# Informe TP Final POO2


## Profesor

Juárez, Andrés

## Alumnos

● Abogado, Martín E
● Cerchi, Fabián
● Coluccio, Nicolás

**2022**

# Índice

# Introducción

El proyecto aborda un escenario representativo de una Clínica/Institución de salud donde existen diferentes perfiles cuyas responsabilidades están diferenciadas.

Por un lado existe el rol único de director de la Clínica, quien tiene a disposición algunas funcionalidades de reportería.

Luego tenemos empleados administrativos quienes generan los turnos, dan el alta a nuevas prestaciones, entre otras funciones.

Existe el rol de Doctor, quien atiende a los pacientes, registra su asistencia y prescribe medicamentos u otras prácticas.

Por último, la figura de Paciente quien se atiende y puede abonar sus prestaciones.

# Decisiones de diseño

Para el diseño de esta aplicación manejada por consola, se decidió utilizar persistencia en listas y diccionarios.

La clínica es un único objeto y contiene toda la información anteriormente enumerada.
Los menú, y sus opciones, cambian dependiendo del rol.

Podemos clasificarlas en varias categorías. Por un lado, las de consulta, que leen información almacenada muchas veces por algún valor que puede interpretarse como un primary key.

También hay de creación, como lo son el registro de pacientes o de prestaciones.

De la misma manera, existe la baja (lógica, no física) como se puede apreciar en la inhabilitación de prestaciones.

Por último, las opciones de actualización, como lo son el cambio de estado en prestaciones, especialidades, doctores, entre otras.

Como valor agregado para generar robustez en los Menú, se creó la clase MenuHelper, la cual valida el ingreso de datos respecto de las opciones ofrecidas. Es sin duda una gran herramienta para implementar “improper input validation” o dicho de otra forma, la validación del ingreso de datos inapropiados.

De gran importancia también, podemos destacar la clase GeneradorDeDatos, cuya responsabilidad es instanciar data inicial para la clínica. De esta forma poblamos a la misma de:

doctores, pacientes, prestaciones, especialidades, turnos, sobreturnos, entre otras cosas.

Por último, queremos destacar la implementación que evita el solapamiento de turnos para con los doctores.

Por requerimiento, los doctores solo pueden atender una especialidad y prestación, lo que limita su agenda. Incluso al agregar una nueva prestación se valida y requiere la creación de un doctor para asociarlo.

La única forma de “solapar” es con un sobreturno, pero justamente esta es la función de los mismos. En la vida real, un sobreturno puede no llegar a efectuarse en el momento indicado.

# Patrones y prácticas empleadas

Para este desarrollo, se empleó el siguiente patrón de diseño:

- Singleton (apreciable en la Clínica)

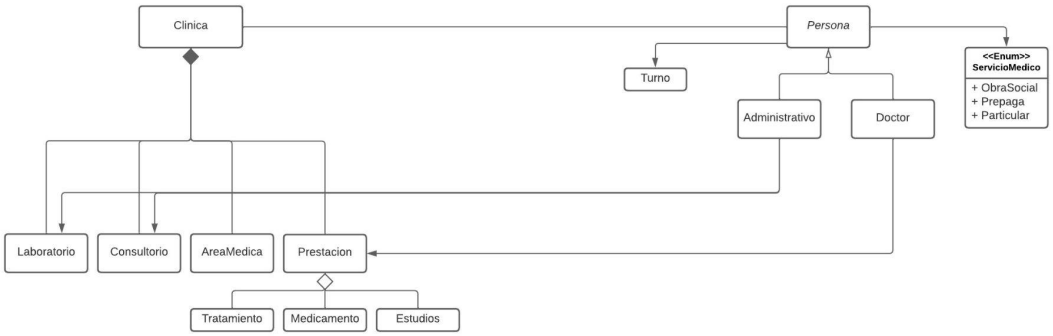
Adicionalmente se hizo uso de herencia, polimorfismo, uso de clases abstractas y refactorización iterativa del código.

Se utilizó como primera herramienta el diseño del diagrama conceptual.

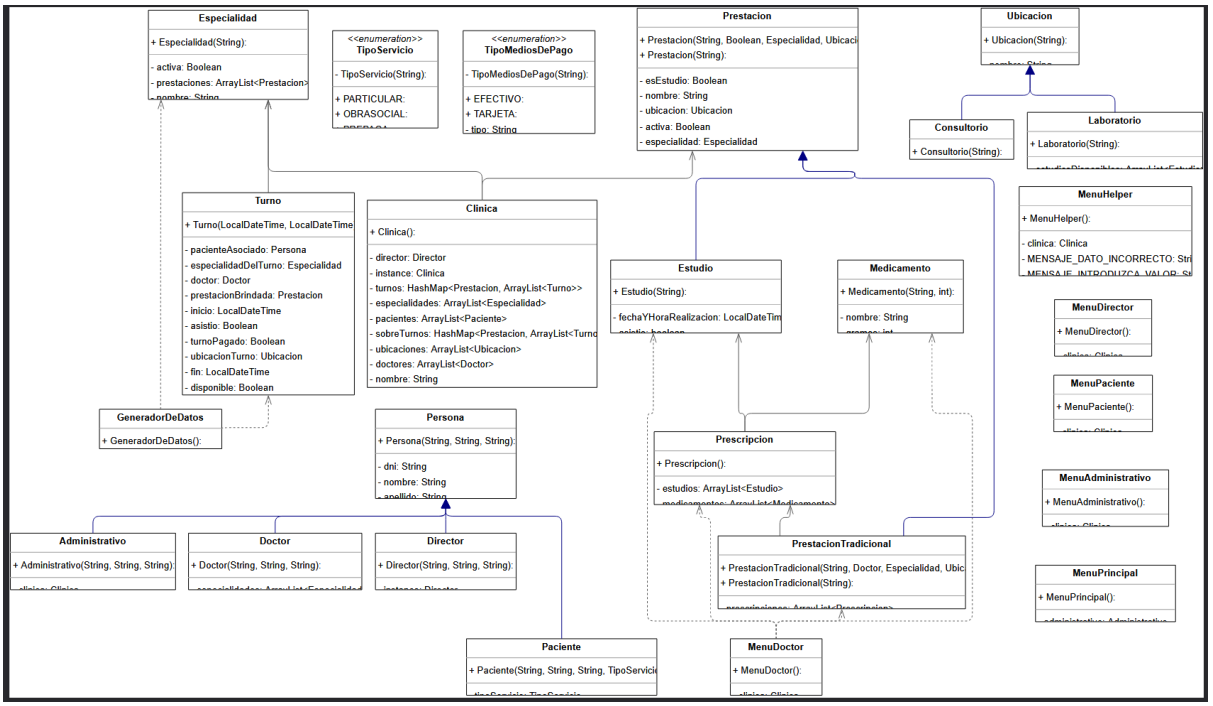
Si bien no se realizó TDD, se aplicó testeo manual y unitario en la medida que los módulos se desarrollaban.

# Diagrama UML

Inicialmente se diseñó el siguiente diagrama conceptual




pero luego, por necesidad de escalar la aplicación y con la aparición de nuevas figuras importantes en el desarrolló, finalizó de esta manera:



# Sobre los tests

Se utilizó JUnit para realizar tests unitarios nativos.
Se propuso un coverage mínimo del 80%, pero la mayoría de los casos este valor fue superado ampliamente, como en GeneradorDeDatos o Administrativo.

Nuestra premisa siempre fue la misma: preferimos mil veces un coverage bajo pero de real importancia, de nada sirve un coverage alto basado únicamente en el testeo de constructores, getters, setters, etc.

# Conclusiones

Detectamos que la primera versión entregada, quizás no fue la mejor, tanto en diseño, como en implementación, pero somos conscientes de ello y a continuación detallamos algunas mejoras aplicables para una eventual segunda versión.

Mejoras aplicables:

- Disminución de acoplamiento
- Aumento de uso de try-catch para el manejo de errores
- Mayor reutilización de componentes, como el MenuHelper
- Migración de diccionarios a listas
- Si lo anterior no es una opción, mejorar el key del diccionario es viable, reemplazando (por ej para los turnos, cuya firma es: `HashMap<Prestacion, ArrayList<Turno>> turnos`)
  por un HasMap<String, ArrayList<Turno>>.