Programación III

Informe Trabajo Práctico Grupal (TPG)

“Gestión de una Clínica”

Integrantes:

- Cazorla Martínez, Nicolás
- Dell Aguila Ureña, Franco
- Maletta, Augusto
- Rodríguez, Juan Gabriel

Universidad Nacional de

Mar del Plata

Facultad de Ingeniería.

Informe Trabajo Práctico Grupal (TPG)
“Gestión de una clínica”

Primera Parte

Flujo completo desde el ingreso hasta la salida de un paciente

A continuación se detalla brevemente cual es el flujo esperado y cuáles son los métodos que interactúan al ingresar un paciente a la clínica hasta el momento de retirarse.

Ingreso del paciente

Se ingresa un paciente a la clínica a partir del método ingresar paciente de la clínica

```
public void ingresarPaciente(String nombre, String apellido, int DNI, String rangoEtario)
```

A partir de este método, se resuelve el conflicto entre la sala de espera y el patio, para determinar donde se ubica el paciente.
También se le asigna al paciente un numero de orden (auto-incremental a medida que llegan nuevos pacientes) que será el que determine el orden de atención de los pacientes. El primer paciente en llegar será el primero en ser atendido.

Atención del paciente

En un momento determinado, se llama al siguiente paciente en espera a ser atendido, esto se resuelve a partir de un método del módulo de atención.

```
public void retiraPaciente() throws EsperaVaciaException {
```

El método quita al paciente de la sala de espera y lo pone en una lista de pacientes en atención, en este momento los médicos pueden realizar consultas a un paciente dado te dado o internar al paciente en una o varias habitaciones hasta que se retire de la clíniernar al paciente en una o varias habitaciones hasta que se retire de la clínica.

Facturación y egreso del paciente

Se selecciona un paciente de la Lista de Atención tomando como clave su dni
Se le confecciona una factura al paciente con la información pertinente
Se trabaja con un hashmap para llevar el conteo de las consultas realizadas por cada médico. También se tiene en cuenta sumarle el honorario correspondiente al médico por la consulta.
Se utiliza un iterator para recorrer todas las consultas efectuadas por el paciente en la última visita, así también como para conocer los servicios utilizados en las mismas
Se muestran por pantalla los datos correspondientes a la factura en forma de tabla
Se le da el alta al paciente (se lo retira de la Lista de atención)

```
public void creaFacturapaciente(Integer dni){
```

Reporte de Actividad Médica

Se realiza un reporte de la actividad de un médico por día en un periodo de varios días presentando el id del mismo.
Para ello se recorre un hashmap el cual va recopilando las consultas de cada médico y los pacientes que fueron atendidos en dichas consultas.

Patrones aplicados

En la primera parte del trabajo se utilizaron los siguientes patrones de diseño:

- Decorator
- Double Dispatch
- Singleton
- Factory

El patrón Decorator fue utilizado en la creación de médicos y habitaciones, decorando según el tipo correspondiente

El patrón Double Dispatch fue utilizado para resolver los conflictos entre la sala de espera y el patio, resolviendo quien tenía la prioridad.

El patrón Singleton se utiliza para la clínica, que siempre tendrá una única instancia que contiene colecciones de pacientes, habitaciones, médicos,etc

El patrón Factory se utiliza para la creación de instancias de médicos, habitaciones y pacientes, ya que se debe tener en cuenta su especialidad o rango etario para el caso de los pacientes.

Javadoc

El Javadoc completo se encuentra en los archivos entregados, mediante index.html se logrará navegar por las documentaciones correspondientes de la siguiente manera:

Packages

Package
PClinica
PException
PHabitaciones
PModulos
PPersona.PMedico
PPersona.PPacientes

Package PClinica

Class Summary

Class	Description
ClinicaSingleton	Esta clase contiene en diferentes colecciones informacion (mÃ©dicos, pacientes atendidos, habitaciones) **PatrÃ³n aplicado:** Singleton

Package PException

Exception Summary

Exception	Description
EsperaVaciaException	Lanzada cuando no hay pacientes para atender
FactoryHabitacionException	Lanzada cuando no se pudo crear la habitacion solicitada
FactoryMedicoException	Lanzada cuando no se pudo crear el medico solicitado
IngresoPacienteHabitacionException	Lanzada cuando no se pudo ingresar el paciente a la habitacion

Package PHabitaciones

Interface Summary

Interface	Description
IHabitacion	Interfaz que declara los metodos a implementar por las habitaciones

Class Summary

Class	Description
DHabitacionCompartida	Clase que representa una habitacion compartida por los pacientes
DHabitacionPrivada	Clase que representa una habitacion privada
DTerapiaIntensiva	Clase que representa una habitacion destinada a terapia intensiva
Habitacion	Clase que representa las habitaciones donde se alojan los pacientes
HabitacionDecorator	Clase destinada a implementar el Patron Decorator aplicado a las habitaciones **Patron aplicado:** Decorator
HabitacionFactory	Esta clase tiene la responsabilidad de generar habitaciones **Patrón aplicado:** Factory

Package PModulos

Class Summary

Class	Description
ModuloAtencion	Modulo que se encarga de atender paciente retirandolo del lugar de espera y ubicandolo en atencion
ModuloFacturacion	Clase dedicada a tomar los datos de un paciente de la clinica para generar su factura correspondiente Retira de la lista de atencion a los pacientes atendidos
ModuloIngreso	Este modulo se encarga de ingresar al paciente, sus responsabilidades son: Asignar numero de orden al paciente Resolver conflictos en la sala de espera privada y el patio Cargar al paciente en los registros de la clinica
TEST	Clase destinada a hacer pruebas con los modulos implementados

Package PPersona.PMedico

Interface Summary

Interface	Description
IMedico	Interface que contiene los metodos a implementar por los medicos

Class Summary

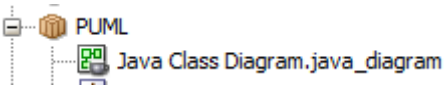
Class	Description
Cirugia	Clase que representa un cirujano
Clinico	Clase que representa a un medico clinico
Consulta	Clase que almacena la fecha y el codigo que pueden pertenecer tanto a un paciente, como a un medico
DDoctor	Clase que representa un doctor
DMagister	Clase que representa un Magister
DoctorDecorator	Clase padre destinada a aplicar el Patron Decorator a los distintos medicos
DPermanente	Clase que representa un permanente
DResidente	Clase que representa un residente
Medico	Esta es la "Master Class" que controla a todos los medicos con sus respectivos decorados.
MedicoFactory	Clase que aplica el patrón Factory para la creación de nuevas instancias de médicos **Patrón aplicado:** Factory
Muestra	Clase que tiene la responsabilidad de mostrar el reporte de actividad de un medico
Pediatria	Clase que representa un pediatra

Package PPersona.PPacientes

Class Summary

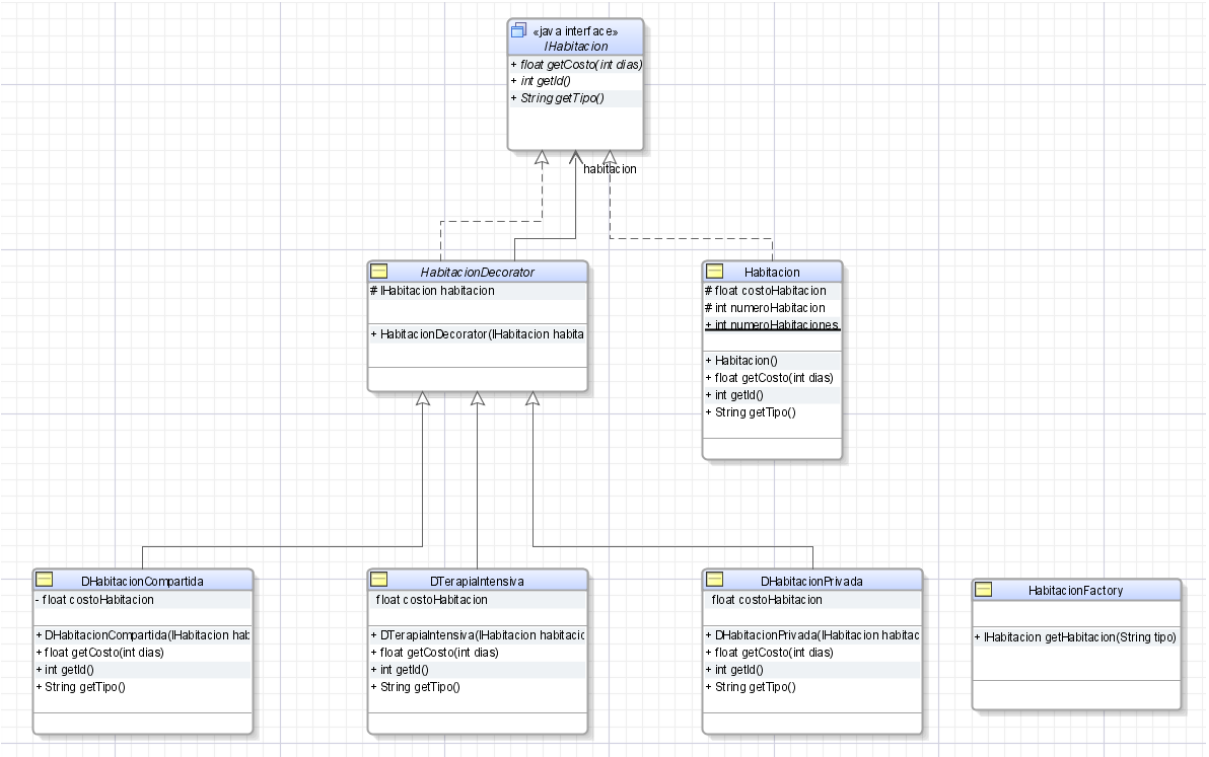
Class	Description
Joven	Esta clase hereda de la clase abstract Paciente, su implementacion corresponde a uno de los grupos etarios **Patrón aplicado:** Double Dispatch.
Mayor	Esta clase hereda de la clase abstract Paciente, su implementacion corresponde a uno de los grupos etarios **Patrón aplicado:** Double Dispatch.
Nino	Esta clase hereda de la clase abstract Paciente, su implementacion corresponde a uno de los grupos etarios **Patrón aplicado:** Double Dispatch.
Paciente	Clase que representa un paciente
PacienteFactory	Clase encargada de de generar pacientes utilizando patrón Factory

Diagrama de clases (UML)

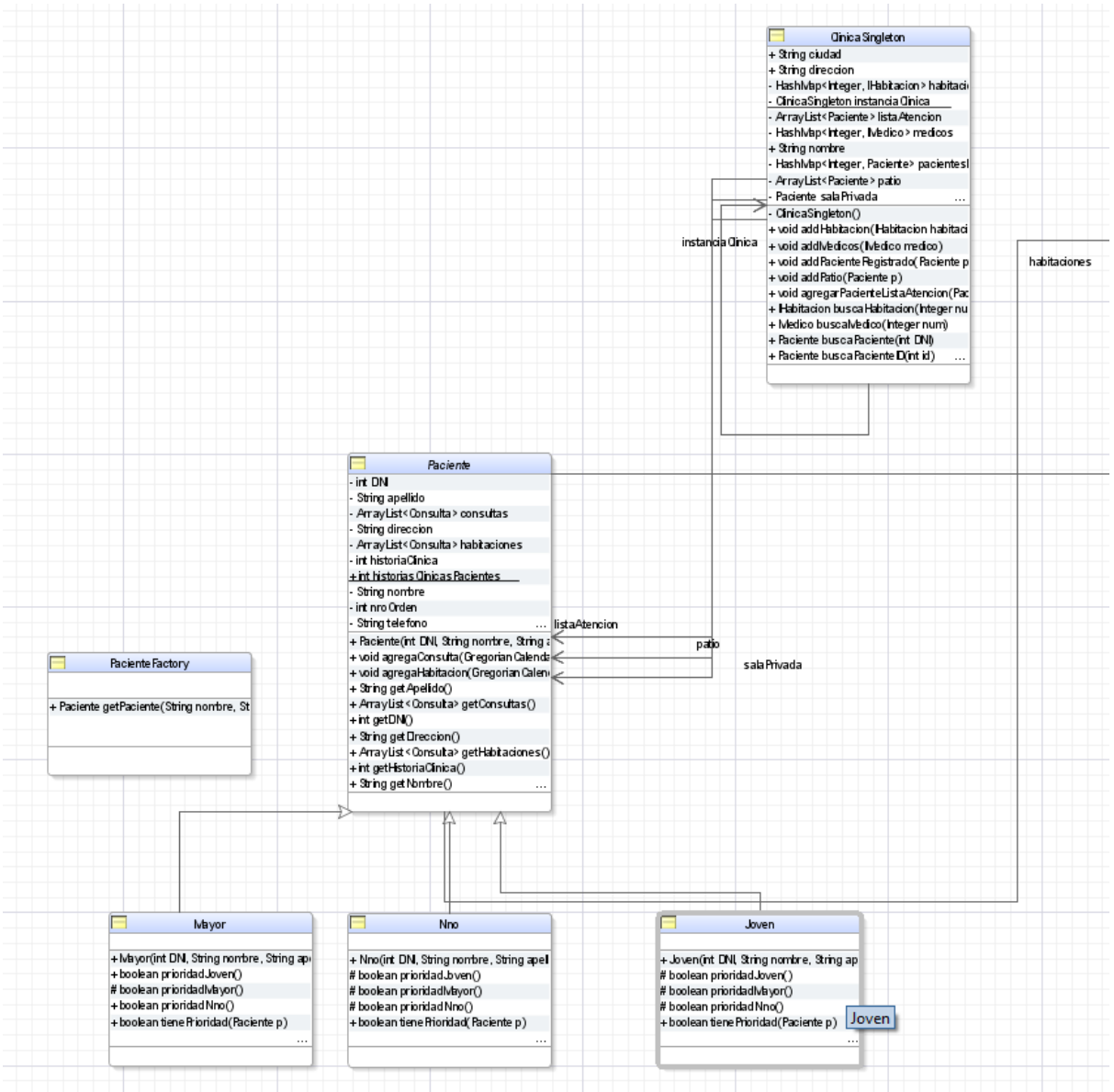
El diagrama de clases puede apreciarse en su totalidad en el trabajo entregado,dentro del paquete PUML 

Sin embargo, a fines de complementar el informe, se presentan a continuación los diagramas de habitacion,paciente y medico respectivamente:

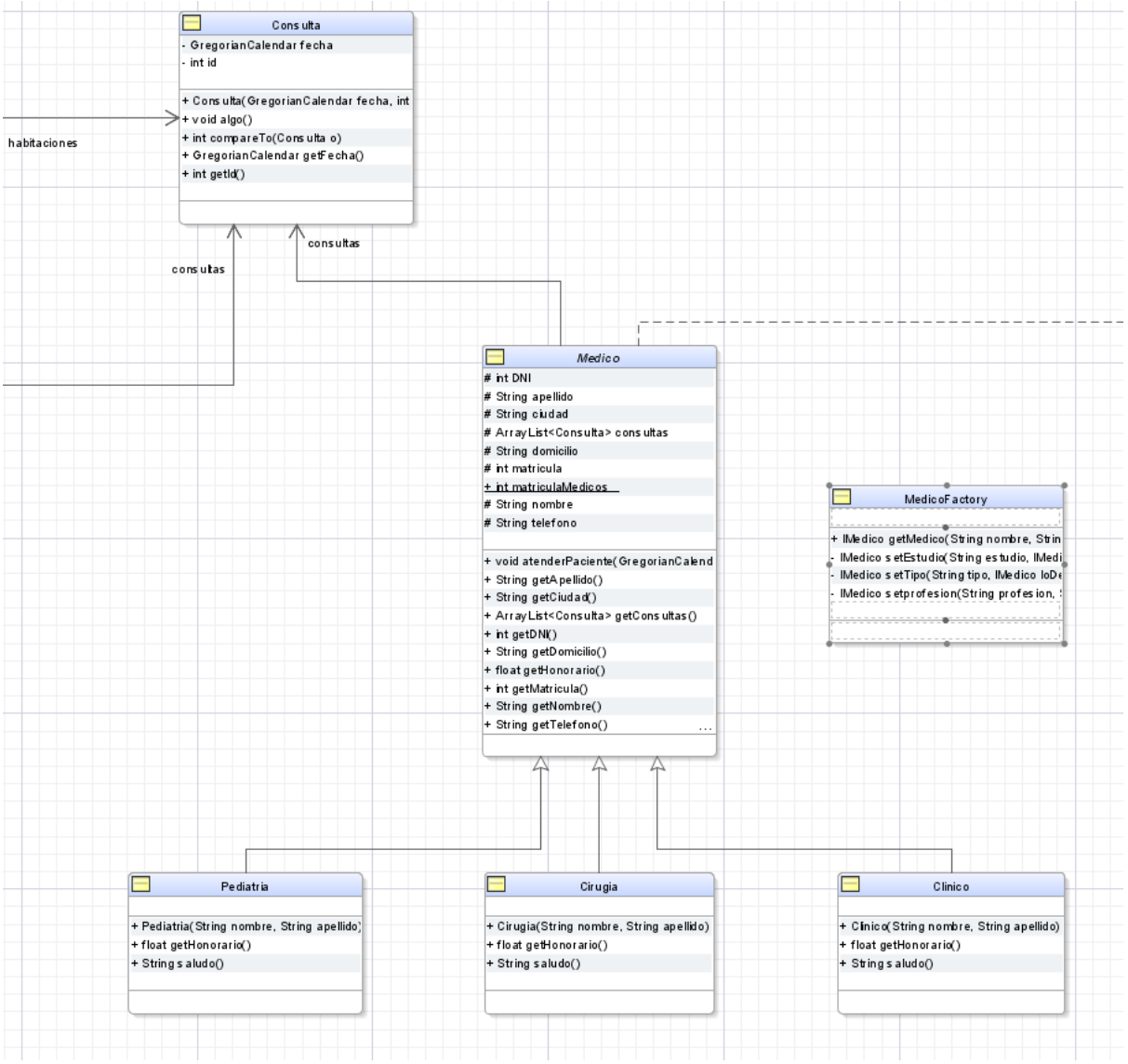
Habitaciones:



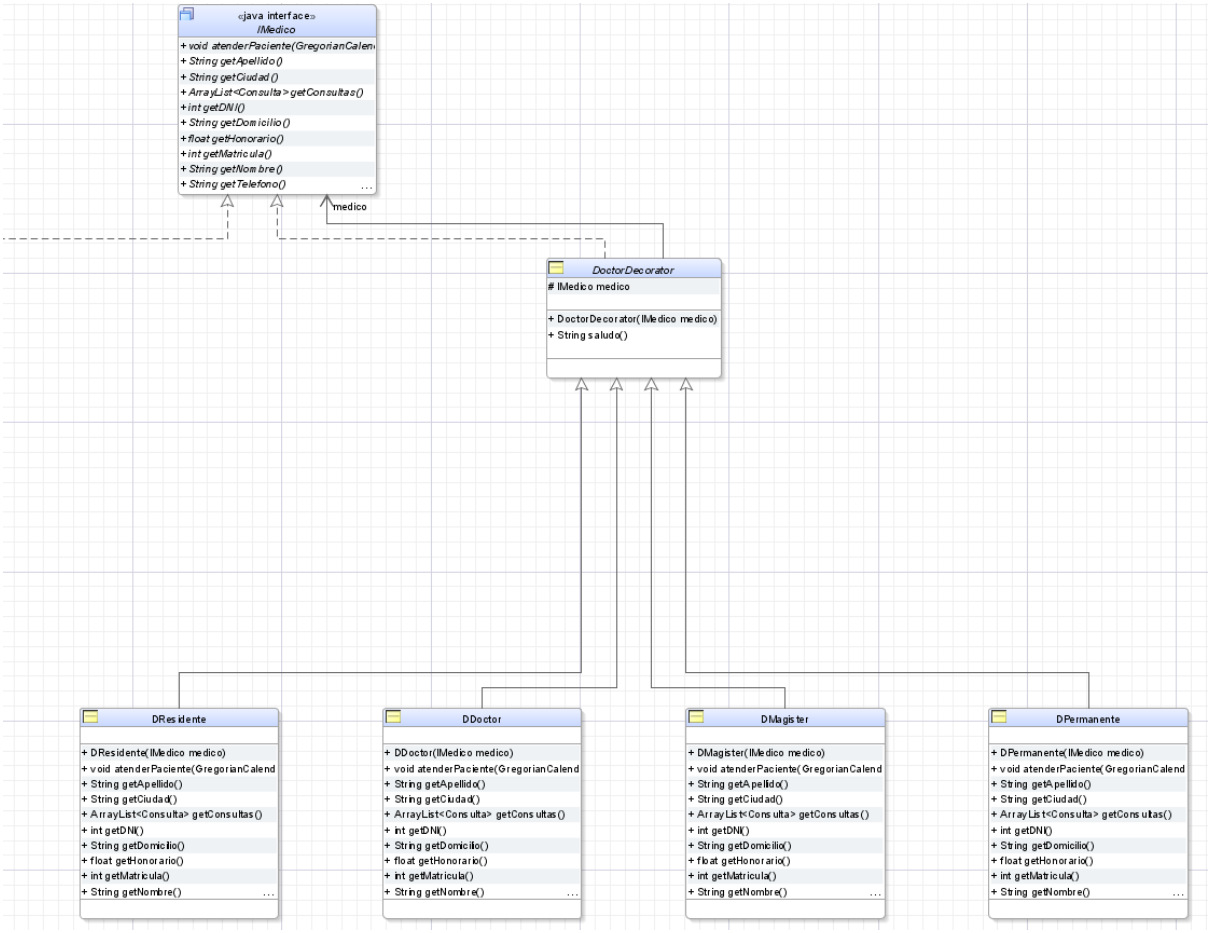
Pacientes:


Clinica Singleton
+ String ciudad
+ String direccion
- HashMap<Integer, IHabitacion> habitaci
- ClinicaSingleton instancia Clinica
- ArrayList<Paciente> listaAtencion
- HashMap<Integer, IMedico> medicos
+ String nombre
- HashMap<Integer, Paciente> pacientesI
- ArrayList<Paciente> patio
- Paciente salaPrivada
- ClinicaSingleton()
+ void addHabitacion(IHabitacion habitaci
+ void addIMedicos(IMedico medico)
+ void addPaciente Registrado(Paciente p
+ void addPatio(Paciente p)
+ void agregarPacienteListaAtencion(Pac
+ IHabitacion buscaHabitacion(Integer nu
+ IMedico buscaIMedico(Integer num)
+ Paciente buscaPaciente(int DNI)
+ Paciente buscaPaciente ID(int id)
instancia Clinica
habitaciones
Paciente
- int DNI
- String apellido
- ArrayList<Consulta> consultas
- String direccion
- ArrayList<Consulta> habitaciones
- int historiaClinica
+ int historias Clinicas Pacientes
- String nombre
- int nro Orden
- String telefono
+ Paciente(int DNI, String nombre, String
+ void agregaConsulta(Gregorian Calend
+ void agregaHabitacion(Gregorian Calen
+ String getApellido()
+ ArrayList<Consulta> getConsultas()
+ int getDNI()
+ String getDireccion()
+ ArrayList<Consulta> getHabitaciones()
+ int getHistoriaClinica()
+ String getNombre()
listaAtencion
patio
salaPrivada
Paciente Factory
+ Paciente getPaciente(String nombre, St
IMayor
+ IMayor(int DNI, String nombre, String ap
+ boolean prioridadJoven()
# boolean prioridadIMayor()
+ boolean prioridadNno()
+ boolean tienePrioridad(Paciente p)
Nno
+ Nno(int DNI, String nombre, String apell
# boolean prioridadJoven()
# boolean prioridadIMayor()
# boolean prioridadNno()
+ boolean tienePrioridad(Paciente p)
Joven
+ Joven(int DNI, String nombre, String ap
# boolean prioridadJoven()
# boolean prioridadIMayor()
# boolean prioridadNno()
+ boolean tienePrioridad(Paciente p)
Joven

Médicos: (1ra parte)



Medicos (2da parte):



Observaciones

- El plantel médico, los pacientes ingresados y las habitaciones se encuentran en colecciones de la clase Clínica, esto facilita el ingreso a los datos ya que la instancia de la clínica es única y se pueden solicitar todos los datos a misma.

- Se consideran las siguientes excepciones:
  - **EsperaVaciaException:** El módulo de atención llama al siguiente paciente a ser atendido, pero no hay ninguno.
  - **FactoryHabitacionException**: El rango etario del paciente no es válido, por lo tanto no se pudo crear un paciente nuevo.
  - **FactoryMedicoException:** Igual que la excepción anterior, pero en este caso la especialidad del médico no es válida.
  - **IngresoPacienteHabitacionException:** Se lanza cuando no es posible ingresar a un paciente a una habitacion dada.
  - **NoHayPacienteException:** Lanzada cuando no se encontro paciente atendido por un medico en un tiempo determinado

Segunda Parte

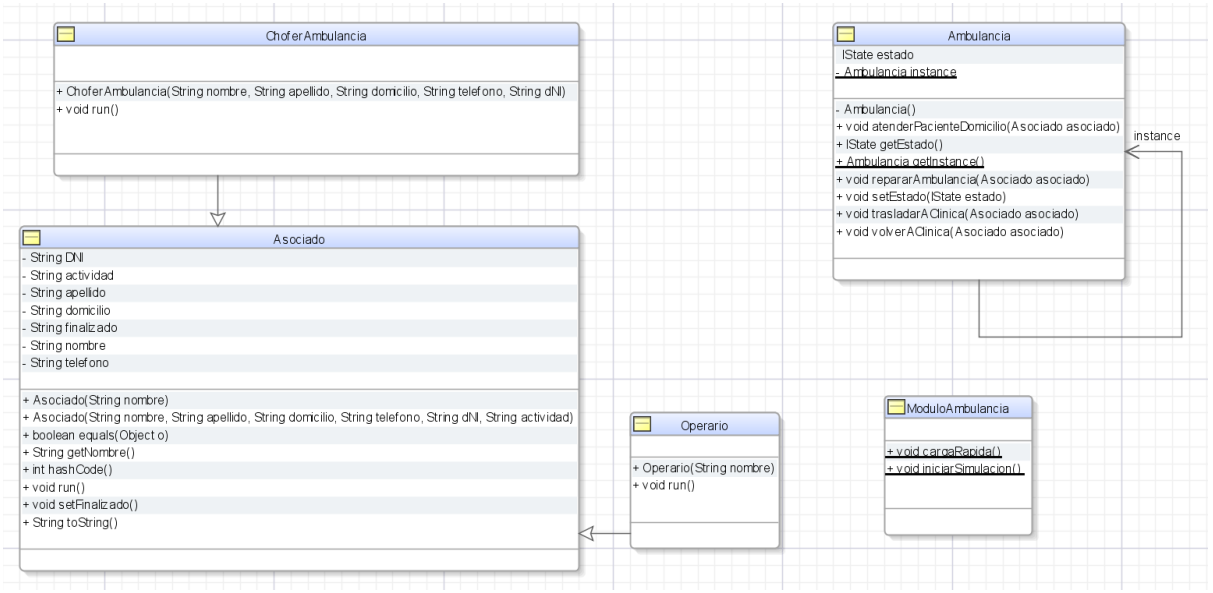
Introducción

Para la segunda parte de este trabajo, se incorpora un nuevo módulo, la ambulancia.

Los Asociados a la Clínica podrán solicitar la ambulancia. Una persona se asocia a la clínica solicitándolo, se requiere conocer dni, nombre y apellido, domicilio, teléfono.

Los Asociados podrán solicitar una ambulancia aclarando si es para traslado o para atención a domicilio.

Además el Operario de la Clínica podrá solicitar la reparación de la Ambulancia.
Esta simulación de uso pretende resolver problemas de concurrencia en Java.
Por otro lado, debe persistir la información de la clínica (pacientes, habitaciones, médicos, etc).



Cambios del modelo de la primera parte

1. Ahora los pacientes y los médicos heredan de una clase padre “Persona” que contiene atributos básicos como nombre,apellido, ciudad, DNI, entre otros.
2. Se modificó una búsqueda con for en la clase clínica, ahora se hace con un while().
3. El módulo de facturación ya no imprime por pantalla los datos de la factura, ahora genera un documento PDF que contiene los cargos y el importe a pagar por el cliente.
4. Ahora el método getCosto(int dias) de la clase habitacion utiliza el valor días

Patrones aplicados

Para la segunda parte del trabajo, se aplicaron los siguientes patrones de diseño para resolver los diferentes problemas que aparecieron:

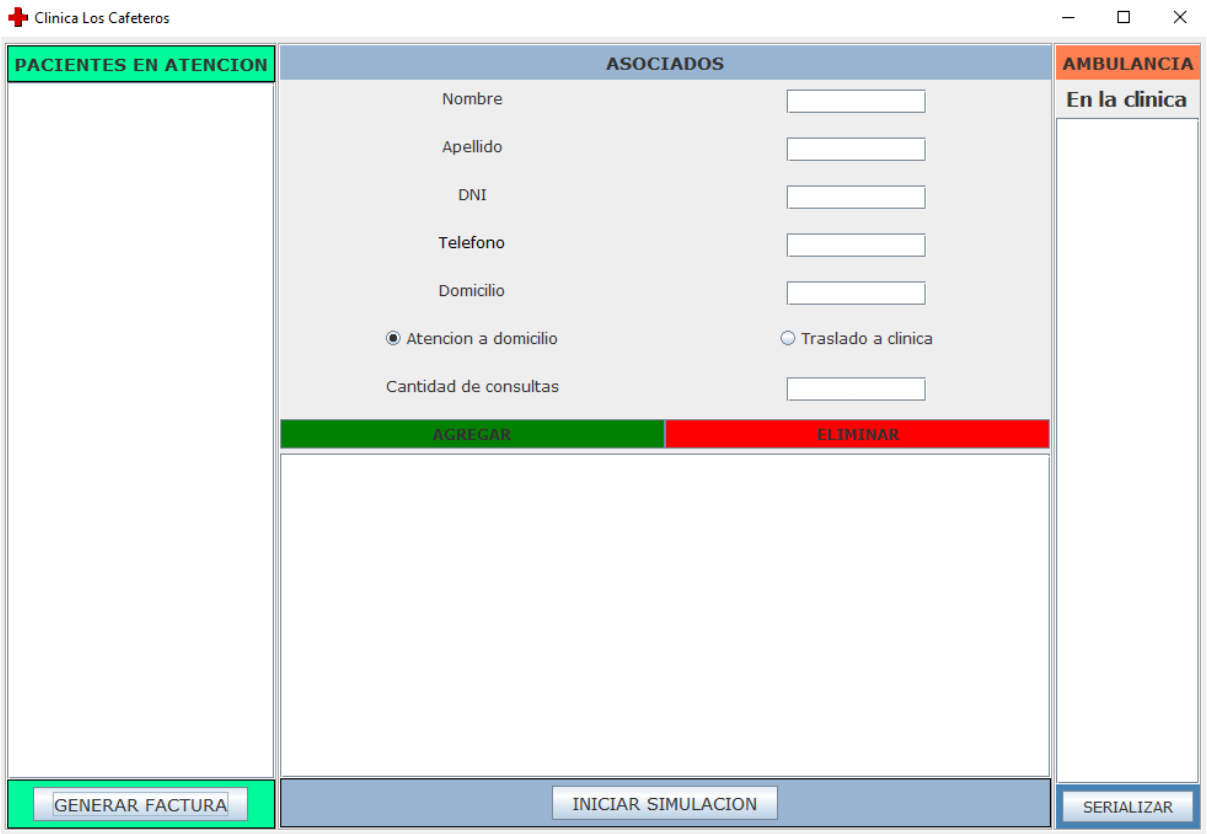
- Patrón State
- Patrón DAO/DTO
- Patrón MVC

El **patrón State** se utilizó para modelar el comportamiento de la ambulancia, las solicitudes de los asociados pueden cumplirse o no dependiendo del estado actual de la ambulancia.

El patrón **DAO/DTO** fue necesario para persistir atributos de la clínica, que aplica patrón Singleton.

El patrón **MVC** nos ayudó a trabajar de forma más prolija, separando responsabilidades del modelo, la vista y el controlador.

Interfaz gráfica



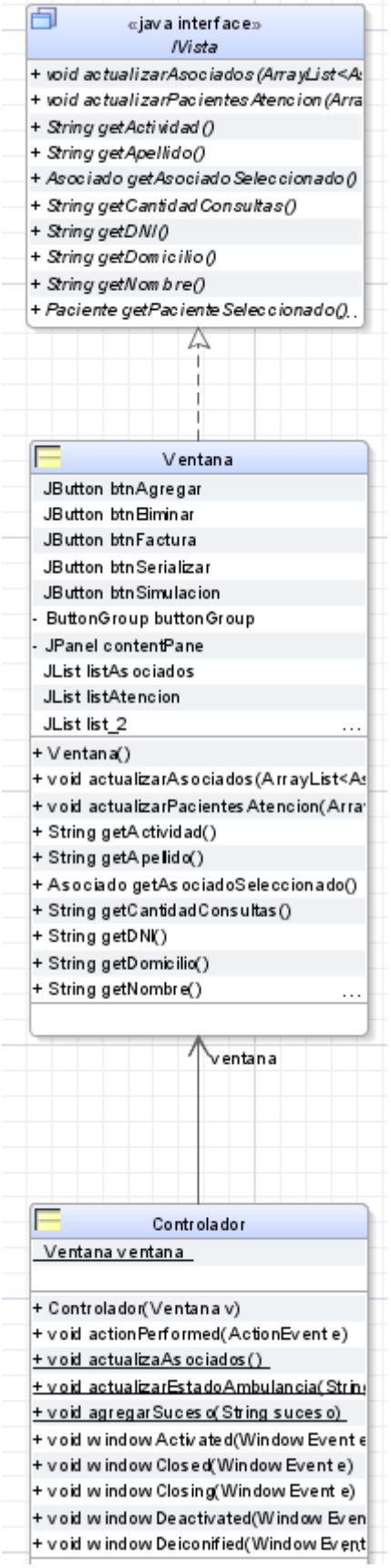
A rasgos generales, la ventana se divide en 3 secciones principales:

- **Parte izquierda:** Contiene una lista de los pacientes en atención, de los cuales se puede generar la factura correspondiente (Presionando el botón “GENERAR FACTURA”)

- **Parte central:** Permite la creación de nuevos asociados de la clínica, se piden los datos de contacto de la persona, el tipo de solicitud que va a hacer y la cantidad de veces que la hará.
  A medida que se agregan asociados, se visualizan en una lista debajo de los botones AGREGAR y ELIMINAR

- **Parte derecha:** Registra todos los cambios y pedidos que recibe la ambulancia, va listando en un área de texto.

En la parte inferior se agregó un botón “SERIALIZAR” que permite al usuario persistir los datos a demanda


«java interface»
IVista
+ void actualizarAsociados(ArrayList<A
+ void actualizarPacientesAtencion(Arra
+ String getActividad()
+ String getApellido()
+ Asociado getAsociadoSeleccionado()
+ String getCantidadConsultas()
+ String getDNI()
+ String getDomicilio()
+ String getNombre()
+ Paciente getPacienteSeleccionado()..
Ventana
JButton btnAgregar
JButton btnBiminar
JButton btnFactura
JButton btnSerializar
JButton btnSimulacion
- ButtonGroup buttonGroup
- JPanel contentPane
JList listAsociados
JList listAtencion
JList list_2 ...
+ Ventana()
+ void actualizarAsociados(ArrayList<As
+ void actualizarPacientesAtencion(Arra
+ String getActividad()
+ String getApellido()
+ Asociado getAsociadoSeleccionado()
+ String getCantidadConsultas()
+ String getDNI()
+ String getDomicilio()
+ String getNombre() ...
ventana
Controlador
Ventana ventana
+ Controlador(Ventana v)
+ void actionPerformed(ActionEvent e)
+ void actualizaAsociados()
+ void actualizarEstadoAmbulancia(Strin
+ void agregarSuceso(String suceso)
+ void windowActivated(WindowEvent e
+ void windowClosed(WindowEvent e)
+ void windowClosing(WindowEvent e)
+ void windowDeactivated(WindowEven
+ void windowDeiconified(WindowEvent

Asociados de la clínica

La clase **Asociado** es la que se encarga de la lógica de los clientes de la clínica.
Está hereda de **Thread** de tal manera que se tenga un hilo por asociado, también implementa la interfaz **Serializable** para poder guardar la información.

Además de los asociados en nuestra simulación existe un **Operario** (que por practicidad hereda de asociado) que pide que sea llevada al taller.

Esto nos llevó a un gran inconveniente: ¿Como hacemos para que la ambulancia al momento de quedarse sin tareas esté disponible en la clínica? Nuestra solución fue la de implementar una clase llamada **ChoferAmbulancia** (que hereda de asociado también) que se encargue de pedir que la ambulancia regrese a la clínica en intervalos aleatorios de tiempo.

Modulo Facturacion y librerías externas

Al momento de decidirnos a crear facturas por PDF nos encontramos que Java no tiene ninguna librería por defecto que lo haga, por lo tanto decidimos buscar por nuestra cuenta.

Itextpdf es un librería que soporta Maven muy utilizada para crear PDF en Java, en nuestro caso usamos la versión 5.5.13.2 la cual a pesar de no ser la más nueva es más que suficiente para lo que buscábamos crear

Esta librería crea los PDF en base a una ruta en el FileSystem y un objeto de clase “Document”.
A este documento se le van agregando párrafos (Clase Paragraph), Tablas(Clase PdfTable) para poder agregar información de las veces que atendieron al paciente y los días internados con sus respectivos gastos.

Módulo de Ambulancia

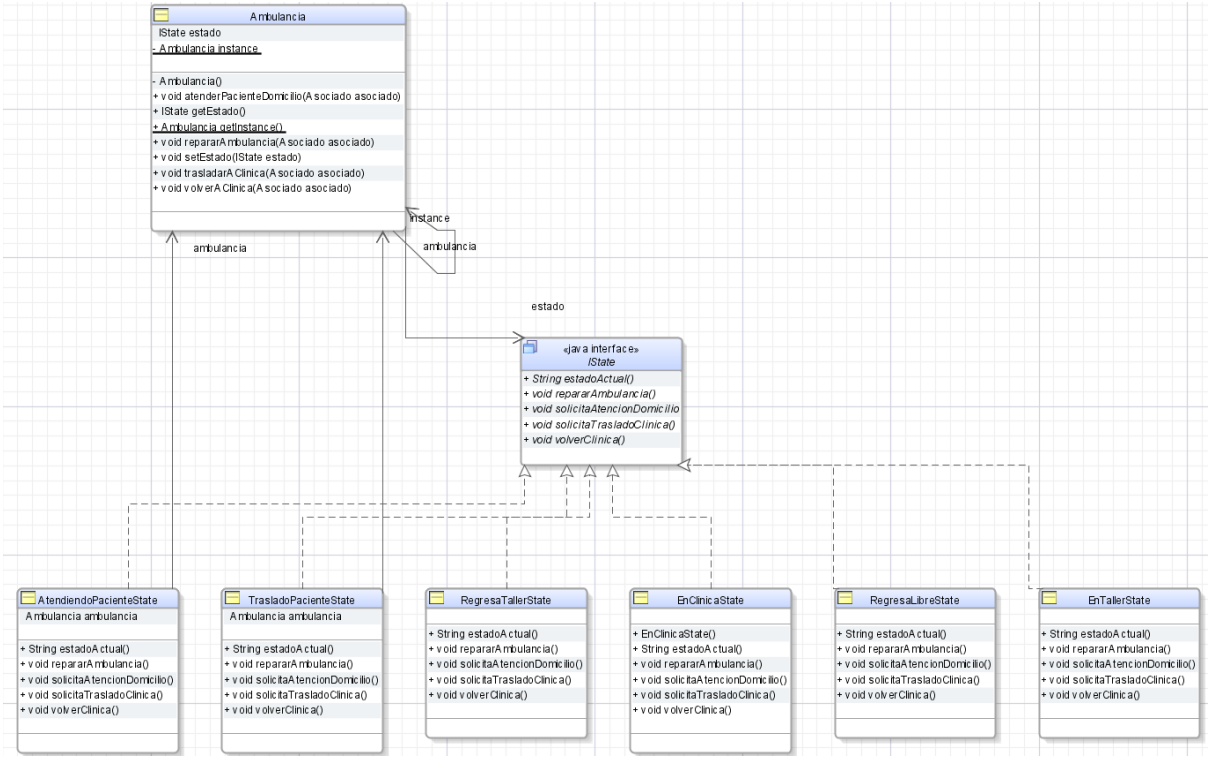
A continuación se adjunta la tabla modificada con todas las posibles solicitudes y estados de la ambulancia

	Paciente solicita atención a domicilio	Paciente solicita traslado a la clínica	Es tiempo de volver a la clínica	Operario solicita reparar la ambulancia
1) Disponible en clínica	Pasa a 3	Pasa a 2	No hay cambio	Pasa a 5
2) Traslado Paciente a la clínica	No hay cambio	No hay cambio	Pasa a 1	No hay cambio
3) Atención Paciente en domicilio	No hay cambio	No hay cambio	Pasa a 4	No hay cambio
4) Regresando de una atención sin paciente	Pasa a 3	Pasa a 2	Pasa a 1	Pasa a 1
5) En Taller	No hay cambio	No hay cambio	Pasa a 6	No hay cambio
6) Regresando del Taller	No hay cambio	No hay cambio	Pasa a 1	No hay cambio

Verde: Interpretaciones distintas a la tabla de enunciado
Amarillo: Cambios de estado de la ambulancia

Para poder determinar los estados de la ambulancia implementamos el patrón **STATE** el cual se encarga de permitir cambiar los estados de la ambulancia (el patrón por sí solo no lo hace, necesita que alguien le pida cambiar) para esto implementamos la interface **IState** que se encarga de definir las peticiones que le podemos hacer a la ambulancia (Cantidad de columnas en la matriz de arriba).

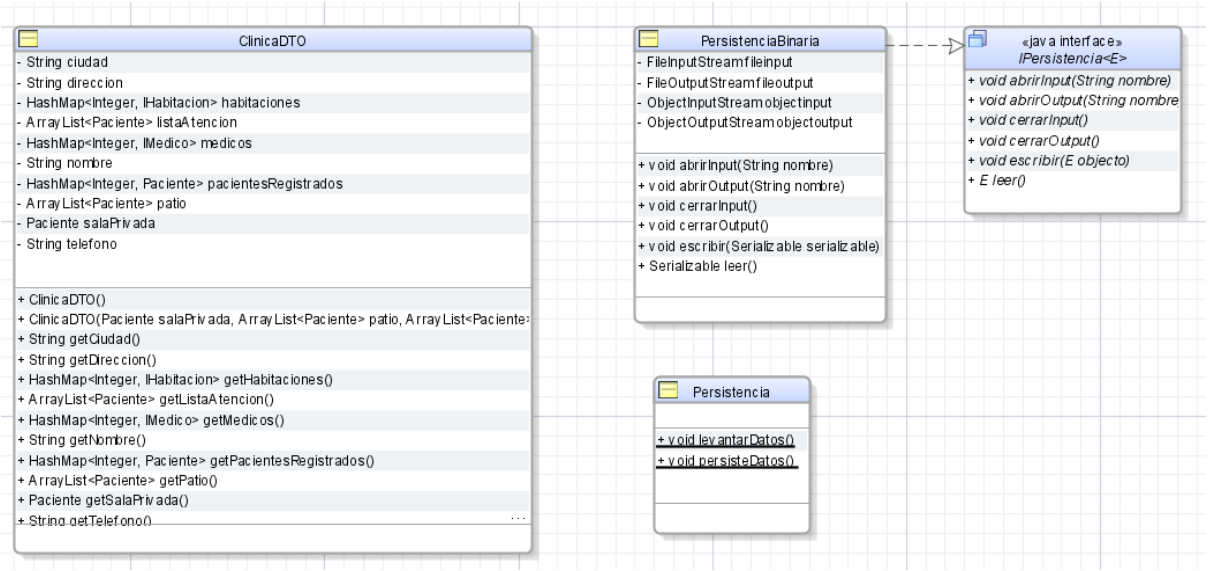
Luego creamos 6 clases que heredan de **IState** que representan todos los estados posibles de la ambulancia (Cantidad de filas de la matriz), en ellas se encuentra toda la lógica de cambio de estado de la ambulancia.



Serialización

Para poder persistir los datos de la aplicación decidimos implementar una serialización binaria ya que no requiere que modifiquemos la estructura de las clases del proyecto como con el **XML**, aun así nos enfrentamos a otro gran problema que fue persistir una clase **ClinicaSingleton**.

Para esto implementamos un **DTO (Data Transfer Object)** al que llamamos **“ClinicaDTO”** que se encargó de “hacer de pasamano” entre la **ClinicaSingleton** y el archivo.


ClinicaDTO
- String ciudad
- String direccion
- HashMap<Integer, IHabitacion> habitaciones
- ArrayList<Paciente> listaAtencion
- HashMap<Integer, IMedico> medicos
- String nombre
- HashMap<Integer, Paciente> pacientesRegistrados
- ArrayList<Paciente> patio
- Paciente salaPrivada
- String telefono
+ ClinicaDTO()
+ ClinicaDTO(Paciente salaPrivada, ArrayList<Paciente> patio, ArrayList<Paciente>
+ String getCiudad()
+ String getDireccion()
+ HashMap<Integer, IHabitacion> getHabitaciones()
+ ArrayList<Paciente> getListaAtencion()
+ HashMap<Integer, IMedico> getMedicos()
+ String getNombre()
+ HashMap<Integer, Paciente> getPacientesRegistrados()
+ ArrayList<Paciente> getPatio()
+ Paciente getSalaPrivada()
+ String getTelefono()
PersistenciaBinaria
- FileInputStream fileinput
- FileOutputStream fileoutput
- ObjectInputStream objectinput
- ObjectOutputStream objectoutput
+ void abrirInput(String nombre)
+ void abrirOutput(String nombre)
+ void cerrarInput()
+ void cerrarOutput()
+ void escribir(Serializable serializable)
+ Serializable leer()
«java interface»
IPersistencia<E>
+ void abrirInput(String nombre)
+ void abrirOutput(String nombre
+ void cerrarInput()
+ void cerrarOutput()
+ void escribir(E objecto)
+ E leer()
Persistencia
+ void levantarDatos()
+ void persisteDatos()