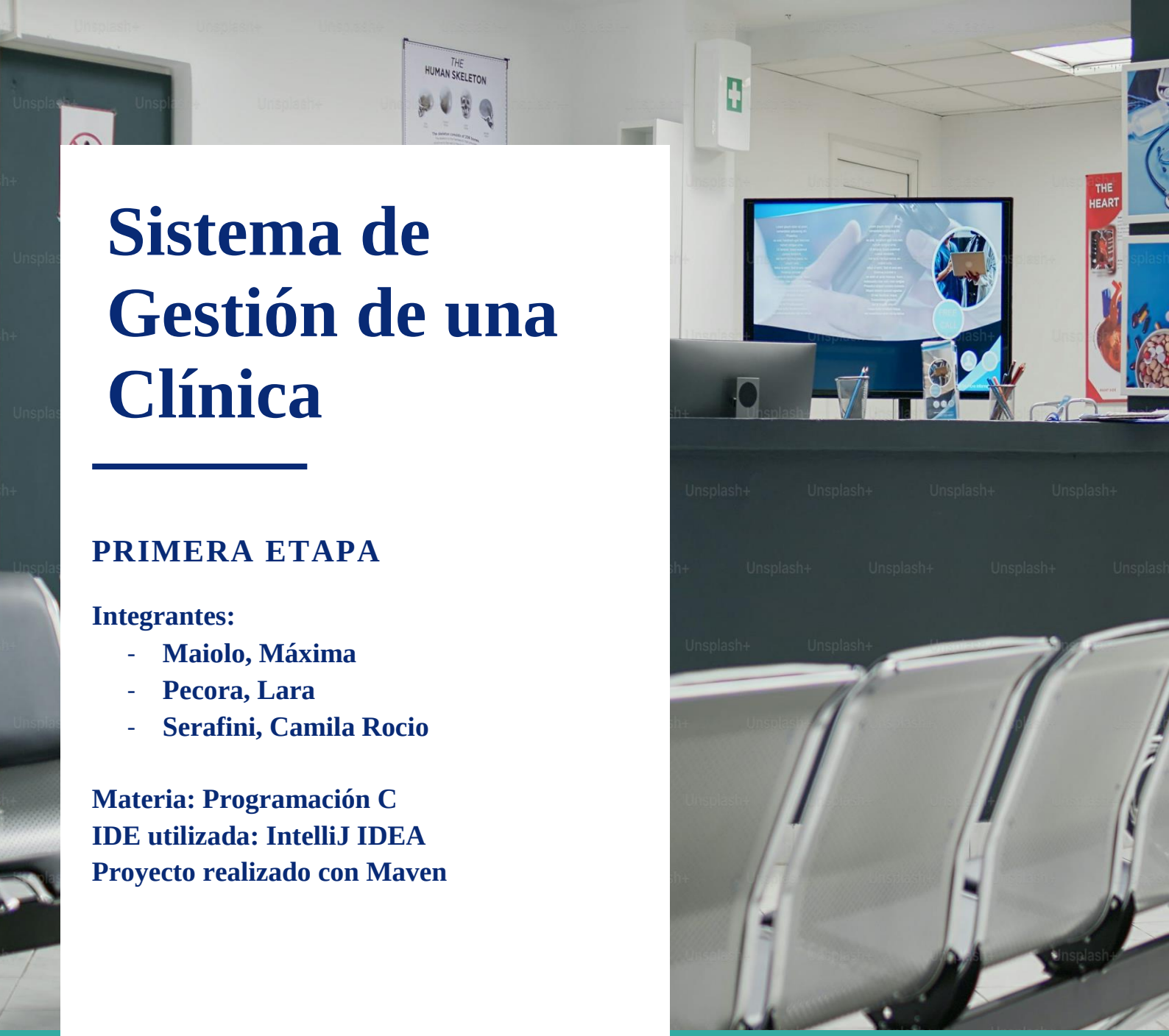


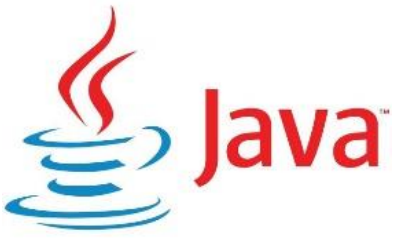
Sistema de Gestión de una Clínica

PRIMERA 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 sistema desarrollado corresponde a la primera etapa de un Sistema de Gestión de una Clínica Privada, que permite manejar información de médicos y pacientes, así como gestionar el ingreso, atención, internación, egreso y facturación de los pacientes.

El objetivo principal fue modelar el dominio de la clínica aplicando los principios de la Programación Orientada a Objetos, junto con patrones de diseño para estructurar el código de manera extensible y modular, los cuales serán explicados más adelante.

El sistema se ejecuta mediante un método **main** sin interfaz gráfica (en esta primera parte), simulando las distintas operaciones de la clínica a través de llamadas a los métodos del sistema.

En el presente informe se incluirá la siguiente información:

- Explicación de la estructura del proyecto.
- Diagrama de clases (UML) correspondiente al sistema desarrollado.
- La documentación generada mediante Javadoc.
- Descripción detallada de los patrones de diseño implementados y su aplicación en el proyecto.

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: sistema**

  Este paquete contiene la base del sistema y las clases principales que gestionan la clínica:

  - Clinica: posee los atributos nombre, direccion, 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.

    Este módulo se apoya en diversas clases del sistema (como Habitacion, RegistroPaciente o 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 la clase principal **Main**, desde la cual se ejecuta el sistema. En esta primera etapa, sin interfaz gráfica, toda la interacción se realiza desde aquí.

- **Paquete: personas**

  Inlcuye las clases que representan a los individuos dentro de la clínica y la clase Domicilio:

  - Persona que será extendida desde Paciente y Medico (tambien 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).

    Entre sus métodos abstractos se incluyen esReemplazado(), reemplazaANinio(), reemplazaAJoven() y reemplazaAMayor(), que determinan las condiciones bajo las cuales un nuevo paciente puede ocupar el lugar de otro en la sala de espera privada, de acuerdo con la lógica de prioridad establecida en el sistema.

    - ✓ 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.

    Esto implementa el patrón Double Dispatch que será explicado luego junto con los demás patrones.

- 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.

  El criterio de reemplazo se basa en las reglas establecidas por el sistema, comentadas anteriormente:
  - Un paciente mayor puede reemplazar a un niño en la sala de espera privada.
  - No puede reemplazar a un joven ni a otro mayor.

  Esta lógica se implementa mediante los métodos esReemplazado(), reemplazaANinio(), reemplazaAJoven() y reemplazaAMayor(), que devuelven valores booleanos según las condiciones de prioridad definidas.

  En conjunto con las clases Joven y Ninio, Mayor permite modelar el comportamiento diferenciado de los pacientes según su grupo etario, lo cual es esencial para la organización del sistema de atención y manejo de la sala de espera.
- Ninio
- Joven

  Estas últimas dos son iguales que la clase Mayor pero con sus propias reglas.

- Domicilio: esta clase es para instanciar a las direcciones, ya sea de la clínica, de cada paciente o de cada médico.

- **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: 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: interfaces**
  - IMedico: componente a ser envuelto, utilizado para el patrón Decorator.

- **Paquete: facturacion**

  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.

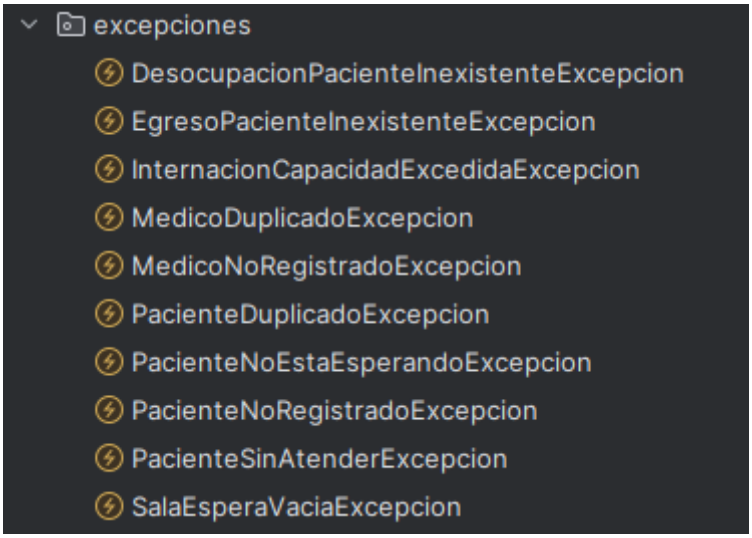
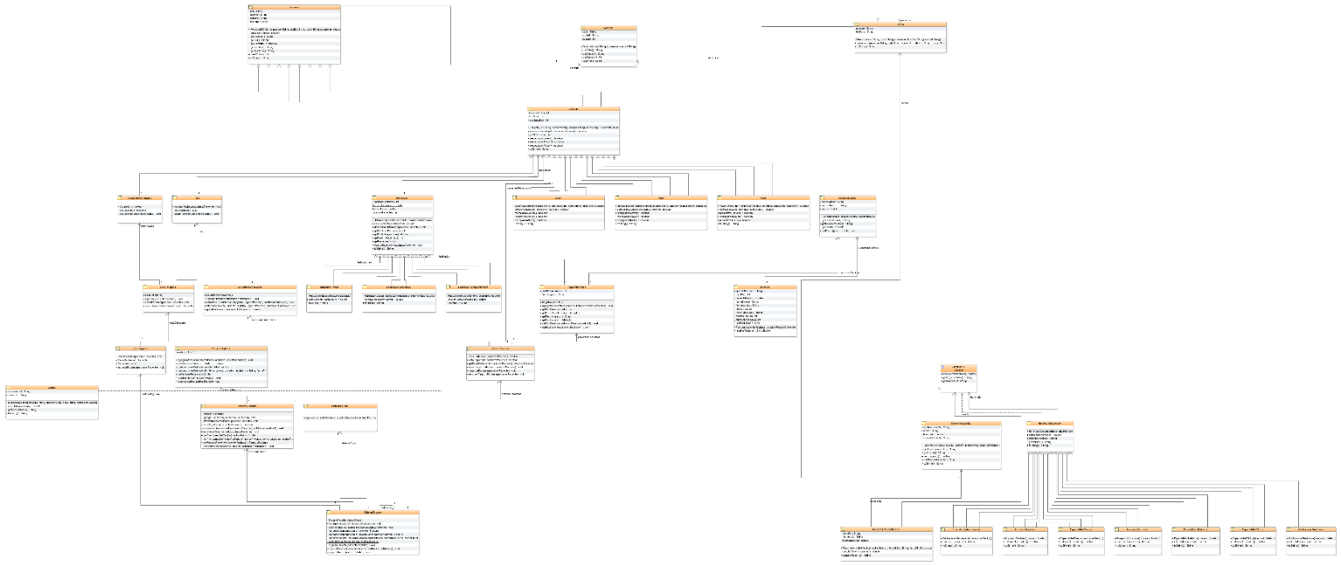


Diagrama de clases (UML)

Diagrama completo del proyecto:



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

Documentacion 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. En general, incluyen:

- Breve descripción, ya sea de métodos o clases.
- Pre y post condiciones.
- @param para los parámetros de los métodos.
- @throws para las excepciones que son lanzadas.
- @return para aclarar que es lo que se retorna en los métodos que tienen retorno.

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:

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. Sin embargo, desde el punto de vista del cliente (actualmente la clase main), todas las operaciones se realizan a travez de una unica interfaz que vendría a ser la fachada.

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.

De esta forma, permite que el cliente invoque el método mediante una única llamada simple, por ejemplo para registrar un paciente o médico desde la clase principal:

```
// Registro
sistema.registraMedico(medClinica);
sistema.registraMedico(medCirugia);
sistema.registraMedico(medPediatra);
sistema.registraPaciente(p1);
sistema.registraPaciente(p2);
sistema.registraPaciente(p3);
sistema.registraPaciente(p4);
```

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 utilizada 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 debemos ver si reemplaza al existente en la sala privada o si va al patio. Entonces ahora le pide al nuevo paciente que resuelva esto, por ejemplo: si el existe es mayor, estamos en la clase Mayor que tenemos:

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

El método reemplazaAMayor() se ejecuta en el tipo de otroPaciente, entonces si es otro mayor o un niño, no se reemplaza. 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 anteriormenete, el objetivo es garantizar que exista una única instancia de esta la clase Clinica y SistemaFacade. La clínica representa el contexto principal del sistema, conteniendo la información general de la institución y actuando como entidad central que coordina los distintos módulos. Y el SistemaFacade es el que centraliza la comunicación entre los ModulosIngreso, ModuloAtencion y ModuloEgreso.

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;
}
```

```
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;
}
```

De esta manera la clase controla completamente la creación del objeto, asegurando que siempre se opere sobre la misma referencia. El uso del patrón se evidencia en el método principal de la siguiente forma:

```
Clinica clinica = Clinica.getInstancia( nombre: "Clinica Central",
        calle: "Avenida San Martin", numero: 105, telefono: "+54 11 1234-5678", ciudad: "Buenos Aires");
SistemaFacade sistema = SistemaFacade.getInstancia(clinica);
```

Entonces cualquier intento posterior para obtener una instancia de la clínica 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 varientes 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 muestra un ejemplo de medico y uno de paciente:

```
// Médicos
IMedico medClinica = factoryMedico.crearMedico( DNI: "10000000", nombre: "Ana", apellido: "Garcia",
        calle: "Calle 1", numero: 1, ciudad: "CABA", telefono: "111-1111", matricula: "M-123",
        especialidad: "CLINICA", contratacion: "PERMANENTE", posgrado: "MASTER");

// 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");
```

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

➢ Patrón Decorator

Cuarto y último patrón utilizado para esta primer etapa del proyecto. 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 sobreescribe el método calcularHonorarios() dandole el valor base.

En segundo lugar, tenemos la clase abstracta HonorarioDecorator que implenta 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 sobreescribe el método calcularHonorarios() aplicando el incremento correspondiente según su tipo, los cuales serán acumulativos con respecto a cada característica (especialidad, contraracion y posgrado).