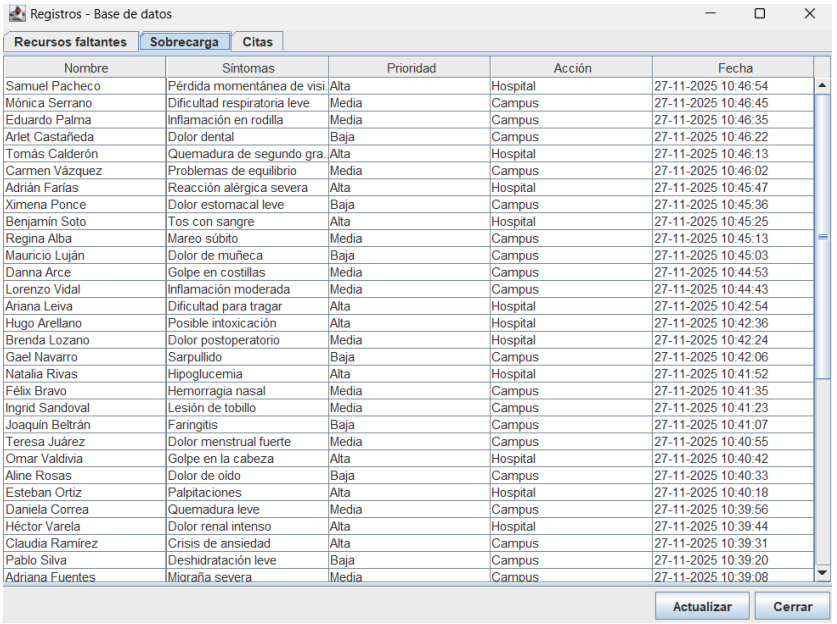
Flujos y Validaciones:

- Navegación: Provee acceso directo a los módulos de "Ver Registros" (historial), "Solicitar Recursos" (inventario) y "Modo Sobrecarga" (emergencia).
- Seguridad: Al ser una ventana independiente, asegura que solo usuarios con credenciales privilegiadas (verificados en el Login) puedan acceder a estas funciones sensibles.

5.2.8 Módulo de Ver Registros

Clase: VerRegistros (Instanciada desde VentanaAdmin)

Descripción: Esta interfaz permite la visualización tabular de la base de datos histórica. Es fundamental para que el administrador pueda consultar citas pasadas, atenciones realizadas o reportes generados sin necesidad de acceder directamente a la consola de SQLite.



Nombre	Síntomas	Prioridad	Acción	Fecha
Samuel Pacheco	Pérdida momentánea de visi	Alta	Hospital	27-11-2025 10:46:54
Mónica Serrano	Dificultad respiratoria leve	Media	Campus	27-11-2025 10:46:45
Eduardo Palma	Inflamación en rodilla	Media	Campus	27-11-2025 10:46:35
Arlet Castañeda	Dolor dental	Baja	Campus	27-11-2025 10:46:22
Tomás Calderón	Quemadura de segundo gra	Alta	Hospital	27-11-2025 10:46:13
Carmen Vázquez	Problemas de equilibrio	Media	Campus	27-11-2025 10:46:02
Adrián Farías	Reacción alérgica severa	Alta	Hospital	27-11-2025 10:45:47
Ximena Ponce	Dolor estomacal leve	Baja	Campus	27-11-2025 10:45:36
Benjamín Soto	Tos con sangre	Alta	Hospital	27-11-2025 10:45:25
Regina Alba	Mareo súbito	Media	Campus	27-11-2025 10:45:13
Mauricio Luján	Dolor de muñeca	Baja	Campus	27-11-2025 10:45:03
Danna Arce	Golpe en costillas	Media	Campus	27-11-2025 10:44:53
Lorenzo Vidal	Inflamación moderada	Media	Campus	27-11-2025 10:44:43
Ariana Leiva	Dificultad para tragar	Alta	Hospital	27-11-2025 10:42:54
Hugo Arellano	Posible intoxicación	Alta	Hospital	27-11-2025 10:42:36
Brenda Lozano	Dolor postoperatorio	Media	Hospital	27-11-2025 10:42:24
Gael Navarro	Sarpullido	Baja	Campus	27-11-2025 10:42:06
Natalia Rivas	Hipoglucemia	Alta	Hospital	27-11-2025 10:41:52
Félix Bravo	Hemorragia nasal	Media	Campus	27-11-2025 10:41:35
Ingrid Sandoval	Lesión de tobillo	Media	Campus	27-11-2025 10:41:23
Joaquín Beltrán	Faringitis	Baja	Campus	27-11-2025 10:41:07
Teresa Juárez	Dolor menstrual fuerte	Media	Campus	27-11-2025 10:40:55
Omar Valdivia	Golpe en la cabeza	Alta	Hospital	27-11-2025 10:40:42
Aline Rosas	Dolor de oído	Baja	Campus	27-11-2025 10:40:33
Esteban Ortiz	Palpitaciones	Alta	Hospital	27-11-2025 10:40:18
Daniela Correa	Quemadura leve	Media	Campus	27-11-2025 10:39:56
Héctor Varela	Dolor renal intenso	Alta	Hospital	27-11-2025 10:39:44
Claudia Ramírez	Crisis de ansiedad	Alta	Campus	27-11-2025 10:39:31
Pablo Silva	Deshidratación leve	Baja	Campus	27-11-2025 10:39:20
Adriana Fuentes	Migraña severa	Media	Campus	27-11-2025 10:39:08

Figura 31. Tabla de Registros

Flujos y Validaciones:

- Consulta: Al abrirse, carga automáticamente los datos almacenados y los dispone en un JTable para su fácil lectura.
- Solo Lectura: Generalmente se implementa en modo de solo lectura para preservar la integridad del historial clínico y administrativo.

5.2.9 Modo Sobrecarga

Clase: Sobrecarga

Descripción: Diseñado específicamente para situaciones de crisis o alta demanda (ej. temporada de influenza o accidentes múltiples). Es un formulario de captura rápida que omite detalles administrativos profundos para priorizar la atención clínica y la clasificación del paciente.

Figura 32. Formulario Sobrecarga

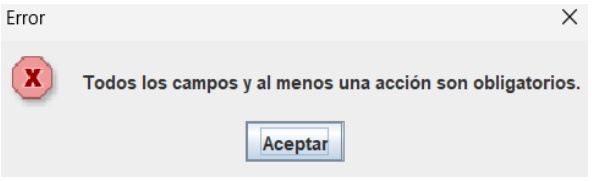


Figura 33. Mensaje de advertencia Sobrecarga

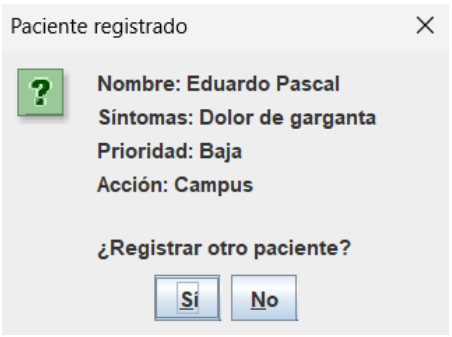


Figura 34. Confirmación del registro

Flujos y Validaciones:

- Campos Obligatorios: El sistema impide guardar el registro si el nombre del paciente o los síntomas están vacíos.
- Lógica de Derivación: Es obligatorio seleccionar una casilla de acción (JCheckBox): "Campus" o "Hospital". Si no se marca ninguna, se lanza una alerta: "Todos los campos y al menos una acción son obligatorios".
- Reinicio Rápido: Tras un registro exitoso, el sistema pregunta "¿Registrar otro paciente?" para limpiar el formulario inmediatamente y continuar con la fila de espera sin cerrar la ventana.

5.2.10 Gestión de Falta de Recursos

Clase: FaltaRecursos

Descripción: Herramienta para el control de inventario crítico. Muestra una tabla con los insumos detectados como agotados o bajo mínimos (RecursoDisponible). Permite al personal

médico validar la falta del insumo y añadir notas para el departamento de compras o farmacia.

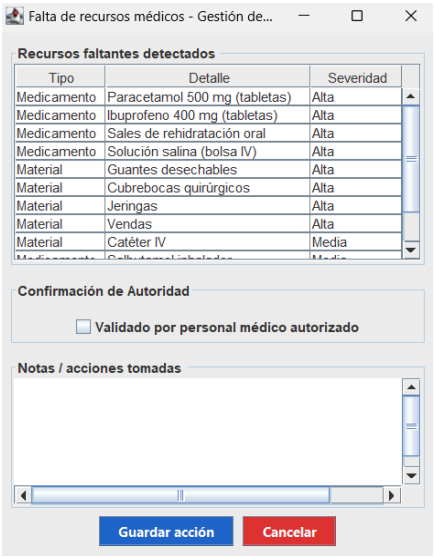


Figura 35. Tabla de recursos

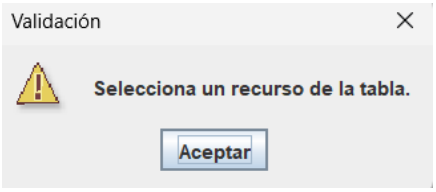


Figura 36. Mensaje de advertencia

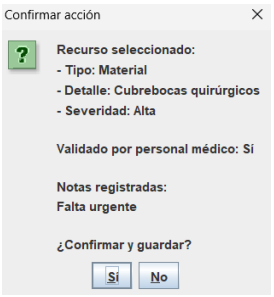


Figura 37. Confirmación del registro

Flujos y Validaciones:

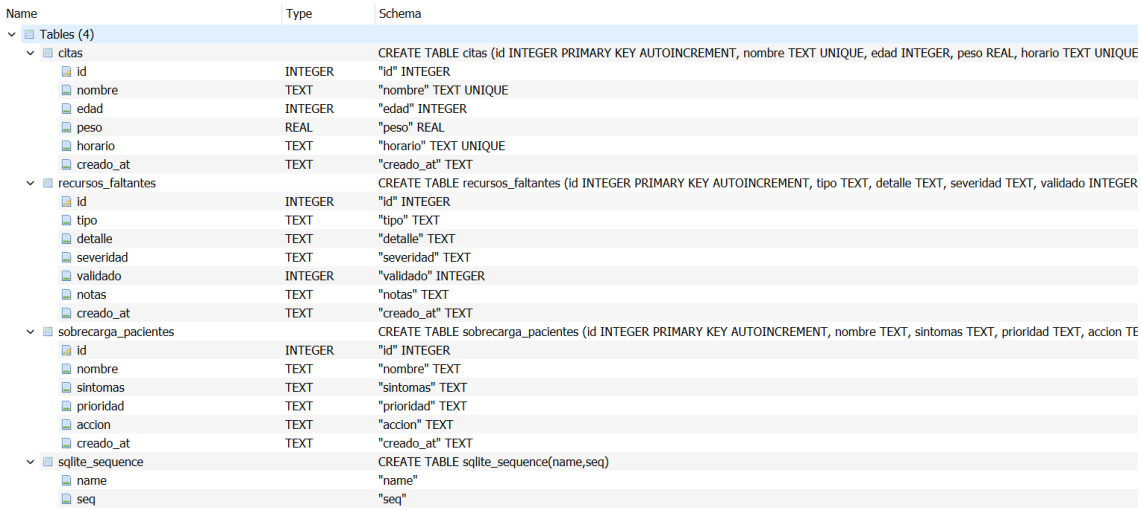
- Selección de Fila: El botón "Guardar acción" verifica primero si el usuario ha seleccionado una fila de la tabla. Si no hay selección, muestra la advertencia: "Selecciona un recurso de la tabla".
- Confirmación de Seguridad: Antes de comprometer los cambios en la base de datos, despliega un cuadro de diálogo con el resumen de la acción (Tipo, Detalle, Severidad y Notas) pidiendo confirmación final, lo que evita solicitudes de recursos erróneas.

6. Integración con Base de Datos SQLite

La persistencia de datos del sistema se gestiona mediante SQLite, un motor de base de datos relacional ligero que no requiere configuración de servidor (serverless), lo cual es ideal para esta aplicación de escritorio. La conexión se realiza a través del controlador JDBC (org.sqlite.JDBC).

6.1 Esquema de la Base de Datos

El diseño de la base de datos formularios.db consta de tres tablas principales que soportan los módulos funcionales, más una tabla interna de secuencia. A continuación se detalla la estructura relacional basada en la implementación final:



Name	Type	Schema
Tables (4)		
citas		CREATE TABLE citas (id INTEGER PRIMARY KEY AUTOINCREMENT, nombre TEXT UNIQUE, edad INTEGER, peso REAL, horario TEXT UNIQUE
id	INTEGER	"id" INTEGER
nombre	TEXT	"nombre" TEXT UNIQUE
edad	INTEGER	"edad" INTEGER
peso	REAL	"peso" REAL
horario	TEXT	"horario" TEXT UNIQUE
creado_at	TEXT	"creado_at" TEXT
recursos_faltantes		CREATE TABLE recursos_faltantes (id INTEGER PRIMARY KEY AUTOINCREMENT, tipo TEXT, detalle TEXT, severidad TEXT, validado INTEGER
id	INTEGER	"id" INTEGER
tipo	TEXT	"tipo" TEXT
detalle	TEXT	"detalle" TEXT
severidad	TEXT	"severidad" TEXT
validado	INTEGER	"validado" INTEGER
notas	TEXT	"notas" TEXT
creado_at	TEXT	"creado_at" TEXT
sobrecarga_pacientes		CREATE TABLE sobrecarga_pacientes (id INTEGER PRIMARY KEY AUTOINCREMENT, nombre TEXT, sintomas TEXT, prioridad TEXT, accion TE
id	INTEGER	"id" INTEGER
nombre	TEXT	"nombre" TEXT
sintomas	TEXT	"sintomas" TEXT
prioridad	TEXT	"prioridad" TEXT
accion	TEXT	"accion" TEXT
creado_at	TEXT	"creado_at" TEXT
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
name		"name"
seq		"seq"

Figura 38. Tablas de la Base de Datos

citas		
Campo	**Tipo**	**Descripción**
id	INTEGER (PK)	Identificador autoincremental.
nombre	TEXT (UNIQUE)	Nombre del paciente.
edad	INTEGER	Edad del paciente.
peso	REAL	Peso en kilogramos.
horario	TEXT (UNIQUE)	Bloque de horario reservado (Ej. "08:00 AM - 08:30 AM").
creado_at	TEXT	Fecha y hora de registro (Timestamp).

Almacena la información de los pacientes y la programación de sus consultas.
Restricciones: Se aplica la restricción UNIQUE en los campos nombre y horario para evitar duplicidad de pacientes activos y traslape de citas (doble reserva).

recursos_faltantes		
Campo	**Tipo**	**Descripción**
id	INTEGER (PK)	Identificador autoincremental.
tipo	TEXT	Categoría del recurso (Ej. Medicamento, Material).
detalle	TEXT	Nombre específico del insumo.
severidad	TEXT	Nivel de urgencia de la reposición.

validado	INTEGER	Booleano (0/1) indicando validación médica.
notas	TEXT	Observaciones adicionales.
creado_at	TEXT	Fecha y hora de registro (Timestamp).

Registra el inventario crítico reportado.

recursos_faltantes		
Campo	**Tipo**	**Descripción**
id	INTEGER (PK)	Identificador autoincremental.
nombre	TEXT	Nombre del paciente en espera.
sintomas	TEXT	Descripción breve del malestar.
prioridad	TEXT	Clasificación de urgencia (Alta, Media, Baja).
accion	TEXT	Destino del paciente (Campus / Hospital).
creado_at	TEXT	Fecha y hora de registro (Timestamp).

Almacena los registros rápidos generados durante el modo de alta demanda (Triage).

6.2 Código de Conexión

La conexión se establece mediante la clase ConexionSQLite. Se utiliza el patrón Singleton implícito en un método estático para obtener la instancia de conexión cuando sea necesario.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConexionSQLite {
    @SuppressWarnings("CallToPrintStackTrace")
    public static Connection conectar() {
        Connection conexion = null;
        try {
            // Cargar el driver JDBC para SQLite
            Class.forName(className: "org.sqlite.JDBC");
            // Ruta a la base de datos SQLite
            String url = "jdbc:sqlite:formularios.db";
            conexion = DriverManager.getConnection(url);
            System.out.println(x: "Conexión exitosa a SQLite");
        } catch (ClassNotFoundException e) {
            System.err.println(x: "Error: No se encontró el driver SQLite");
        } catch (SQLException e) {
            System.err.println(x: "Error de conexión a la base de datos");
            e.printStackTrace();
        }
        return conexion;
    }
```

Figura 39. Código de ConexiónSQLite

6.3 Operaciones CRUD

La clase DatabaseHelper actúa como una capa de acceso a datos, centralizando todas las operaciones CRUD (Create, Read, Update, Delete). Se hace uso estricto de PreparedStatement para prevenir inyección SQL y manejar los tipos de datos correctamente.

6.3.1 Creación (Insert)

Ejemplo de inserción de un reporte de recursos faltantes. Se observa la conversión de booleano a entero para compatibilidad con SQLite.

```java
public static void insertRecurso(String tipo, String detalle, String severidad, boolean validado, String notas) {
    // Validar que los datos no sean nulos ni vacíos
    if (tipo == null || tipo.trim().isEmpty()) tipo = "(Sin tipo)";
    if (detalle == null || detalle.trim().isEmpty()) detalle = "(Sin detalle)";
    if (severidad == null || severidad.trim().isEmpty()) severidad = "(Sin severidad)";
    if (notas == null) notas = "";
    String sql = "INSERT INTO recursos_faltantes(tipo, detalle, severidad, validado, notas, creado_at) VALUES(?,?,?,?,?,?)";
    try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setString(parameterIndex: 1, tipo);
        ps.setString(parameterIndex: 2, detalle);
        ps.setString(parameterIndex: 3, severidad);
        ps.setInt(parameterIndex: 4, validado ? 1 : 0);
        ps.setString(parameterIndex: 5, notas);
        ps.setString(parameterIndex: 6, LocalDateTime.now().format(TF));
        ps.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figura 40. Ejemplo de inserción

6.3.2 Lectura y Validación (Select)

Una función crítica es validar que no existan citas traslapadas. El método isHorarioAvailable consulta si existe algún registro que coincida con el horario solicitado, excluyendo al usuario actual (en caso de modificación).

```java
public static boolean isHorarioAvailable(String horario, String nombreExclusion) {
    if (horario == null || horario.trim().isEmpty()) return true;
    String sql = "SELECT nombre FROM citas WHERE horario = ?" + (nombreExclusion != null ? " AND nombre != ?" : "") + " LIMIT 1";
    try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setString(parameterIndex: 1, horario);
        if (nombreExclusion != null) ps.setString(parameterIndex: 2, nombreExclusion);
        try (ResultSet rs = ps.executeQuery()) {
            return !rs.next();
        }
    } catch (SQLException e) {
        e.printStackTrace();
        // En caso de error de BD, conservador: indicar que no está disponible
        return false;
    }
}
```

Figura 41. Ejemplo de lectura

6.3.3 Actualización (Update)

Utilizado en el módulo ModificarCita, permite cambiar los datos de un paciente existente buscando su nombre (clave única).

```java
public static void updateCitaFieldsByNombre(String nombreClave, int edad, double peso, String horario) {
    String sql = "UPDATE citas SET edad = ?, peso = ?, horario = ?, creado_at = ? WHERE nombre = ?";
    try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setInt(parameterIndex: 1, edad);
        ps.setDouble(parameterIndex: 2, peso);
        ps.setString(parameterIndex: 3, horario);
        ps.setString(parameterIndex: 4, LocalDateTime.now().format(TF));
        ps.setString(parameterIndex: 5, nombreClave);
        ps.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figura 42. Ejemplo de actualización

6.3.3 Eliminación (Delete)

Permite cancelar una cita, liberando el horario para otros estudiantes.

```java
public static void deleteCitaByNombre(String nombre) {
    String sql = "DELETE FROM citas WHERE nombre = ?";
    try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setString(parameterIndex: 1, nombre);
        ps.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figura 43. Ejemplo de eliminación

6.4 Interacción Interfaz - Base de Datos

A continuación se muestran capturas de los formularios enviando y recuperando información de la base de datos.

Figura 44. Registrando unos datos

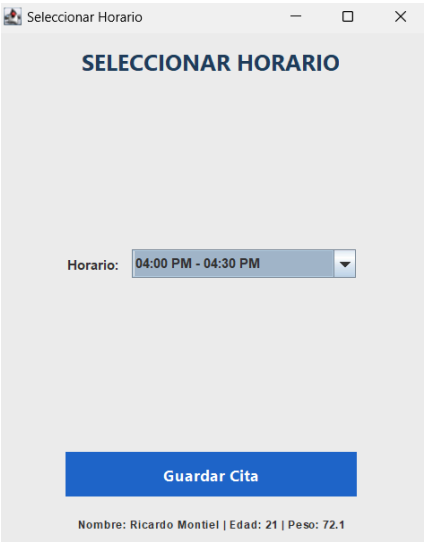


Figura 45. Seleccionando un horario

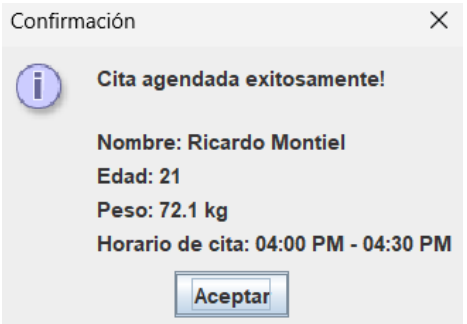


Figura 46. COnfirmación de registro

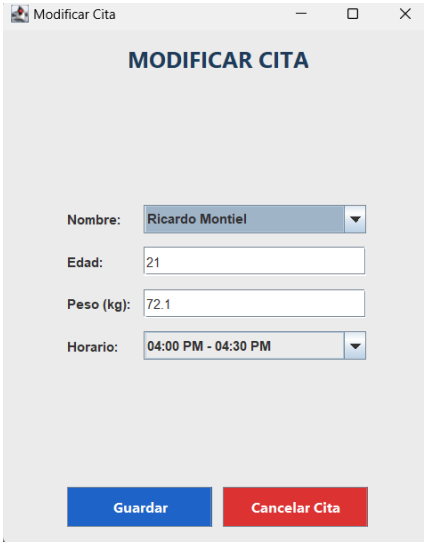


Figura 47. Lectura de datos para modificarlos

7. Pruebas y Validación

Para garantizar la fiabilidad del sistema Servicios Médicos UDLAP, se implementó una estrategia de pruebas automatizadas utilizando el framework JUnit. Las pruebas abarcan tres niveles: pruebas unitarias (lógica de negocio), pruebas de integración (persistencia en base de datos) y pruebas de lógica de interfaz (validación de formularios).

7.1. Descripción de Pruebas Realizadas

A continuación, se detalla el objetivo y alcance de cada clase de prueba desarrollada.

7.1.1. Pruebas de Modelo y Dominio (TestAlumno)

Se validó el encapsulamiento y la integridad de los datos de la clase Alumno.

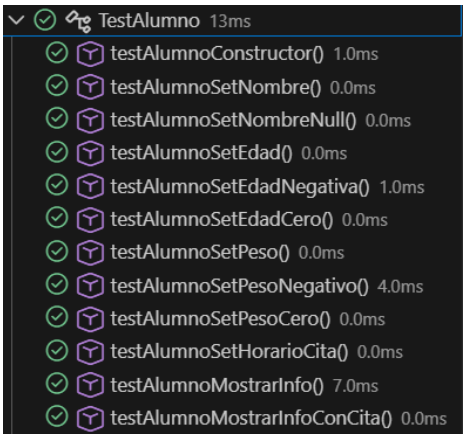


Figura 48. Tests para la clase Alumno

```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;

public class TestAlumno {

    private Alumno alumnoTest;

    @Before
    public void setUp() {
        alumnoTest = new Alumno(nombre: "Juan Pérez", edad: 20, peso: 70.5);
    }

    @Test
    public void testAlumnoConstructor() {
        assertEquals(message: "Nombre debe ser Juan Pérez", expected: "Juan Pérez", alumnoTest.getNombre());
        assertEquals(message: "Edad debe ser 20", expected: 20, alumnoTest.getEdad());
        assertEquals(message: "Peso debe ser 70.5", expected: 70.5, alumnoTest.getPeso(), delta: 0.01);
        assertNull(message: "Horario debe ser null inicialmente", alumnoTest.getHorarioCita());
    }

    @Test
    public void testAlumnoSetNombre() {
        alumnoTest.setNombre(nombre: "María García");
        assertEquals(message: "Nombre debe actualizar a María García", expected: "María García", alumnoTest.getNombre());
    }

    @Test
    public void testAlumnoSetNombreNull() {
        alumnoTest.setNombre(nombre: null);
        assertNull(message: "Nombre puede ser null", alumnoTest.getNombre());
    }

```

Figura 49. TestsAlumno parte 1

```java
    @Test
    public void testAlumnoSetEdad() {
        alumnoTest.setEdad(edad: 25);
        assertEquals(message: "Edad debe actualizar a 25", expected: 25, alumnoTest.getEdad());
    }

    @Test
    public void testAlumnoSetEdadNegativa() {
        try {
            alumnoTest.setEdad(-5);
            fail(message: "Se esperaba IllegalArgumentException al establecer edad negativa");
        } catch (IllegalArgumentException ex) {}
    }

    @Test
    public void testAlumnoSetEdadCero() {
        alumnoTest.setEdad(edad: 0);
        assertEquals(message: "Edad cero se acepta", expected: 0, alumnoTest.getEdad());
    }

    @Test
    public void testAlumnoSetPeso() {
        alumnoTest.setPeso(peso: 75.0);
        assertEquals(message: "Peso debe actualizar a 75.0", expected: 75.0, alumnoTest.getPeso(), delta: 0.01);
    }

    @Test
    public void testAlumnoSetPesoNegativo() {
        try {
            alumnoTest.setPeso(-10.0);
            fail(message: "Se esperaba IllegalArgumentException al establecer peso negativo");
        } catch (IllegalArgumentException ex) {}
    }

```

Figura 50. TestsAlumno parte 2

```java
    @Test
    public void testAlumnoSetPesoCero() {
        alumnoTest.setPeso(peso: 0.0);
        assertEquals(message: "Peso cero se acepta", expected: 0.0, alumnoTest.getPeso(), delta: 0.01);
    }

    @Test
    public void testAlumnoSetHorarioCita() {
        alumnoTest.setHorarioCita(horarioCita: "08:00 AM - 08:30 AM");
        assertEquals(message: "Horario debe ser 08:00 AM - 08:30 AM",
            expected: "08:00 AM - 08:30 AM", alumnoTest.getHorarioCita());
    }

    @Test
    public void testAlumnoMostrarInfo() {
        String info = alumnoTest.mostrarInfo();
        assertTrue(message: "Info debe contener nombre", info.contains(s: "Juan Pérez"));
        assertTrue(message: "Info debe contener edad", info.contains(s: "20"));
        assertTrue(message: "Info debe contener peso", info.contains(s: "70.5"));
        assertTrue(message: "Info debe contener 'Sin cita' si no hay horario", info.contains(s: "Sin cita agendada"));
    }

    @Test
    public void testAlumnoMostrarInfoConCita() {
        alumnoTest.setHorarioCita(horarioCita: "10:00 AM - 10:30 AM");
        String info = alumnoTest.mostrarInfo();
        assertTrue(message: "Info debe contener horario de cita", info.contains(s: "10:00 AM - 10:30 AM"));
    }

}
```

Figura 51. TestsAlumno parte 3

- Validación de Constructores: Verificación de la correcta asignación inicial de nombre, edad y peso.
- Manejo de Excepciones: Se comprueba que el sistema lance IllegalArgumentException si se intentan ingresar edades negativas (ej. -5) o pesos inválidos, protegiendo la lógica del negocio.
- Formato de Salida: Se asegura que el método mostrarInfo() genere cadenas de texto que contengan los datos clave y el estado de la cita (agendada o no).

7.1.2. Pruebas de Lógica de Chatbot (TestChatbot)

Se verificó la estructura del árbol de decisiones del asistente virtual.

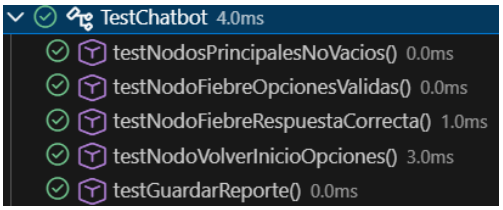


Figura 52. Tests para la clase Chatbot

```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;

public class TestChatbot {

    private String[] opcionesRoot;
    private String[] opcionesFiebre;
    private String respuestaFiebre;
    private String[] opcionesVolver;

    @Before
    public void setUp() {
        // Configuramos la información tal como está en tu Chatbot
        opcionesRoot = new String[]{"Síntomas", "Agendar cita", "Información general"};
        opcionesFiebre = new String[]{"Sí, agendar cita", "Volver al inicio"};
        respuestaFiebre = "La fiebre puede deberse a infección o deshidratación. ¿Deseas agendar cita?";
        opcionesVolver = new String[]{"Síntomas", "Agendar cita", "Información general"};
    }

    // ==================== TESTS PARA NODOS PRINCIPALES ====================

    @Test
    public void testNodosPrincipalesNoVacios() {
        assertNotNull(message: "Las opciones del nodo root no deben ser nulas", opcionesRoot);
        assertEquals(message: "Root debe tener 3 opciones", expected: 3, opcionesRoot.length);

        for (String opcion : opcionesRoot) {
            assertTrue(message: "Cada opción del root debe tener contenido", opcion.length() > 0);
        }
    }

```

Figura 53. TestsChatbot parte 1

```java
    // ==================== TEST DEL NODO FIEBRE ====================

    @Test
    public void testNodoFiebreOpcionesValidas() {
        assertNotNull(message: "Las opciones de fiebre no deben ser nulas", opcionesFiebre);
        assertEquals(message: "El nodo fiebre debe tener 2 opciones", expected: 2, opcionesFiebre.length);

        for (String opcion : opcionesFiebre) {
            assertTrue(message: "Las opciones del nodo fiebre deben tener texto", opcion.length() > 0);
        }
    }

    @Test
    public void testNodoFiebreRespuestaCorrecta() {
        assertNotNull(message: "La respuesta del nodo Fiebre no debe ser nula", respuestaFiebre);
        assertTrue(message: "La respuesta debe mencionar fiebre", respuestaFiebre.toLowerCase().contains(s: "fiebre"));
    }

    // ==================== TEST DEL NODO VOLVER AL INICIO ====================

    @Test
    public void testNodoVolverInicioOpciones() {
        assertNotNull(message: "El nodo Volver al inicio no debe ser nulo", opcionesVolver);
        assertEquals(message: "Volver al inicio debe tener 3 opciones", expected: 3, opcionesVolver.length);
    }

```

Figura 54. TestsChatbot parte 2

```java
    // ==================== TEST DEL REPORTE ====================

    @Test
    public void testGuardarReporte() {
        String usuario = "Usuario prueba";
        String bot = "Mensaje prueba";

        assertNotNull(message: "El usuario no debe ser nulo", usuario);
        assertNotNull(message: "El mensaje del bot no debe ser nulo", bot);

        assertTrue(message: "El usuario debe tener contenido", usuario.length() > 0);
        assertTrue(message: "El mensaje del bot debe tener contenido", bot.length() > 0);
    }
}
```

Figura 55. TestsChatbot parte 3

- Integridad de Nodos: Se valida que los nodos principales ("Root", "Fiebre") no sean nulos y contengan las opciones de respuesta esperadas.
- Flujo de Conversación: Se asegura que las respuestas del bot contengan las palabras clave correctas (ej. detectar "fiebre" en la respuesta del nodo correspondiente) y que el mecanismo de reporte no genere archivos vacíos.

7.1.3. Pruebas de Integración de Base de Datos (TestCitas y TestFaltaRecursos)

Estas pruebas son críticas, ya que interactúan directamente con DatabaseHelper y SQLite para verificar las operaciones CRUD.

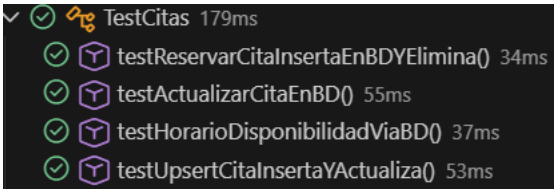


Figura 56. Tests para las Citas

```java
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.*;
import java.util.List;

public class TestCitas {

    @BeforeClass
    public static void setup() {
        DatabaseHelper.createTables();
    }

    @Test
    public void testReservarCitaInsertaEnBDYElimina() {
        String nombre = "Test Citas A";
        String horario = "08:00 AM - 08:30 AM";
        // Asegurar que el horario esté libre para la prueba
        DatabaseHelper.deleteCitaByHorario(horario);
        // Intentar reservar
        boolean ok = DatabaseHelper.reservarOCita(nombre, edad: 30, peso: 70.0, horario);
        assertTrue(message: "La reserva debe retornar true si el horario está disponible", ok);

        // Verificar que exista en la BD
        List<String[]> citas = DatabaseHelper.getCitas();
        boolean encontrado = false;
        for (String[] c : citas) {
            if (c[0].equals(nombre) && horario.equals(c[3])) {
                encontrado = true;
                break;
            }
        }
        assertTrue(message: "La cita reservada debe encontrarse en la BD", encontrado);

```

Figura 57. TestCitas parte 1

```
38
39          // Limpieza
40          DatabaseHelper.deleteCitaByNombre(nombre);
41          List<String[]> despues = DatabaseHelper.getCitas();
42          boolean sigue = false;
43          for (String[] c : despues) if (c[0].equals(nombre)) { sigue = true; break; }
44          assertFalse(message: "La cita debe eliminarse en la limpieza", sigue);
45      }
46
47      @Test
48      public void testActualizarCitaEnBD() {
49          String nombre = "UpdateTest";
50          // Asegurar limpieza previa
51          DatabaseHelper.deleteCitaByNombre(nombre);
52          DatabaseHelper.deleteCitaByHorario(horario: "08:00 AM - 08:30 AM");
53          DatabaseHelper.deleteCitaByHorario(horario: "10:00 AM - 10:30 AM");
54          DatabaseHelper.reservarOCita(nombre, edad: 25, peso: 65.0, horario: "08:00 AM - 08:30 AM");
55
56          // Actualizar por nombre
57          DatabaseHelper.updateCitaByNombre(nombre, nombre, edad: 26, peso: 66.0, horario: "10:00 AM - 10:30 AM");
58
59          List<String[]> citas = DatabaseHelper.getCitas();
60          boolean actualizado = false;
61          for (String[] c : citas) {
62              if (c[0].equals(nombre) && "10:00 AM - 10:30 AM".equals(c[3]) && Integer.parseInt(c[1]) == 26) {
63                  actualizado = true;
64                  break;
65              }
66          }
67          assertTrue(message: "La cita debe actualizarse en la BD", actualizado);
```

Figura 58. TestCitas parte 2

```
67          assertTrue(message: "La cita debe actualizarse en la BD", actualizado);
68
69          // Limpieza
70          DatabaseHelper.deleteCitaByNombre(nombre);
71          List<String[]> despues = DatabaseHelper.getCitas();
72          boolean sigue = false;
73          for (String[] c : despues) if (c[0].equals(nombre)) { sigue = true; break; }
74          assertFalse(message: "La cita debe eliminarse en la limpieza", sigue);
75      }
76
77      @Test
78      public void testHorarioDisponibilidadViaBD() {
79          String nombre = "BusyUser";
80          String horario = "08:00 AM - 08:30 AM";
81          // Asegurar que el horario esté libre y luego reservarlo
82          DatabaseHelper.deleteCitaByHorario(horario);
83          DatabaseHelper.reservarOCita(nombre, edad: 22, peso: 60.0, horario);
84
85          // Para otro nombre, el horario debe estar ocupado
86          boolean disponible = DatabaseHelper.isHorarioAvailable(horario, nombreExclusion: "OtherUser");
87          assertFalse(message: "Horario ocupado debería reportarse como no disponible", disponible);
88
89          // Limpieza
90          DatabaseHelper.deleteCitaByNombre(nombre);
91          List<String[]> despues = DatabaseHelper.getCitas();
92          boolean sigue = false;
93          for (String[] c : despues) if (c[0].equals(nombre)) { sigue = true; break; }
94          assertFalse(message: "La cita debe eliminarse en la limpieza", sigue);
95      }
96
```

Figura 59. TestCitas parte 3

```
97      @Test
98      public void testUpsertCitaInsertaYActualiza() {
99          String nombre = "UpsertUser";
100         // Asegurar limpieza previa en horarios objetivo
101         DatabaseHelper.deleteCitaByNombre(nombre);
102         DatabaseHelper.deleteCitaByHorario(horario: "08:00 AM - 08:30 AM");
103         DatabaseHelper.deleteCitaByHorario(horario: "10:00 AM - 10:30 AM");
104
105         // Usar horarios libres (08:00 y 10:00) para evitar colisiones con datos existentes
106         DatabaseHelper.upsertCita(nombre, edad: 20, peso: 60.0, horario: "08:00 AM - 08:30 AM");
107
108         // Volver a upsert con mismo nombre y horario distinto (libre)
109         DatabaseHelper.upsertCita(nombre, edad: 21, peso: 61.5, horario: "10:00 AM - 10:30 AM");
110
111         List<String[]> citas = DatabaseHelper.getCitas();
112         boolean found = false;
113         for (String[] c : citas) {
114             if (c[0].equals(nombre) && "10:00 AM - 10:30 AM".equals(c[3]) && Integer.parseInt(c[1]) == 21) {
115                 found = true;
116                 break;
117             }
118         }
119         assertTrue(message: "Upsert debe actualizar la cita existente para el mismo nombre", found);
120
121         // Limpieza
122         DatabaseHelper.deleteCitaByNombre(nombre);
123         List<String[]> despues = DatabaseHelper.getCitas();
124         boolean sigue = false;
125         for (String[] c : despues) if (c[0].equals(nombre)) { sigue = true; break; }
126         assertFalse(message: "La cita debe eliminarse en la limpieza", sigue);
127     }
128 }
129
```

Figura 60. TestCitas parte 4

Prevención de Conflictos (Citas):

- Se validó la función isHorarioAvailable para asegurar que el sistema impida agendar una cita en un horario ya ocupado por otro usuario.
- Se probó la lógica de Upsert (actualizar si existe, insertar si no) para modificaciones de citas.

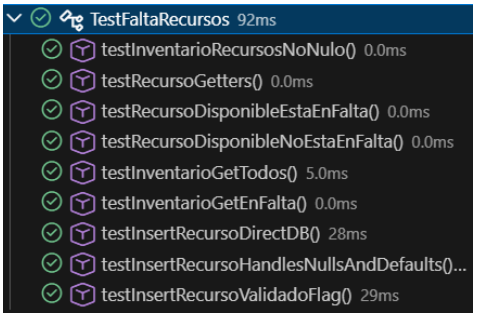


Figura 61. Tests para la clase FaltaRecursos

```java
import org.junit.Test;
import org.junit.BeforeClass;
import org.junit.AfterClass;
import static org.junit.Assert.*;
import java.util.List;
import java.sql.Connection;
import java.sql.PreparedStatement;


public class TestFaltaRecursos {

    @BeforeClass
    public static void setup() {
        DatabaseHelper.createTables();
    }

    @AfterClass
    public static void cleanupAllTestRows() throws Exception {
        // Eliminar cualquier fila generada por tests cuyo tipo empiece con TEST_
        try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(
                sql: "DELETE FROM recursos_faltantes WHERE tipo LIKE 'TEST_%' OR tipo = 'TEST_ORDER'")) {
            ps.executeUpdate();
        }
    }

    @Test
    public void testInventarioRecursosNoNulo() {
        InventarioRecursos inventario = new InventarioRecursos();
        assertNotNull(message: "Inventario no debe ser null", inventario);
    }

```

Figura 62. TestFaltaRecursos parte 1

```java
    @Test
    public void testRecursoGetters() {
        RecursoDisponible recurso = new RecursoDisponible(id: "R001", tipo: "Medicamento", nombre: "Paracetamol",
            cantidadActual: 5, cantidadMinima: 30, demanda: "Alta", sinAl… true);
        assertEquals(message: "ID debe ser R001", expected: "R001", recurso.getId());
        assertEquals(message: "Tipo debe ser Medicamento", expected: "Medicamento", recurso.getTipo());
        assertEquals(message: "Nombre debe ser Paracetamol", expected: "Paracetamol", recurso.getNombre());
        assertEquals(message: "Cantidad actual debe ser 5", expected: 5, recurso.getCantidadActual());
        assertEquals(message: "Cantidad mínima debe ser 30", expected: 30, recurso.getCantidadMinima());
        assertEquals(message: "Demanda debe ser Alta", expected: "Alta", recurso.getDemanda());
        assertTrue(message: "Sin alternativa debe ser true", recurso.isSinAlternativa());
    }

    @Test
    public void testRecursoDisponibleEstaEnFalta() {
        RecursoDisponible recurso = new RecursoDisponible(id: "R001", tipo: "Medicamento", nombre: "Paracetamol",
            cantidadActual: 2, cantidadMinima: 10, demanda: "Alta", sinAl… false);
        assertTrue(message: "Recurso debe estar en falta (2 < 10)", recurso.estaEnFalta());
    }

    @Test
    public void testRecursoDisponibleNoEstaEnFalta() {
        RecursoDisponible recurso = new RecursoDisponible(id: "R002", tipo: "Medicamento", nombre: "Ibuprofeno",
            cantidadActual: 15, cantidadMinima: 10, demanda: "Alta", sinAl… false);
        assertFalse(message: "Recurso no debe estar en falta (15 >= 10)", recurso.estaEnFalta());
    }

    @Test
    public void testInventarioGetTodos() {
        InventarioRecursos inventario = new InventarioRecursos();
        assertNotNull(message: "Lista de recursos no debe ser null", inventario.getTodos());
        assertTrue(message: "Debe haber al menos un recurso", inventario.getTodos().size() > 0);
    }
```

Figura 63. TestFaltaRecursos parte 2

```
71      @Test
72      public void testInventarioGetEnFalta() {
73          InventarioRecursos inventario = new InventarioRecursos();
74          assertNotNull(message: "Lista de faltantes no debe ser null", inventario.getEnFalta());
75          assertTrue(message: "Lista de faltantes debe tener size >= 0", inventario.getEnFalta().size() >= 0);
76      }
77
78      @Test
79      public void testInsertRecursoDirectDB() throws Exception {
80          String tipo = "TEST_RECURSO_" + System.currentTimeMillis();
81          String detalle = "Detalle prueba";
82          String severidad = "Alta";
83          boolean validado = true;
84          String notas = "NOTAS_TEST_" + System.currentTimeMillis();
85
86          // Insertar
87          DatabaseHelper.insertRecurso(tipo, detalle, severidad, validado, notas);
88
89          // Verificar que esté en la BD
90          boolean found = false;
91          List<String[]> recursos = DatabaseHelper.getRecursos();
92          for (String[] r : recursos) {
93              if (r.length >= 6 && tipo.equals(r[0]) && detalle.equals(r[1])) {
94                  found = true;
95                  assertEquals(message: "Severidad debe coincidir", severidad, r[2]);
96                  assertTrue(message: "Campo validado debe contener Sí o No", r[3].equals(anObject: "Sí") || r[3].equals(anObject: "No"));
97                  assertEquals(message: "Notas debe coincidir", notas, r[4]);
98                  break;
99              }
100         }
```

Figura 64. TestFaltaRecursos parte 3

```
101
102         // Limpieza: eliminar por tipo
103         try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql: "DELETE FROM recursos_faltant
104             ps.setString(parameterIndex: 1, tipo);
105             ps.executeUpdate();
106         }
107
108         assertTrue(message: "Registro insertado debe encontrarse en la BD", found);
109     }
110
111     @Test
112     public void testInsertRecursoHandlesNullsAndDefaults() throws Exception {
113         String detalle = "DL_NULL_TEST_" + System.currentTimeMillis();
114
115         // tipo=null, severidad=null, notas=null -> debe usar valores por defecto
116         DatabaseHelper.insertRecurso(tipo: null, detalle, severidad: null, validado: false, notas: null);
117
118         boolean found = false;
119         for (String[] r : DatabaseHelper.getRecursos()) {
120             if (r.length >= 6 && detalle.equals(r[1])) {
121                 found = true;
122                 assertEquals(message: "Tipo por defecto", expected: "(Sin tipo)", r[0]);
123                 assertEquals(message: "Detalle debe coincidir", detalle, r[1]);
124                 assertEquals(message: "Severidad por defecto", expected: "(Sin severidad)", r[2]);
125                 assertEquals(message: "Notas por defecto vacías", expected: "", r[4]);
126                 assertTrue(message: "Validado debe ser 'No'", r[3].equals(anObject: "No") || r[3].equals(anObject: "Sí"));
127                 break;
128             }
129         }
130
```

Figura 65. TestFaltaRecursos parte 4

```
131         // Limpieza
132         try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql: "DELETE FROM recursos_faltant
133             ps.setString(parameterIndex: 1, detalle);
134             ps.executeUpdate();
135         }
136
137         assertTrue(message: "Registro con valores nulos debe insertarse y encontrarse", found);
138     }
139
140     @Test
141     public void testInsertRecursoValidadoFlag() throws Exception {
142         String tipo = "TEST_VALIDADO_" + System.currentTimeMillis();
143         String detalle = "DetalleValFlag" + System.currentTimeMillis();
144
145         DatabaseHelper.insertRecurso(tipo, detalle, severidad: "Baja", validado: false, notas: "");
146
147         boolean found = false;
148         for (String[] r : DatabaseHelper.getRecursos()) {
149             if (r.length >= 6 && detalle.equals(r[1])) {
150                 found = true;
151                 assertEquals(message: "Validado debe ser 'No' para false", expected: "No", r[3]);
152                 break;
153             }
154         }
155
156         // Limpieza
157         try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql: "DELETE FROM recursos_faltant
158             ps.setString(parameterIndex: 1, tipo);
159             ps.executeUpdate();
160         }
161
162         assertTrue(message: "Registro insertado debe encontrarse en la BD", found);
163     }
164 }
```

Figura 66. TestFaltaRecursos parte 5

Gestión de Inventario (Recursos):

- Se verificó la inserción correcta de recursos faltantes, incluyendo el manejo de valores nulos (defaults) y la persistencia de banderas booleanas (validado).
- Se implementaron rutinas de limpieza (@AfterClass) para borrar los datos de prueba (TEST_%) y no ensuciar la base de datos real.

7.1.4. Pruebas de Sistema y Seguridad (TestSM)

Se validó el mecanismo de autenticación simulado en DatosInicio.

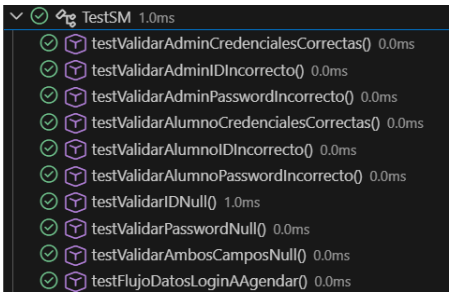


Figura 67. Tests para validar credenciales y flujo

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class TestSM {

    // ================== TESTS PARA DatosInicio ==================
    @Test
    public void testValidarAdminCredencialesCorrectas() {
        assertTrue(message: "Admin debe validarse con ID 12345 y PW Admin.1",
            DatosInicio.validar(id: "12345", contra: "Admin.1"));
    }

    @Test
    public void testValidarAdminIDIncorrecto() {
        assertFalse(message: "Admin con ID incorrecto no debe validarse",
            DatosInicio.validar(id: "99999", contra: "Admin.1"));
    }

    @Test
    public void testValidarAdminPasswordIncorrecto() {
        assertFalse(message: "Admin con PW incorrecta no debe validarse",
            DatosInicio.validar(id: "12345", contra: "WrongPassword"));
    }

    @Test
    public void testValidarAlumnoCredencialesCorrectas() {
        assertTrue(message: "Alumno debe validarse con ID 123456 y PW Alumno.1",
            DatosInicio.validar(id: "123456", contra: "Alumno.1"));
    }

```

Figura 68. TestSM parte 1

```java
    @Test
    public void testValidarAlumnoIDIncorrecto() {
        assertFalse(message: "Alumno con ID incorrecto no debe validarse",
            DatosInicio.validar(id: "654321", contra: "Alumno.1"));
    }

    @Test
    public void testValidarAlumnoPasswordIncorrecto() {
        assertFalse(message: "Alumno con PW incorrecta no debe validarse",
            DatosInicio.validar(id: "123456", contra: "WrongPassword"));
    }

    @Test
    public void testValidarIDNull() {
        assertFalse(message: "ID null no debe validarse",
            DatosInicio.validar(id: null, contra: "Admin.1"));
    }

    @Test
    public void testValidarPasswordNull() {
        assertFalse(message: "PW null no debe validarse",
            DatosInicio.validar(id: "12345", contra: null));
    }

    @Test
    public void testValidarAmbosCamposNull() {
        assertFalse(message: "Ambos null no deben validarse",
            DatosInicio.validar(id: null, contra: null));
    }
```

Figura 69. TestSM parte 2

```java
    // Tests ligero de flujo
    @Test
    public void testFlujoDatosLoginAAgendar() {
        assertTrue(message: "Alumno debe validarse", DatosInicio.validar(id: "123456", contra: "Alumno.1"));
        Alumno alumno = new Alumno(nombre: "Carlos López", edad: 21, peso: 72.0);
        assertNotNull(message: "Alumno debe crearse", alumno);
        alumno.setHorarioCita(horarioCita: "09:00 AM - 09:30 AM");
        assertEquals(message: "Horario debe asignarse", expected: "09:00 AM - 09:30 AM", alumno.getHorarioCita());
    }
}
```

Figura 70. TestSM parte 3

- Credenciales: Se probaron combinaciones de éxito (Admin/Admin.1) y fracaso (ID incorrecto, Contraseña errónea, valores null), asegurando que solo usuarios autorizados accedan a los paneles correspondientes.

7.1.5. Pruebas de Componentes GUI (TestSobrecarga)

Se utilizó introspección (Java Reflection) para validar la existencia y estado de los componentes gráficos sin necesidad de interacción manual.

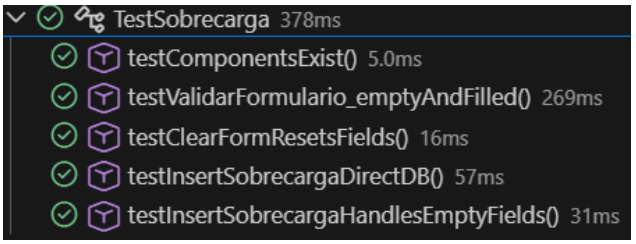


Figura 71. Tests para la clase Sobrecarga

```java
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.*;

import java.lang.reflect.Field;
import java.sql.Connection;
import java.sql.PreparedStatement;
import javax.swing.*;

public class TestSobrecarga {

    @BeforeClass
    public static void setup() {
        // Asegurarse que la tabla exista antes
        DatabaseHelper.createTables();
    }

    @Test
    public void testComponentsExist() throws Exception {
        Sobrecarga window = new Sobrecarga();

        // Acceder a campos privados
        Field fNombre = Sobrecarga.class.getDeclaredField(name: "txtNombre");
        Field fSintomas = Sobrecarga.class.getDeclaredField(name: "txtSintomas");
        Field fPrioridad = Sobrecarga.class.getDeclaredField(name: "comboPrioridad");
        Field fCampus = Sobrecarga.class.getDeclaredField(name: "campusCheck");
        Field fHospital = Sobrecarga.class.getDeclaredField(name: "hospitalCheck");

        fNombre.setAccessible(flag: true);
        fSintomas.setAccessible(flag: true);
        fPrioridad.setAccessible(flag: true);
        fCampus.setAccessible(flag: true);
        fHospital.setAccessible(flag: true);
```

Figura 72. TestSobrecarga parte 1

```
40          assertNotNull(message: "Campo txtNombre debe existir", fNombre.get(window));
41          assertNotNull(message: "Campo txtSintomas debe existir", fSintomas.get(window));
42          assertNotNull(message: "Combo prioridad debe existir", fPrioridad.get(window));
43          assertNotNull(message: "Checkbox campus debe existir", fCampus.get(window));
44          assertNotNull(message: "Checkbox hospital debe existir", fHospital.get(window));
45      }
46
47      @Test
48      public void testValidarFormulario_emptyAndFilled() throws Exception {
49          Sobrecarga s = new Sobrecarga();
50
51          // Empeiza vacío, por lo que debería de ser invalido
52          assertFalse(message: "Formulario vacío debe ser inválido", s.validarFormulario());
53
54          // Llenar los campos
55          Field fNombre = Sobrecarga.class.getDeclaredField(name: "txtNombre");
56          Field fSintomas = Sobrecarga.class.getDeclaredField(name: "txtSintomas");
57          Field fCampus = Sobrecarga.class.getDeclaredField(name: "campusCheck");
58          fNombre.setAccessible(flag: true);
59          fSintomas.setAccessible(flag: true);
60          fCampus.setAccessible(flag: true);
61
62          JTextField nombreField = (JTextField) fNombre.get(s);
63          JTextArea sintomasArea = (JTextArea) fSintomas.get(s);
64          JCheckBox campus = (JCheckBox) fCampus.get(s);
65
66          nombreField.setText(t: "Paciente Test");
67          sintomasArea.setText(t: "Dolor de cabeza");
68          campus.setSelected(b: true);
69
70          assertTrue(message: "Formulario lleno debe ser válido", s.validarFormulario());
71      }
72
```

Figura 73. TestSobrecarga parte 2

```
73      @Test
74      public void testClearFormResetsFields() throws Exception {
75          Sobrecarga s = new Sobrecarga();
76          Field fNombre = Sobrecarga.class.getDeclaredField(name: "txtNombre");
77          Field fSintomas = Sobrecarga.class.getDeclaredField(name: "txtSintomas");
78          Field fPrioridad = Sobrecarga.class.getDeclaredField(name: "comboPrioridad");
79          Field fCampus = Sobrecarga.class.getDeclaredField(name: "campusCheck");
80          Field fHospital = Sobrecarga.class.getDeclaredField(name: "hospitalCheck");
81
82          fNombre.setAccessible(flag: true);
83          fSintomas.setAccessible(flag: true);
84          fPrioridad.setAccessible(flag: true);
85          fCampus.setAccessible(flag: true);
86          fHospital.setAccessible(flag: true);
87
88          JTextField nombreField = (JTextField) fNombre.get(s);
89          JTextArea sintomasArea = (JTextArea) fSintomas.get(s);
90          JComboBox<?> prioridad = (JComboBox<?>) fPrioridad.get(s);
91          JCheckBox campus = (JCheckBox) fCampus.get(s);
92          JCheckBox hospital = (JCheckBox) fHospital.get(s);
93
94          nombreField.setText(t: "X");
95          sintomasArea.setText(t: "Y");
96          prioridad.setSelectedIndex(anIndex: 2);
97          campus.setSelected(b: true);
98          hospital.setSelected(b: true);
99
100         s.clearForm();
101
102         assertEquals(expected: "", nombreField.getText());
103         assertEquals(expected: "", sintomasArea.getText());
104         assertEquals(expected: 0, prioridad.getSelectedIndex());
105         assertFalse(campus.isSelected());
106         assertFalse(hospital.isSelected());
107     }
```

Figura 74. TestSobrecarga parte 3

```java
@Test
public void testInsertSobrecargaDirectDB() throws Exception {
    String nombre = "TEST_SOBRECARGA_" + System.currentTimeMillis();
    String sintomas = "Síntomas de prueba";
    String prioridad = "Media";
    String accion = "Campus";

    // Insertar directamente
    DatabaseHelper.insertSobrecarga(nombre, sintomas, prioridad, accion);

    boolean found = false;
    for (String[] row : DatabaseHelper.getSobrecarga()) {
        if (row.length > 0 && nombre.equals(row[0])) {
            found = true;
            assertEquals(message: "Prioridad debe coincidir", prioridad, row[2]);
            assertEquals(message: "Acción debe coincidir", accion, row[3]);
            break;
        }
    }

    // Limpieza de datos de prueba
    try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql: "DELETE FROM sobrecarga_pacie
        ps.setString(parameterIndex: 1, nombre);
        ps.executeUpdate();
    }
    assertTrue(message: "Registro insertado debe encontrarse en la BD", found);
}
```

Figura 75. TestSobrecarga parte 4

```java

    @Test
    public void testInsertSobrecargaHandlesEmptyFields() throws Exception {
        String nombre = "TEST_EMPTY_FIELDS_" + System.currentTimeMillis();
        // Insretar sin datos
        DatabaseHelper.insertSobrecarga(nombre, sintomas: "", prioridad: "Baja", accion: "");

        boolean found = false;
        for (String[] row : DatabaseHelper.getSobrecarga()) {
            if (row.length > 0 && nombre.equals(row[0])) {
                found = true;
                break;
            }
        }

        // Limpieza de datos
        try (Connection conn = ConexionSQLite.conectar(); PreparedStatement ps = conn.prepareStatement(sql: "DELETE FROM sobrecarga_pacie
            ps.setString(parameterIndex: 1, nombre);
            ps.executeUpdate();
        }

        assertTrue(message: "Registro con campos vacíos debe insertarse", found);
    }
}

```

Figura 76. TestSobrecarga parte 5

- Validación de Formularios: Se probó el método validarFormulario() para asegurar que retorna false si los campos están vacíos y true solo cuando el nombre, síntomas y la acción (Campus/Hospital) han sido completados.

7.2 Resultados de Pruebas

Al ejecutar la suite de pruebas en el entorno de desarrollo, se obtuvo el siguiente reporte de salida, confirmando que el 100% de los 56 casos de prueba pasaron exitosamente.

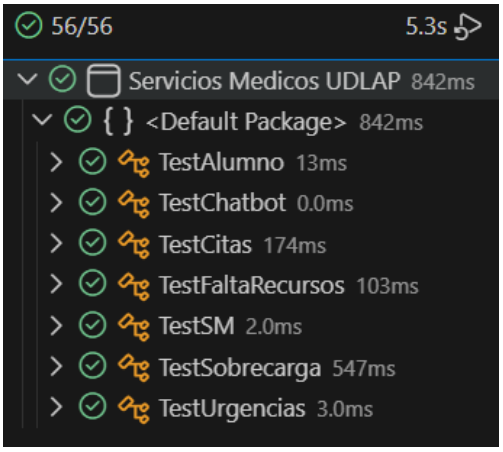


Figura 77. Validación de todos los Tests

8. Conclusiones del Equipo

Durante el ciclo de desarrollo, el equipo enfrentó desafíos técnicos significativos, principalmente en la integración de la interfaz gráfica (Swing) con la lógica de negocio. Asegurar una navegación fluida entre las distintas ventanas y mantener la persistencia de datos estable con SQLite requirió múltiples iteraciones de depuración. Asimismo, la validación robusta de datos (como evitar el traslape de horarios o la entrada de valores nulos) fue compleja de implementar correctamente en todas las pantallas. Otro reto importante fue la alineación de los modelos UML con el código final; los diagramas de clases y actividades tuvieron que ser refactorizados varias veces para reflejar fielmente los cambios que surgieron durante la programación de los módulos de Sobrecarga y Falta de Recursos.

El proyecto reforzó la importancia del diseño modular. Comprendimos que separar la lógica de conexión (DatabaseHelper) de la interfaz visual (JFrames) no solo ordena el código, sino que facilita enormemente las pruebas y el mantenimiento. También valoramos el rol de las herramientas de Inteligencia Artificial como apoyo en la fase de diseño (generación de escenarios y flujos), aprendiendo que, aunque aceleran el proceso creativo, el criterio humano es indispensable para filtrar y adaptar esas propuestas a las limitaciones reales del sistema. Además, consolidamos conocimientos sobre el manejo de excepciones en Java y la importancia de la experiencia de usuario al diseñar formularios claros y validaciones inmediatas.

Finalmente, el sistema queda con posibilidades de mejora, como incorporar notificaciones automáticas, ampliar funcionalidad del chatbot, optimizar el manejo del inventario o integrar el sistema con plataformas institucionales de las cuales requieren de un permiso proporcionado por la universidad, estas extensiones permiten aumentar la utilidad y acercarlo más a un entorno clínico completo.