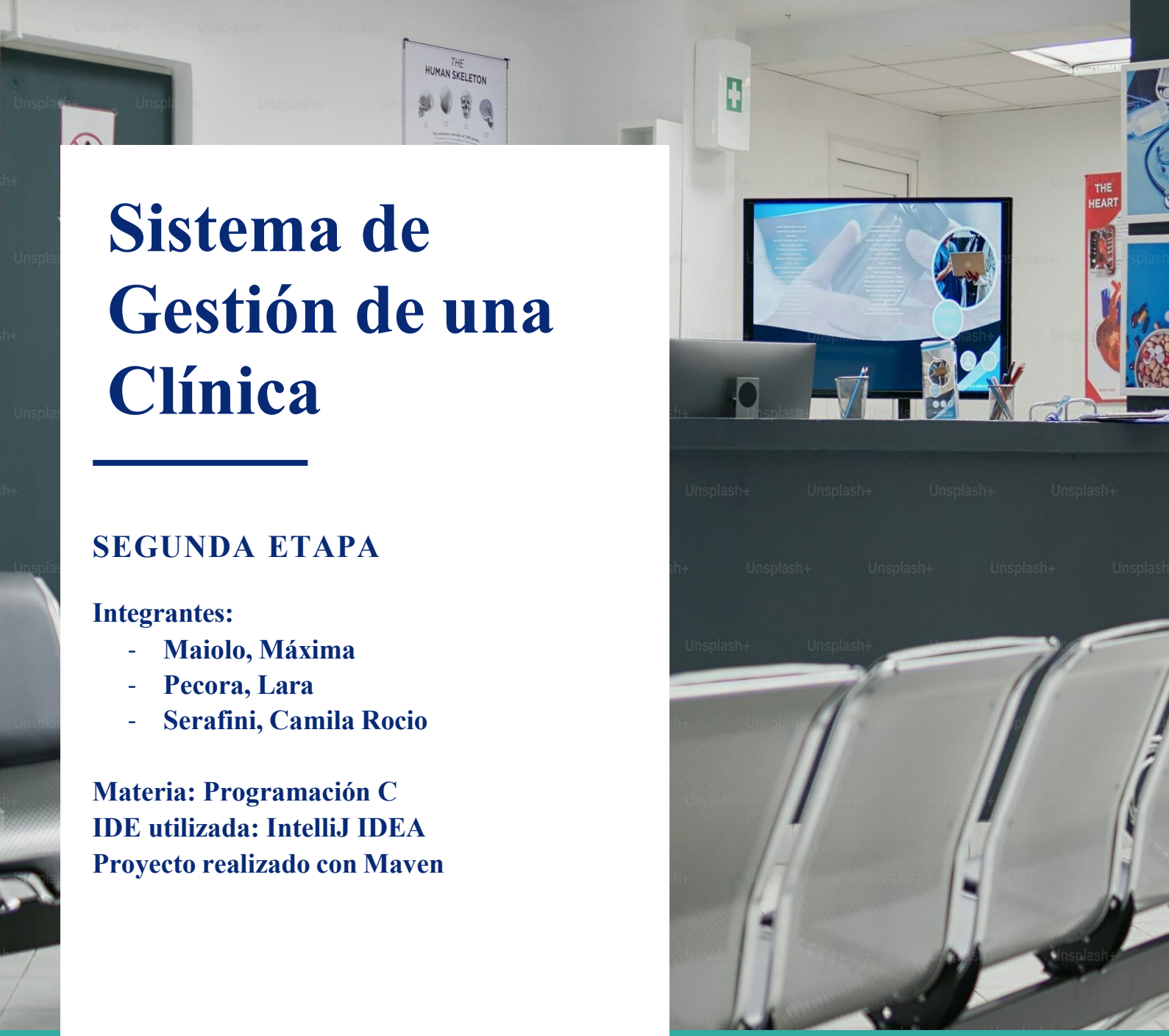


Sistema de Gestión de una Clínica

SEGUNDA ETAPA

Integrantes:

- **Maiolo, Máxima**
- **Pecora, Lara**
- **Serafini, Camila Rocio**

Materia: Programación C
IDE utilizada: IntelliJ IDEA
Proyecto realizado con Maven

Descripción general del sistema

El presente informe documenta la implementación completa del Sistema de Gestión de una Clínica Privada, un proyecto desarrollado en dos etapas. El objetivo general del mismo ha sido modelar un sistema robusto que maneje la gestión de pacientes y médicos (Etapa I) y se amplíe con un módulo de simulación de ambulancia con concurrencia y una interfaz gráfica de usuario (Etapa II).

Este informe detalla la estructura final del proyecto, la justificación de las decisiones de diseño, y la aplicación práctica de los patrones de diseño requeridos, presentando un Diagrama de Clases UML y e código representativo de las implementaciones clave.

Estructura del proyecto

El proyecto se encuentra organizado en distintos paquetes, diseñados para reflejar la separación de responsabilidades y modularizar el funcionamiento de la clínica.

- **Paquete: controlador**

  Contiene la clase Controlador: la cual actúa como intermediaria entre la vista y el modelo que serán explicados luego. Interpreta eventos de la vista y ejecuta operaciones en el modelo.

- **Paquete: modelo**

  Contiene toda la lógica del dominio. Dentro tiene los siguientes paquetes:

  - Paquete sistema: este paquete contiene la base del sistema y las clases principales que gestionan la clínica:
    - ✓ Clinica: posee los atributos nombre, dirección, teléfono e instancia (estático). Implementa el patrón Singleton mediante un constructor privado y un método estático público getInstancia.
    - ✓ SistemaFacade: cumple el rol de fachada, centralizando la comunicación entre los módulos ModuloIngreso, ModuloAtencion y ModuloEgreso. También implementa el patrón Singleton.

  Centralizando la comunicación entre los módulos, estos se encuentran en tres subpaquetes que corresponden a cada uno:

  - ModuloAtencion: El paquete ModuloAtencion concentra toda la lógica interna relacionada con el proceso de atención médica dentro de la clínica. Su función principal es gestionar la interacción entre médicos y pacientes, desde el momento en que un paciente es registrado y atendido hasta su posible internación y posterior egreso (delegado al ModuloEgreso). Contiene las siguientes clases:
    - ✓ SistemaAtencion, clase en la que se coordinan las operaciones entre distintos servicios especializados:
    - ✓ ServicioPacientes, encargado de registrar a los pacientes, iniciar sus atenciones y mantener actualizado su registro durante todo el proceso.
    - ✓ ServicioMedicos, responsable de registrar a los médicos, asociar las atenciones realizadas y permitir la consulta de historiales o reportes por período.
    - ✓ ServicioInternaciones, que administra las internaciones, verificando la capacidad de las habitaciones y registrando los días de estadía de cada paciente.
    - ✓ ServicioAsociados: responsable de registrar y eliminar asociados.

  Este módulo se apoya en diversas clases del sistema (como Habitacion, RegistroPaciente

ConsultaMedica) y en la gestión de excepciones específicas, asegurando que solo se ejecuten acciones válidas (por ejemplo, que un paciente esté previamente registrado antes de ser atendido o internado).

El Módulo de Atención representa el núcleo operativo de la clínica: controla la atención médica, registra las consultas, gestiona internaciones y mantiene la trazabilidad completa de la relación entre pacientes y profesionales dentro del sistema.

- ModuloEgreso: administra los procesos relacionados con la salida del paciente, como la facturación y en caso de haber sido internado, la desocupación de la habitación correspondiente. Contiene la clase SistemaEgreso.
- ModuloIngreso: se encarga del ingreso del paciente a la clínica, determinando si será dirigido a la sala de espera privada o al patio, según ciertas prioridades. Contiene la clase SistemaIngreso.

❖ Paquete prueba:

Contiene las clases principales: MainConsola, MainVentana desde las cuales se ejecuta el sistema.

❖ Paquete personas: Incluye las clases que representan a los individuos dentro de la clínica.

- ✓ Persona que será extendida desde Paciente y Medico (también clases de este paquete).
- ✓ Paciente que representa a las personas que reciben atención médica dentro del sistema. Es una clase abstracta que extiende a la clase base Persona, heredando sus atributos generales (como DNI, nombre, apellido, dirección y teléfono) y añadiendo características propias del contexto clínico, como el número de historia clínica y un número de orden.

  Su diseño abstracto permite definir un comportamiento común para todos los pacientes, dejando que las subclases concreten las reglas específicas según el tipo de paciente (niño, joven o mayor).
- ✓ Medico: clase que representa al profesional encargado de la atención de los pacientes dentro de la clínica. Extiende a la clase Persona, heredando sus atributos personales básicos (DNI, nombre, apellido, domicilio y teléfono), e implementa la interfaz IMedico, lo que le permite integrarse con el sistema de cálculo de honorarios definido en el paquete de honorarios.

  A los datos generales de una persona se le añaden atributos propios, como la matrícula y la especialidad, los cuales son utilizados para registrar y clasificar a los médicos dentro del sistema.

  El método calcularHonorarios() devuelve un valor base de 20.000, que representa la tarifa inicial del médico. Este valor puede ser modificado posteriormente

mediante el uso de decoradores (por ejemplo, por tipo de contratación, especialidad o posgrado), siguiendo el patrón de diseño Decorator implementado en el sistema y explicado luego en el presente informe.

De esta manera, la clase Medico sirve como punto de partida para la representación de los médicos dentro del modelo, interactuando con los módulos de atención, facturación y honorarios.

- ✓ Mayor es la clase que representa a un paciente de mayor dentro del sistema de la clínica. Extiende de la clase abstracta Paciente, heredando sus atributos y comportamientos generales, y definiendo su propia lógica particular para determinar si puede reemplazar a otro paciente dentro de la sala de espera privada.
- ✓ Ninio: igual que con Mayor pero con diferentes reglas.
- ✓ Joven: igual que con Mayor y Ninio pero con diferentes reglas.
- ✓ Domicilio: esta clase es para instanciar a las direcciones, ya sea de la clínica, de cada paciente o de cada médico.
- ✓ Asociado: quien se registra en la clínica para poder acceder al servicio de la ambulancia.
- ✓ Operario: Empleado encargado de solicitar y coordinar el mantenimiento de la ambulancia.

❖ Paquete observadores_y_observables:

- ✓ ObservableAsociado: representa un asociado que puede generar eventos. Usa setChanged() y notifyObservers(mensaje) para enviar actualizaciones.
- ✓ ObservadorAsociado: Este observar recibe los eventos y se los pasa a la vista llamando a vista.actualizarConsolaAsociado(this,mensaje).
- ✓ ObservadorAmbulancia: Observa el estado de la ambulancia, cuando esta cambia de estado se actualiza automáticamente en la vista.
- ✓ ObservableHilo: Asociado a un hilo de ejecución, es quien avisa que un hilo terminó.
- ✓ ObservadorHilos: Supervisa hilos de atención y notifica a la simulación cuando alguno termina.
- ✓ ObservadorOperario: observa el hilo de un operario, y cuando uno en particular termina avisa a SimulacionAmbulancia que debe cerrar ese operario.

❖ Paquete lugares: Aquí nos encontramos con algunas de las diferentes áreas que contiene la clínica. Las clases incluidas en este paquete son:

- ✓ Habitacion: que es extendida desde HabitacionCompartida, HabitacionPrivada y

HabitacionTerapiaIntensiva (estas tres clases tambien son de este mismo paquete). Cada una de estas calcula el costo teniendo en primer lugar un costo de asignación, costo base del tipo de habitación y una regla específica según la habitación.

- ✓ SalaDeEspera: Su método principal, ingresar(Paciente paciente), determina el destino del paciente al llegar:
  - Si la sala de espera privada está vacía, el paciente ingresa directamente a ella.
  - Si está ocupada, se evalúa si el nuevo paciente puede reemplazar al que ya está en la sala (según la lógica definida en las clases Niño, Joven y Mayor).
  - En caso de no poder reemplazarlo, el paciente se envía al patio.

  La clase también maneja la atención y retiro de pacientes, mediante los métodos sacarPaciente() y sacarPacienteConMenorOrden(), lanzando excepciones personalizadas cuando la sala está vacía o el paciente no se encuentra en espera.

  De esta forma, SalaDeEspera actúa como un coordinador general, controlando los movimientos entre los distintos espacios y manteniendo la coherencia del flujo de atención.

  Contiene dos atributos de tipo SalaDeEsperaPrivada y Patio que son también dos clases de este mismo paquete.
- ✓ SalaDeEsperaPrivada: representa un área exclusiva donde solo puede haber un paciente a la vez**.**
- ✓ Patio: representa el área común de espera donde permanecen los pacientes que no pueden acceder a la sala de espera privada. Internamente, mantiene una lista de pacientes mediante un ArrayList<Paciente> y proporciona métodos para:
  - Agregar un paciente (agregarPaciente()).
  - Remover un paciente cuando es atendido (sacarPaciente()), controlando condiciones de error mediante las excepciones SalaEsperaVaciaExcepcion y PacienteNoEstaEsperandoExcepcion.
  - Obtener la lista actual de pacientes (getPacientes()).

Esta clase cumple un rol complementario a la sala privada, asegurando que todos los pacientes en espera estén correctamente registrados y puedan ser gestionados según el orden o la prioridad establecida.

- ❖ Paquete interfaces:
  - ✓ IMedico: componente a ser envuelto, utilizado para el patrón Decorator.
  - ✓ IAmbulanciaState: establece el contrato de comportamiento que deben cumplir todos los estados de la ambulancia, permitiendo que cada estado implemente sus respuestas y restricciones de manera independiente.

- ❖ Paquete: honorarios

Agrupa las clases utilizadas para implementar el patrón Decorator, explicado en otra sección del informe. Las clases que se incluyen en este paquete son:

- ✓ EspecialidadCirugia: al calcular el honorario tiene un 10% de aumento con respecto al valor base.
- ✓ EspecialidadClinica: al calcular el honorario tiene un 5% de aumento con respecto al valor base.
- ✓ EspecialidadPediatria: al calcular el honorario tiene un 7% de aumento con respecto al valor base.
- ✓ PosgradoDoctorado: al calcular el honorario tiene un 10% de aumento con respecto al valor que incluye especialidad.
- ✓ PosgradoMaster: al calcular el honorario tiene un 5% de aumento con respecto al valor que incluye especialidad.
- ✓ ContratacionPermanente: al calcular el honorario tiene un 10% de aumento con respecto al valor que incluye especialidad y posgrado.
- ✓ ContratacionResidente: al calcular el honorario tiene un 5% de aumento con respecto al valor que incluye especialidad y posgrado.

- ❖ Paquete facturación_y_registros: Aquí se manejan los cálculos de las facturas de los pacientes y el registro de sus atenciones. Incluye:
  - ✓ ConsultaMedica: representa cada consulta individual realizada a un paciente.
  - ✓ Factura: se encarga de generar el documento final que resume los gastos asociados a la atención de un paciente.

    A partir de los datos de un objeto RegistroPaciente, la factura incluye:
    - Información personal del paciente.
    - Fechas de ingreso y egreso.

- Costo de internación (si corresponde), calculado en base al tipo de habitación y cantidad de días.
- Detalle de las consultas médicas realizadas, mostrando el nombre del médico, especialidad y subtotal.

Durante la generación, se aplica además un incremento del 20% en los honorarios médicos, simulando los ajustes administrativos que la clínica realiza sobre el valor base.

La clase asigna un número de factura único y correlativo, y permite imprimir el resultado a través del método ImprimeFactura(), que devuelve el contenido.

- ✓ PacienteAtendido: registra los datos básicos de un paciente que ya fue atendido, incluyendo la fecha de atención y el monto del honorario médico correspondiente.

Estas instancias se utilizan para elaborar reportes de los médicos y mantener un historial de la actividad médica realizada.

- ✓ RegistroPaciente: mantiene un seguimiento completo de las atenciones recibidas por un paciente. Contiene la fecha de ingreso, la cantidad de días internado, una posible habitación asignada, y una lista de consultas médicas registradas a través del método agregarConsultaMedica(). Este registro es fundamental para el cálculo posterior de los costos totales y para la generación de la factura final del paciente.
- ✓ ReporteActividadMedica: genera un informe detallado de las consultas efectuadas por un determinado médico dentro de un rango de fechas. Aquí se muestra: fecha, nombre y apellido del paciente y el honorario de cada consulta (sin el incremento del 20% que se comentó anteriormente en la clase Factura). Por último, se muestra el total de honorarios y cantidad de consultas.

❖ Paquete factoria:

Nos encontramos con dos clases que manejaran el patrón Factory, que luego será explicado con mayor detalle junto con los demás patrones anteriormente nombrados. Estas dos clases son:

- ✓ FactoryPaciente
- ✓ FactoryMedico

❖ Paquete: excepciones:

Agrupa todas las excepciones utilizadas a lo largo del proyecto.

✓ Paquete ambulancia:

- ✓ Ambulancia: mantiene el estado actual de la ambulancia, atiende solicitudes y notifica cambios a sus observadores.
- ✓ AtendiendoADomState: un estado de la ambulancia.
- ✓ DisponibleState: un estado de la ambulancia.
- ✓ EnTallerState: un estado de la ambulancia.
- ✓ HiloAmbulancia: representa la actividad de un asociado solicitando una ambulancia.
- ✓ RegresandoDelTallerState: un estado de la ambulancia.
- ✓ RegresandoSinPacienteState: un estado de la ambulancia.
- ✓ RetornoAutomatico: pide el retorno de la ambulancia a la clínica, evita que la ambulancia quede bloqueada en un estado y garantiza que el sistema pueda finalizar correctamente.
- ✓ SimulacionAmbulancia: coordinador general de la simulación, crea hilos, controla la finalización y se comunica con la vista.
- ✓ TraslandoPacienteState: un estado de la ambulancia.

- **Paquete: persistencia**

La capa de persistencia cumple la función de mantener la información del sistema almacenada de forma permanente, permitiendo que los datos sobrevivan al cierre de la aplicación y puedan ser recuperados en sesiones posteriores. Esta capa actúa como un intermediario entre el modelo de dominio y la base de datos MySQL. En este programa se aplica únicamente para la persistencia de asociados.

El proceso de persistencia se basa en tres clases principales que trabajan en conjunto:

El primero es el objeto AsociadoDTO, que funciona como un contenedor simple y plano que transporta únicamente la

información necesaria para almacenar, sin incluir ninguna lógica de negocio

El segundo componente es el AsociadoMapper, cuya función es transformar los objetos del dominio en objetos de transferencia y viceversa. Cuando la información debe ser guardada, el convertidor extrae los datos relevantes del objeto complejo del dominio y los organiza en el formato simple requerido para la persistencia. Cuando la información debe ser recuperada, realiza el proceso inverso, tomando los datos planos almacenados y reconstruyendo los objetos del dominio con toda su estructura y relaciones.

El tercer componente es el gestor de acceso a datos, que se encarga de toda la comunicación con la base de datos. Su función principal es mantener una única instancia de conexion activa con la base de datos durante toda la ejecución de la aplicación.

Al iniciar la aplicación, el SistemaFacade obtiene la instancia única del DAO mediante dataBaseDAO.getInstancia(), El DAO lee la configuración desde Config.properties (URL, nombre de BD, usuario, contraseña), **para modificar las credenciales, acceder al archivo mencionado.** De esta manera si es posible establecer una conexión, el gestor consulta la base de datos y obtiene los registros almacenados, transformándolos en objetos DTO. Estos pasan por el convertidor, que los transforma nuevamente en objetos del dominio completos.

Aclaracion: Si es la primera vez que se ejecuta el programa, presionar el botón Crear Tablas, para establecer tanto en la memoria como en la base de datos una lista de Asociados iniciales.

Caso contrario al guardar información, el sistema transforma el objeto del dominio en su versión simplificada mediante el convertidor. El gestor de acceso recibe esta información y la almacena en la base de datos de forma estructurada. La conexión se mantiene activa durante toda la sesión, permitiendo múltiples operaciones sin necesidad de reestablecerla constantemente.

- ✓ AsociadoDTO: objeto simple y serializable, transporta datos entre

la base y capa de dominio y no contiene lógica de negocio.

- ✓ AsociadoMapper: convierte entre objetos del dominio y objetos DTO.
- ✓ DataBaseDAO: encargado de toda la lógica de acceso a la base de datos. Implementa Singleton.

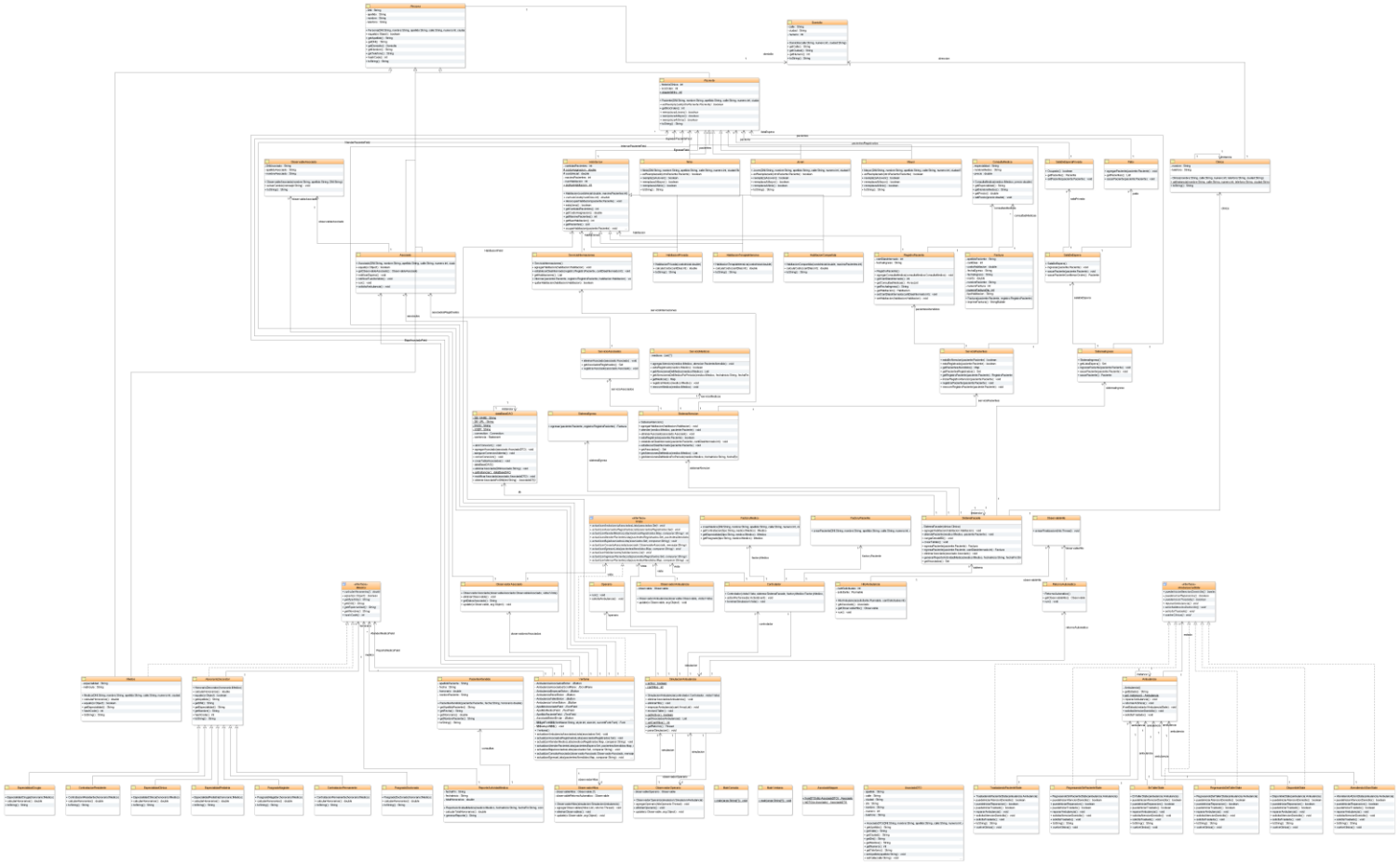
- **Paquete vista:**

❖ Componentes_vista: incluye clases auxiliares generadas por el diseñador gráfico de IntelliJ. Se encarga de manejar el layout visual.

- ✓ IVista: representa la vista del patrón MVC. Define todas las operaciones que la Vista ofrece al Controlador. Obtiene fatos del usuario, le muestra información al usuario, actualiza listas o paneles dinámicos, maneja la parte visual de la ambulancia y sus observadores, y delega eventos del usuario al Controlador.
- ✓ Ventana: interfaz gráfica construida con Swing. Es la implementación de la vista en el patrón MVC, no contiene la lógica del negocio.

Diagrama de clases (UML)

Diagrama completo del proyecto:



La imagen será subida al repositorio de GitHub para una mejor visualización de la misma.

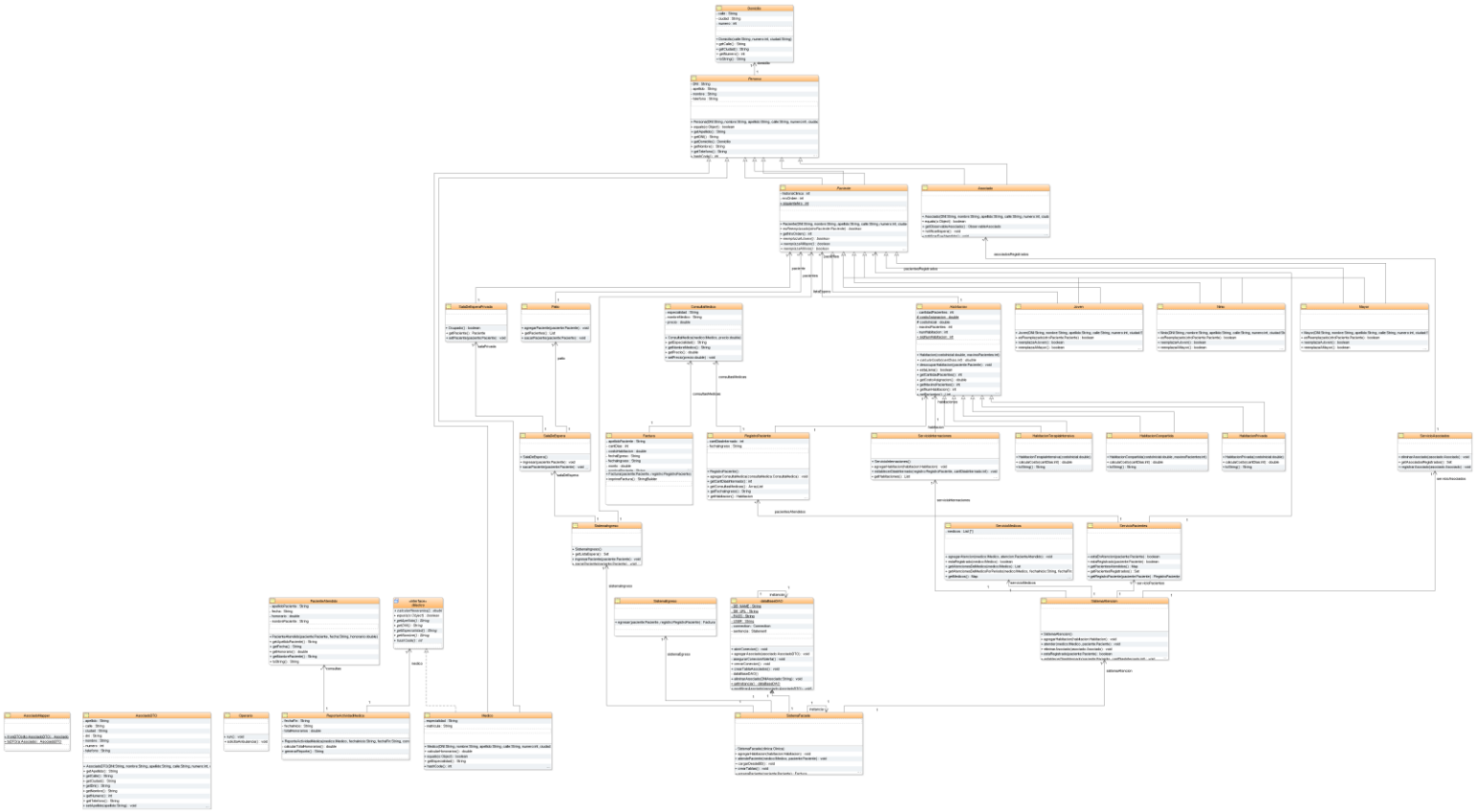
Documentación generada mediante Javadoc

Documentación HTML a partir de comentarios especiales en el código fuente de Java. Se ha utilizado para las clases y, en su mayoría para los métodos.

El JavaDoc será entregado en una carpeta subida al repositorio de GitHub junto al código para poder ingresar al HTML de forma correctamente visual entrando a index.html.

Patrones de diseño

➢ Patrón Facade:



La imagen será subida al repositorio de GitHub para una mejor visualización de la misma.

CÓDIGO REPRESENTATIVO (con solo algunos métodos)

```
public class SistemaFacade {

    private final Clinica clinica;

    private final SistemaAtencion sistemaAtencion;
    private final SistemaIngreso sistemaIngreso;
    private final SistemaEgreso sistemaEgreso;
    private dataBaseDAO db;
    private static SistemaFacade instancia;

   //Patrón Singleton
    private SistemaFacade(Clinica clinica) {
        this.clinica = clinica;
        this.sistemaIngreso = new SistemaIngreso();
        this.sistemaAtencion = new SistemaAtencion();
        this.sistemaEgreso = new SistemaEgreso();
        this.db = dataBaseDAO.getInstancia();
    }

    public static SistemaFacade getInstancia(Clinica clinica) {
        if (instancia == null) {
            instancia = new SistemaFacade(clinica);
        }
        return instancia;
    }

    public void registraMedico(IMedico medico) throws MedicoDuplicadoExcepcion {
```

```
            sistemaAtencion.registrarMedico(medico);
        }

        public void registraPaciente(Paciente paciente) throws PacienteDuplicadoExcepcion{
            sistemaAtencion.registrarPaciente(paciente);
        }

        public void registraAsociado(Asociado asociado) throws AsociadoDuplicadoExcepcion,
ErrorPersistenciaExcepcion {
            sistemaAtencion.registrarAsociado(asociado);
             db.agregarAsociado(AsociadoMapper.toDTO(asociado));
        }
```

En el proyecto se implementó este patrón con el objetivo de simplificar la interacción entre las distintas partes del sistema. Permite ocultar la complejidad interna de las clases y subsistemas promoviendo una mayor facilidad de mantenimiento y extensibilidad.

Disponemos de la clase SistemaFacade que cumple el rol de fachada, centralizando la comunicación entre: SistemaIngreso, SistemaAtencion y SistemaEgreso. Cada uno de estos módulos encapsula la lógica correspondiente a una etapa específica de la atención médica.

Desde aquí es donde se crean las tablas para asociados y se añaden a la base de datos.

En el método principal se instancia el sistema de la siguiente forma:

```
Clinica clinica = Clinica.getInstancia( nombre: "Clinica Central", direccion: "Av. Salud 123",
        telefono: "+54 11 1234-5678", ciudad: "Buenos Aires");
SistemaFacade sistema = new SistemaFacade(clinica);
```

A partir de este objeto sistema (instancia de SistemaFacade), se pueden ejecutar las operaciones como el registro tanto de médicos como de pacientes, el ingreso, atención e internación de estos últimos, generación de reportes y gestión del egreso generando la factura correspondiente.

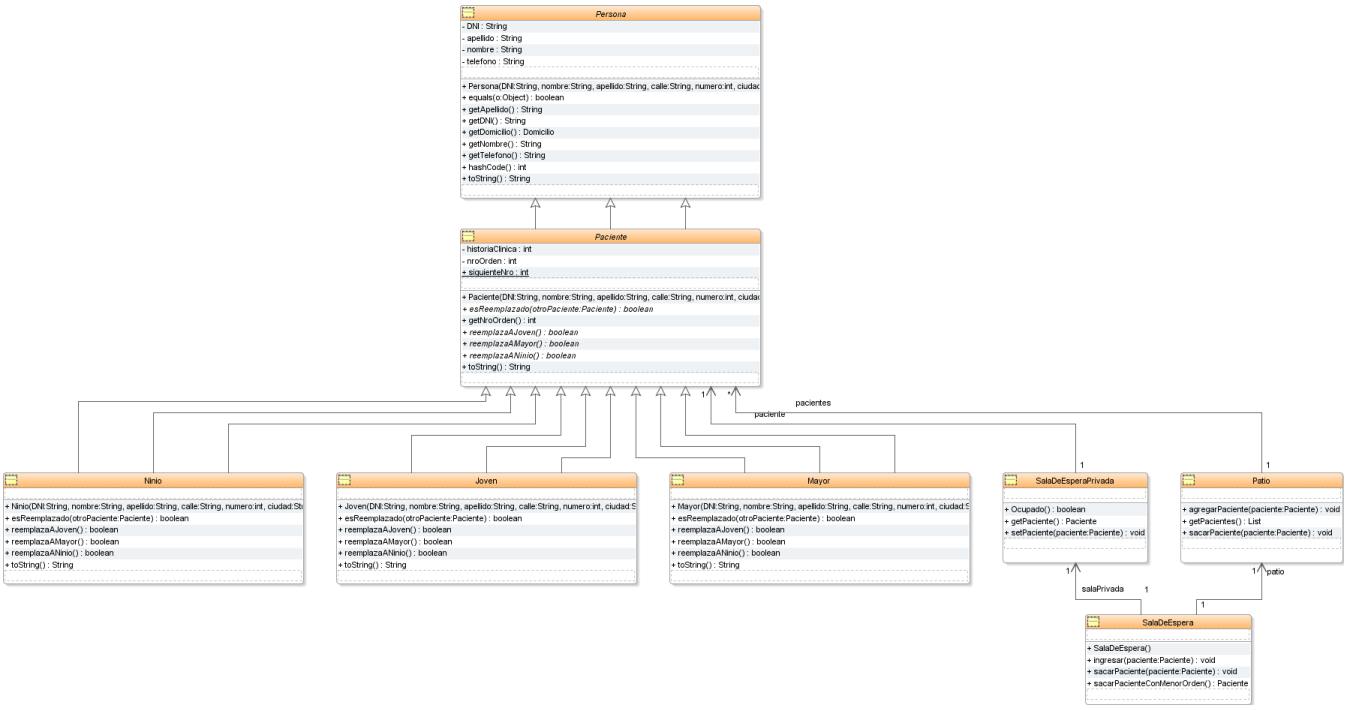
Además en este mismo método se crea la instancia del controlador, el cual en su constructor tiene al sistema:

```
Controlador controlador = new Controlador(v,sistemaFacade,factoryMedico,factoryPaciente);
```

```
public Controlador(IVista vista, SistemaFacade sistema, FactoryMedico factoryMedico, FactoryPaciente factoryPaciente)
{
    this.vista = vista;
    this.vista.addActionListener(this);
    this.sistema = sistema;
    this.simulacion = new SimulacionAmbulancia( controlador: this, vista);
    this.factoryMedico = factoryMedico;
    this.factoryPaciente = factoryPaciente;
}
```

Se puede ver que para la clase Clinica se implementa el patrón Singleton para asegurar la existencia de una única instancia. A continuación, explicaremos en mayor detalle el uso de este último patrón.

➢ Patrón Double Dispatch:



Este patrón es utilizado para la resolución de conflictos en la sala de espera que requiere determinar, de manera dinámica y según las prioridades de la clínica, la ubicación de un nuevo paciente (Sala de Espera Privada o Patio).

La lógica de asignación depende de dos tipos de objetos, el paciente que está en la sala privada (si no está vacía) y el paciente que ingresa.

Las prioridades eran:

- ✓ Entre un Niño y un Joven, la sala privada queda para el Niño y el otro se va al patio.
- ✓ Entre un Niño y un Mayor, la sala privada queda para el Mayor y el otro se va al patio.
- ✓ Entre un Mayor y un Joven, la sala privada queda para el Joven y el otro se va al patio.

El patrón permite que la operación se resuelva basándose en el tipo de dos objetos diferentes y no solo en uno. La interacción comienza cuando el paciente que ya está en la sala privada llama al método "esReemplazado(Paciente otroPaciente)", siendo otroPaciente el nuevo paciente que ingresa y este verá si reemplaza al existente en la sala privada o si va al patio.

Por ejemplo: si quien está en la sala privada es mayor, estamos en la clase Mayor que tenemos:

```
public boolean esReemplazado( @NotNull Paciente otroPaciente)
    return otroPaciente.reemplazaAMayor();
}
```

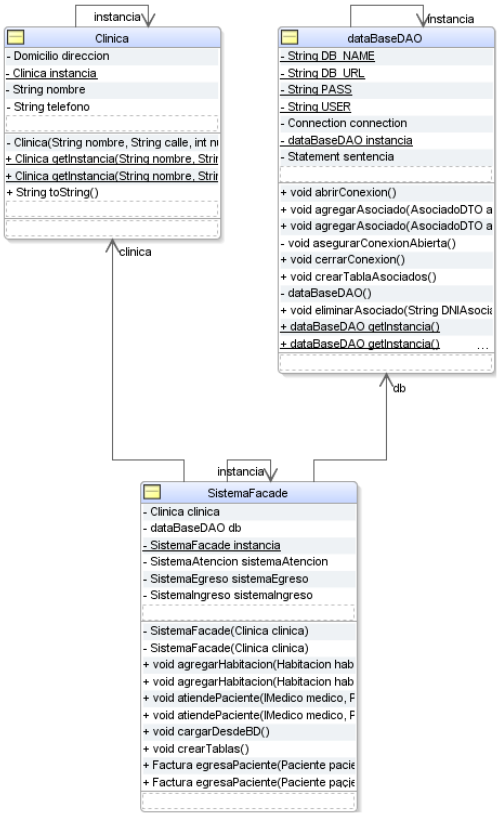
El método esReemplazado() llama al método reemplazaAMayor de otroPaciente. Si otroPaciente es mayor o un niño, no lo reemplazará. Solo un joven reemplaza al mayor. Esta lógica

sería igual para los demás tipos. Entonces si por ejemplo, el otroPaciente era un niño:

```
public boolean reemplazaAMayor()
    return false;
}
```

Lo cual indica que no reemplazará al mayor.

➢ Patrón Singleton:



Como se ha nombrado anteriormente, el objetivo es garantizar que exista una única instancia de la clase Clinica, SistemaFacade y la conexión a base de datos.

El patrón se implementa mediante las siguientes características principales:

- Un atributo estático privado que almacena la única instancia de la clase.
- Un constructor privado, el cual impide la creación de objetos desde fuera de la clase.
- Un método público estático que devuelve esa única instancia existente, creándola en el caso de que no exista.

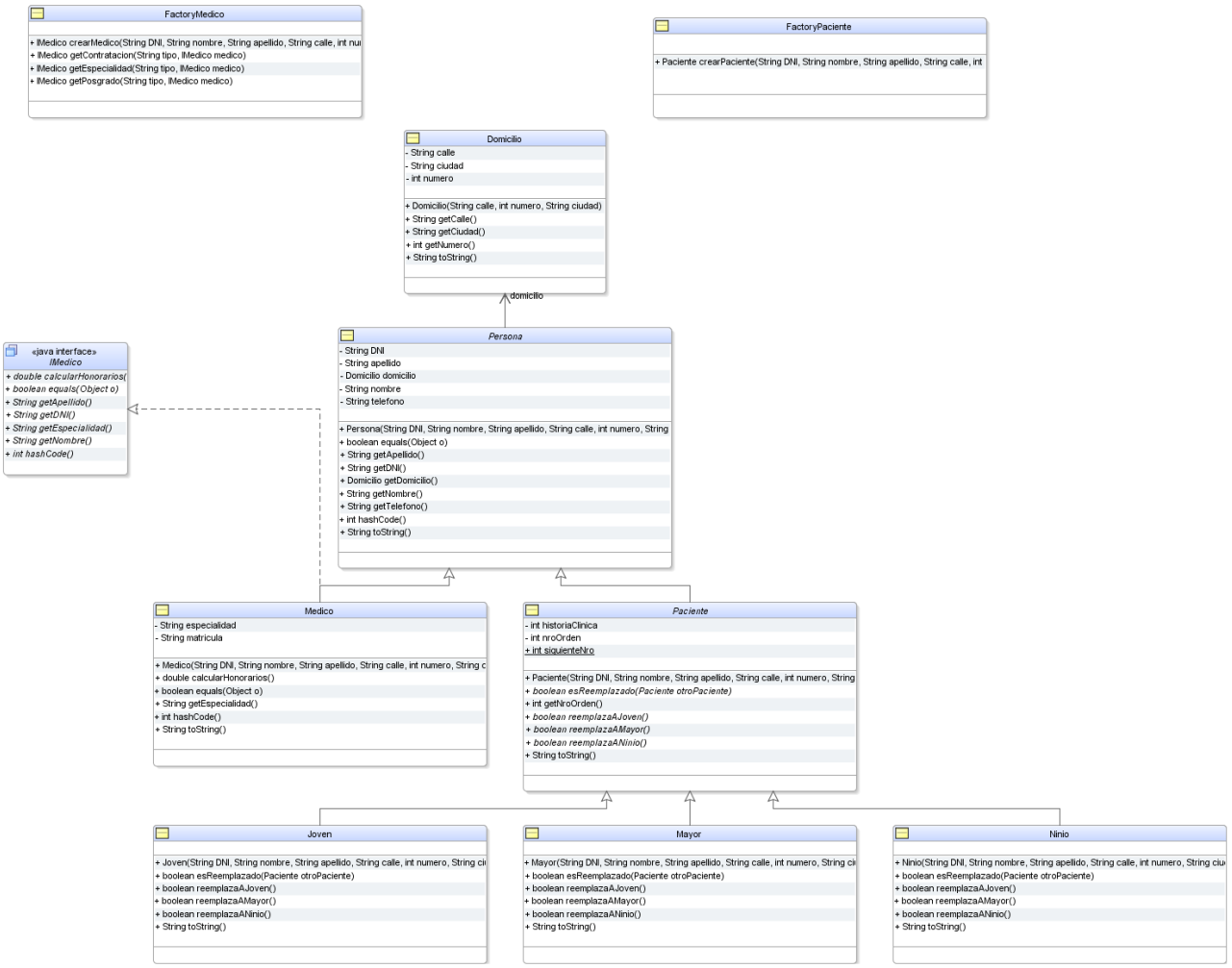
```
private static Clinica instancia;  3 usages

@Contract(pure = true)
private Clinica(String nombre, String direccion, String telefono,
                String ciudad) {
    this.nombre = nombre;
    this.direccion = direccion;
    this.telefono = telefono;
    this.ciudad = ciudad;
}
public static Clinica getInstancia(String nombre, String direccion,
                                   String telefono, String ciudad) {
    if (instancia == null) {
        instancia = new Clinica(nombre, direccion, telefono, ciudad);
    }
    return instancia;
}
```

```java
private static SistemaFacade instancia;  3 usages

private SistemaFacade(Clinica clinica) {
    this.clinica = clinica;
    this.sistemaIngreso = new SistemaIngreso();
    this.sistemaAtencion = new SistemaAtencion();
    this.sistemaEgreso = new SistemaEgreso();
}

public static SistemaFacade getInstancia(Clinica clinica) {
    if (instancia == null) {
        instancia = new SistemaFacade(clinica);
    }
    return instancia;
}
```

```java
private static dataBaseDAO instancia;  3 usage

private dataBaseDAO() {  1 usage  Lara
    DB_URL = Config.get("db.url");
    DB_NAME = Config.get("db.name");
    USER = Config.get("db.user");
    PASS = Config.get("db.password");
}

public static dataBaseDAO getInstancia() {
    if  (instancia == null) {
        instancia = new dataBaseDAO();
    }
    return instancia;
}
```

Entonces cualquier intento posterior para obtener una instancia de la clínica, del sistema o de la base de datos, retornará esta misma ya que ya existe la única instancia posible, sin generar nuevos objetos.

➢ Patrón Factory



Este patrón fue implementado con el objetivo de centralizar y simplificar la creación de aquellos objetos que presentas múltiples variantes según sus características, como lo son los médicos y los pacientes.

En el programa principal la creación de los objetos se realiza a través de las instancias de las clases FactoryMedico y FactoryPaciente de la siguiente manera:

```
FactoryMedico factoryMedico = new FactoryMedico();
FactoryPaciente factoryPaciente = new FactoryPaciente();
```

La clase FactoryMedico encapsula toda la lógica para crear un médico con las combinaciones posibles de especialidad, contratación y posgrado, aplicando los decoradores correspondientes que modifican el cálculo de los honorarios de los médicos (Patrón Decorator que se explicará luego).

Del mismo modo, FactoryPaciente crea instancia del tipo paciente que corresponde según su rango etario.

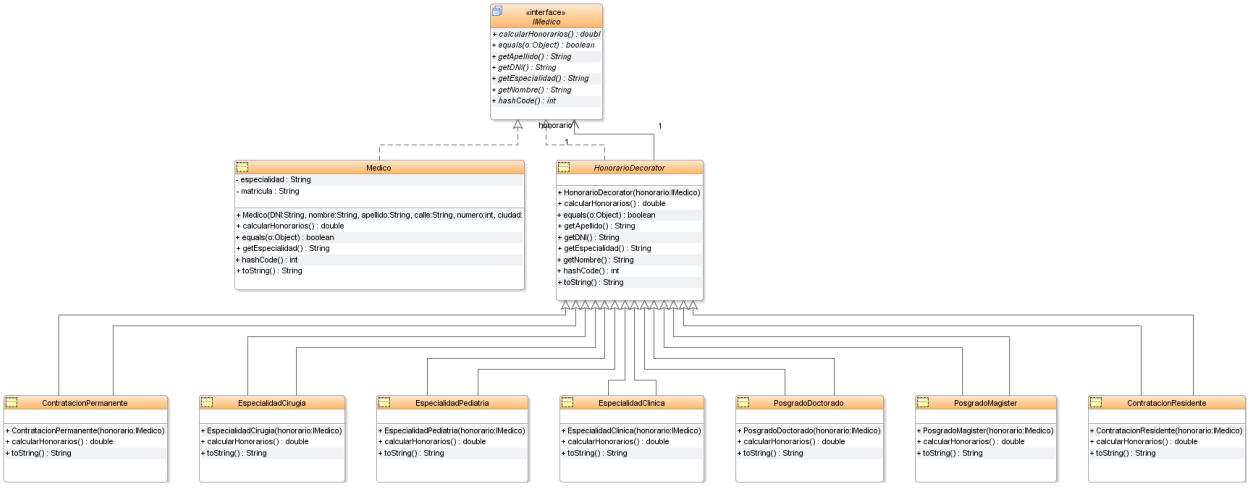
Se muestran algunos ejemplos:

```
//Algunos medicos y pacientes registrados para iniciar el programa
//Medicos
IMedico med1 = factoryMedico.crearMedico( DNI: "10000000", nombre: "Ana", apellido: "García", calle: "Calle 1", numero: 1,
        ciudad: "CABA", telefono: "111-1111", matricula: "M-123", especialidad: "CLINICA", contratacion: "PERMANENTE", posgrado: "MASTER");
IMedico med2 = factoryMedico.crearMedico( DNI: "10000001", nombre: "Bruno", apellido: "López", calle: "Calle 2", numero: 2,
        ciudad: "CABA", telefono: "222-2222", matricula: "M-456", especialidad: "CIRUGIA", contratacion: "RESIDENTE", posgrado: "DOCTORADO");
IMedico med3 = factoryMedico.crearMedico( DNI: "10000002", nombre: "Bianca", apellido: "González", calle: "Calle 3", numero: 3,
        ciudad: "CABA", telefono: "333-3333", matricula: "M-534", especialidad: "PEDIATRIA", contratacion: "PERMANENTE", posgrado: "DOCTORADO");
IMedico med4 = factoryMedico.crearMedico( DNI: "10000003", nombre: "Roque", apellido: "Pérez", calle: "Calle 4", numero: 4,
        ciudad: "CABA", telefono: "444-4444", matricula: "M-765", especialidad: "CIRUGIA", contratacion: "RESIDENTE", posgrado: "MASTER");
IMedico med5 = factoryMedico.crearMedico( DNI: "10000004", nombre: "Oscar", apellido: "Rodriguez", calle: "Calle 5", numero: 5,
        ciudad: "CABA", telefono: "555-5555", matricula: "M-876", especialidad: "CLINICA", contratacion: "PERMANENTE", posgrado: "NINGUNO");
IMedico med6 = factoryMedico.crearMedico( DNI: "10000005", nombre: "Pedro", apellido: "Fernández", calle: "Calle 6", numero: 6,
        ciudad: "CABA", telefono: "666-6666", matricula: "M-234", especialidad: "CIRUGIA", contratacion: "RESIDENTE", posgrado: "MASTER");
IMedico med7 = factoryMedico.crearMedico( DNI: "10000006", nombre: "Andres", apellido: "Gutiérrez", calle: "Calle 7", numero: 7,
        ciudad: "CABA", telefono: "777-7777", matricula: "M-543", especialidad: "PEDIATRIA", contratacion: "PERMANENTE", posgrado: "NINGUNO");

//Pacientes
Paciente p1 = factoryPaciente.crearPaciente( DNI: "20000000", nombre: "Juan", apellido: "Pérez", calle: "Calle 10", numero: 10, ciudad: "CABA", telefono: "300-0000", historiaClinica: 12345, rangoEtario: "JOVEN");
Paciente p2 = factoryPaciente.crearPaciente( DNI: "20000001", nombre: "Lucía", apellido: "Suárez", calle: "Calle 11", numero: 11, ciudad: "CABA", telefono: "300-0001", historiaClinica: 22345, rangoEtario: "NINIO");
Paciente p3 = factoryPaciente.crearPaciente( DNI: "20000002", nombre: "Mario", apellido: "Gómez", calle: "Calle 12", numero: 12, ciudad: "CABA", telefono: "300-0002", historiaClinica: 32345, rangoEtario: "MAYOR");
Paciente p4 = factoryPaciente.crearPaciente( DNI: "20000003", nombre: "Paulo", apellido: "Moreno", calle: "Calle 13", numero: 13, ciudad: "CABA", telefono: "300-0003", historiaClinica: 42345, rangoEtario: "JOVEN");
Paciente p5 = factoryPaciente.crearPaciente( DNI: "20000004", nombre: "Rocio", apellido: "Flores", calle: "Calle 14", numero: 14, ciudad: "CABA", telefono: "300-0004", historiaClinica: 52345, rangoEtario: "NINIO");
Paciente p6 = factoryPaciente.crearPaciente( DNI: "20000005", nombre: "Martin", apellido: "Torres", calle: "Calle 15", numero: 15, ciudad: "CABA", telefono: "300-0005", historiaClinica: 62345, rangoEtario: "MAYOR");
```

A través de este enfoque el cliente no necesita conocer que clases concretas intervienen en la creación de cada objeto, sino que solo debe conocer los datos necesarios.

➢ Patrón Decorator

Este es utilizado para calcular los honorarios de los médicos de acuerdo con las distintas combinaciones de características (especialidad, contratación y posgrado).



En primer lugar tenemos la interfaz IMedico que define el contrato común que deben cumplir todos los tipos de médicos, incluyendo el método calcularHonorarios() que determina el monto a percibir según las características anteriormente nombradas.

En segundo lugar tenemos el componente concreto que en este caso sería la clase Medico que sobrescribe el método calcularHonorarios() dándole el valor base.

En tercer lugar, tenemos la clase abstracta HonorarioDecorator que implementa dicha interfaz y actúa como decorador base.

Y por último, tenemos los decoradores concretos que representan las distintas condiciones del médico: EspecialidadClinica, EspecialidadCirugia, EspecialidadPediatria,

ContratacionPermanente, ContratacionResidente, PosgradoMaster, PosgradoDoctorado. Cada uno de estos decoradores sobrescribe el método calcularHonorarios() aplicando el incremento correspondiente según su tipo, los cuales serán acumulativos con respecto a cada característica (especialidad, contratación y posgrado).

```java
public class Medico extends Persona implements IMedico {
    /**
     * Método para comenzar a calcular modelo.honorarios
     * @return valor base de los modelo.honorarios: 20000
     */
    public double calcularHonorarios() { return 20000; }
```

```java
public interface IMedico {
    public double calcularHonorarios();  9 imple
    public String getDNI();  2 implementations
    public String getNombre();  2 implementations
    public String getApellido();  2 implementatior
    public String getEspecialidad();  2 implemer
    @Contract(value = "null -> false", pure = true)
    public boolean equals(Object o);  2 implemer
    public int hashCode();  2 implementations
```

Ejemplo de una de las clases decoradoras:

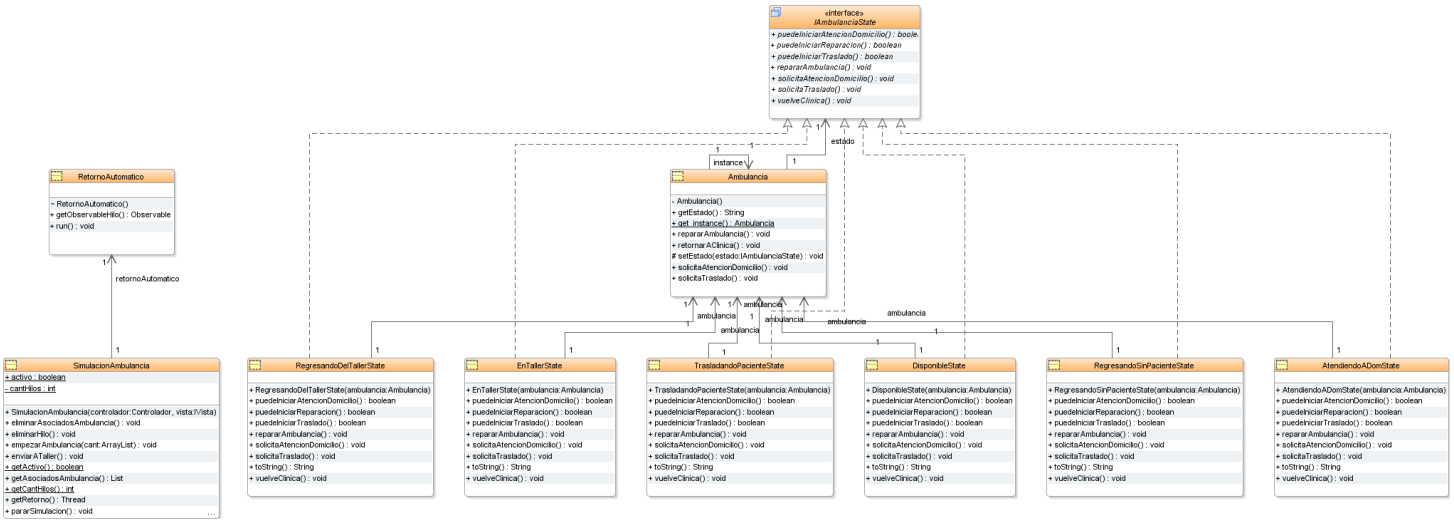
```java
public class ContratacionPermanente extends HonorarioDecorator {  1 usage  Máxima Maiolo +1
    @Contract("null -> fail")
    public ContratacionPermanente(IMedico honorario) { super(honorario); }

    /**
     * Cuando contratacion es Permanente, hay un 10% de aumento sobre el honorario que incluye la especialidad y el posgrado
     * @return honorario con el aumento de la contratación
     */
    @Override  CRSerafini +1
    public double calcularHonorarios() {
        double honorarioConAumento = super.calcularHonorarios() * 1.1;

        assert honorarioConAumento >= 0 : "El honorario calculado no puede ser negativo";

        return honorarioConAumento;
    }

    @Override  Máxima Maiolo
    public String toString() { return " Contratacion Permanente" + super.toString(); }
}
```

➢ Patrón State

El patrón State permite que un objeto cambie su comportamiento dinámicamente según el estado interno en el que se encuentre. Cada estado representa una forma distinta de responder a las mismas acciones. En este caso es la Ambulancia la que delega su comportamiento.

La ambulancia puede encontrar en varias situaciones diferentes y cada una de estas determina que acciones están o no permitidas y como debe reaccionar la ambulancia.

Todas las clases de estado implementan una interfaz común que es IAmbulanciaState y heredan de Ambulancia.




```java
public interface IAmbulanciaState {  15 usages

    void solicitaAtencionDomicilio();  1 usage

    void solicitaTraslado();  1 usage  6 implemer

    void vueleClinica();  1 usage  6 implementati

    void repararAmbulancia();  1 usage  6 impleme

    boolean puedeIniciarAtencionDomicilio();

    boolean puedeIniciarTraslado();  1 usage  6

    boolean puedeIniciarReparacion();  1 usage

}
```

La interfaz establece el contrato de comportamiento que deben cumplir todos los estados de la ambulancia, permitiendo que cada estado implemente sus respuestas y restricciones de manera independiente.

```java
public class RegresandoDelTallerState implements IAmbulanciaState {
    private Ambulancia ambulancia;

    @Contract(value = "null -> fail", pure = true)
    public RegresandoDelTallerState(Ambulancia ambulancia) {
        assert ambulancia != null : "La ambulancia no puede ser null";
        this.ambulancia = ambulancia;
    }

    @Override
    public void solicitaAtencionDomicilio() {
        // No disponible en este estado
    }

    @Override
    public void solicitaTraslado() {
        // No disponible en este estado
    }

    @Override
    public void vuelveClinica() { ambulancia.setEstado(new DisponibleState(ambulancia)); }

    @Override
    public void repararAmbulancia() {
        // No disponible en este estado
    }

    @Override
    public boolean puedeIniciarAtencionDomicilio() { return false; }

    @Override
    public boolean puedeIniciarTraslado() { return false; }

    @Override
    public boolean puedeIniciarReparacion() { return false; }

    @Override
    public String toString() { return "Regresando del Taller"; }
}
```

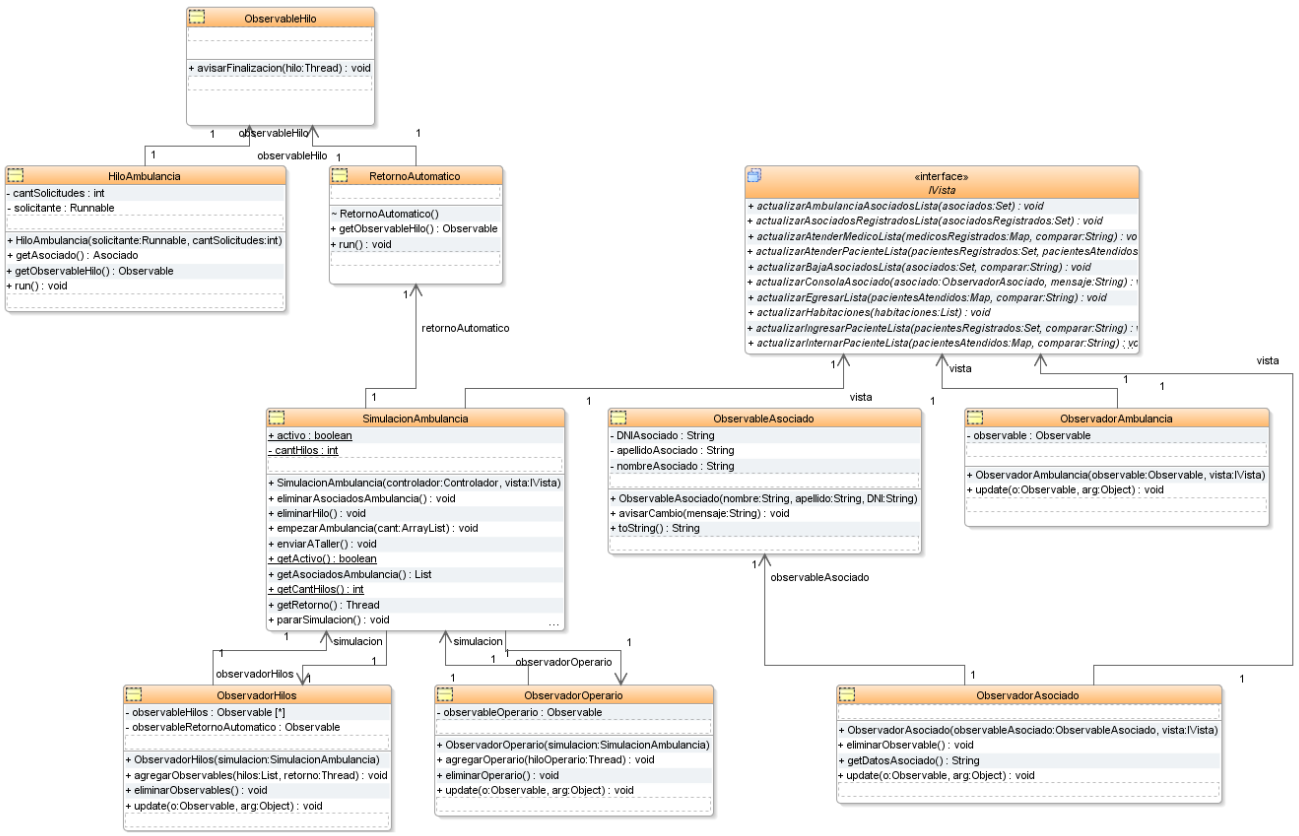
A modo de ejemplo de uno de los estados tenemos la clase RegresandoDelTallerState, que representa el estado en el que la ambulancia está volviendo a la clínica luego de haber sido reparada. Entonces define como debe comportarse la clínica mientras se encuentra en este estado.

Mientras la ambulancia está regresando del taller no puede iniciar atención a domicilio, ni tampoco iniciar traslado o volver a reparación. La única acción válida es vuelveClinica cambiando su estado a Disponible.

➢ Patrón Observer

El patrón Observer se usa cuando un objeto (llamado Observable) debe notificar automáticamente a otros objetos (llamados Observadores) cada vez que ocurre un evento relevante, sin necesidad de que el observable sepa detalles de quién lo observa. Entonces el observable avisa y los observers reaccionan.

El patrón mantiene sincronizada la vista con el modelo, permitiendo que distintos componentes del sistema reciban notificaciones automáticas cuando ocurren cambios en el modelo. Garantizando así, que la vista muestre siempre información actualizada sin necesidad de consultar continuamente el modelo. También se utiliza para coordinar la finalización de los hilos.




```java
public class ObservadorAsociado implements Observer {  8 usages
    private ObservableAsociado observableAsociado;  3 usages
    private IVista vista;  2 usages

    public ObservadorAsociado( @NotNull ObservableAsociado observableAsociado, IVista vista) {
        this.vista = vista;
        this.observableAsociado = observableAsociado;
        assert observableAsociado != null;
        assert vista != null;
        observableAsociado.addObserver( o: this);
    }
```

```java
    /**
     * Recibe actualizaciones del observable y las envía a la vista.
     */
    @Override  Máxima Maiolo
    public void update(Observable o, Object arg) {
        this.vista.actualizarConsolaAsociado( asociado: this, (String) arg);
    }
}
```

```java
public class ObservadorAmbulancia implements Observer {  3 usages
    private Observable observable;  2 usages
    private IVista vista;  2 usages

    public ObservadorAmbulancia(Observable observable, IVista vista) {
        this.observable = observable;
        this.observable.addObserver( o: this);
        this.vista = vista;
    }
```

```java
/**
 * Se ejecuta cuando la ambulancia cambia de estado.
 *
 * @param o   Observable que generó la notificación.
 * @param arg
 */
@Override  Máxima Maiolo
public void update(Observable o, Object arg) {
    this.vista.cambiarEstadoAmbulancia(Ambulancia.get_instance().getEstado());
}
```

```java
public class ObservadorHilos implements Observer {  3 usages
    private ArrayList<Observable>  observableHilos = new ArrayList<>();
    private SimulacionAmbulancia simulacion;  2 usages
    private Observable observableRetornoAutomatico;  3 usages

    public ObservadorHilos(SimulacionAmbulancia simulacion) {
        this.simulacion = simulacion;
    }
```

```java
/**
 * Llamado cuando un hilo notifica su finalización.
 */
@Override
public void update(Observable o, Object arg) {
    this.simulacion.eliminarHilo();
}
```

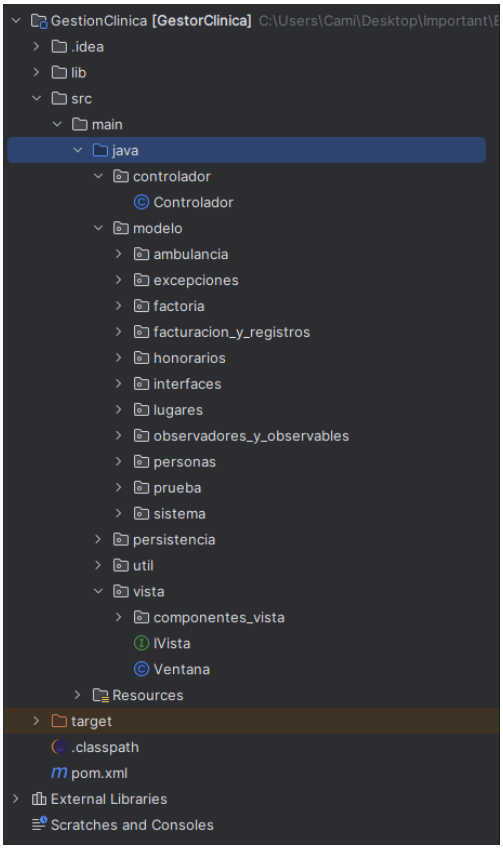
```java
public class ObservadorOperario implements Observer {  3 usages
    private SimulacionAmbulancia simulacion;  2 usages
    private Observable observableOperario;  3 usages

    @Contract(pure = true)
    public ObservadorOperario(SimulacionAmbulancia simulacion) {
        this.simulacion = simulacion;
        assert simulacion != null;
    }
```

```java
/**
 * Se ejecuta cuando el hilo observado finaliza.
 */
@Override
public void update(Observable o, Object arg) {
    this.simulacion.terminarOperario();
}
```

➢ Patrón MVC

En el proyecto se implementa el patrón MVC para separar responsabilidades y mantener el código organizado, escalable y fácil de mantener. El modelo contiene la lógica del dominio de la clínica, la vista es la GUI del proyecto (muestra datos al usuario, recibe notificaciones de este, no ejecuta lógica de negocio y depende del Controlador para interactuar con el modelo), y el Controlador es el intermediario entre los dos anteriores.

Si el usuario presiona un botón en la vista, esta lo notifica al controlador y este llama al sistema facade que maneja los distintos módulos del modelo, donde se ejecuta la lógica. El controlador recibe los nuevos datos y ordena a la Vista que se actualice la interfaz gráfica.



MODELO

```java
public class SistemaFacade {  11 usages

    private final Clinica clinica;  1 usage
    private final SistemaAtencion sistemaAtencion;  24 usages
    private final SistemaIngreso sistemaIngreso;  4 usages
    private final SistemaEgreso sistemaEgreso;  3 usages
    private dataBaseDAO db;  8 usages
    private static SistemaFacade instancia;  4 usages

    @Contract("null -> fail")
    private SistemaFacade(Clinica clinica) {  1 usage
        assert clinica!=null: "La clinica no puede ser null";
        this.clinica = clinica;
        this.sistemaIngreso = new SistemaIngreso();
        this.sistemaAtencion = new SistemaAtencion();
        this.sistemaEgreso = new SistemaEgreso();
        this.db = dataBaseDAO.getInstancia();
    }

    @Contract("null -> fail")
    public static SistemaFacade getInstancia(Clinica clinica) {  2 usages
        assert clinica!=null: "La clinica no puede ser null";
        if (instancia == null) {
            instancia = new SistemaFacade(clinica);
        }
        assert instancia!= null : "Si no existia la instancia se creo una entonces de ninguna forma puede ser null";
        return instancia;
    }
```

VISTA

```
public Ventana() {  1 usage
    setContentPane(MainPanel);
    setTitle("Sistema Clínica");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize( width: 1600,  height: 800);
    setLocationRelativeTo(null);
    setVisible(true);
    EstadoText.setText("DISPONIBLE");
    PanelAmbu.setLayout(new BoxLayout(PanelAmbu, BoxLayout.Y_AXIS));
    AmbulanciaEmpezarBoton.setVisible(false);
    AmbulanciaPararBoton.setVisible(false);
    AmbulanciaVolverBoton.setVisible(false);
    AmbulanciaTallerBoton.setVisible(false);
    MuestraDeExcepcionPacienteLabel.setVisible(false);
    MuestraDeExcepcionMedicoLabel.setVisible(false);
}
```

CONTROLADOR

```
public class Controlador implements ActionListener {  6 usages
    private IVista vista;  108 usages
    private SistemaFacade sistema;  40 usages
    private SimulacionAmbulancia simulacion;  7 usages
    private FactoryMedico factoryMedico;  2 usages
    private FactoryPaciente  factoryPaciente;  2 usages

    public Controlador(IVista vista, SistemaFacade sistema, FactoryMedico factoryMedico, FactoryPaciente factoryPaciente)
    {
        this.vista = vista;
        this.vista.addActionListener(this);
        this.sistema = sistema;
        this.simulacion = new SimulacionAmbulancia( controlador: this, vista);
        this.factoryMedico = factoryMedico;
        this.factoryPaciente = factoryPaciente;
    }
```