Trabajo Práctico Grupal (TPG)

Parte II

Asignatura: Programación 3

Grupo Nº 4

Integrantes:

Bengoa, Sebastián
Echeverría, Noelia
Ezama, María Camila
Mateos, Juan Cruz

Introducción

Este trabajo tiene como propósito la generación de un sistema de gestión para una clínica privada. El sistema será implementado, utilizando el lenguaje de programación Java y el paradigma de programación propuesto por la cátedra es el Orientado a Objetos. Dependiendo del contexto en donde estén enmarcadas cada una de las problemática asociadas a los módulos del sistema, se utilizarán distintos patrones de diseño. Como se mencionó en la primera entrega el programa debe ser extensible, para dar soporte a nuevas funcionalidades.

Requerimientos y descripción del sistema implementado

En esta segunda entrega se han agregado algunas funcionalidades al programa implementado con anterioridad, tales como:

- Interfaz gráfica para confección de factura para cliente (a.k.a paciente)
- Nueva categoría: asociados
    - Pueden solicitar ambulancia para atención a domicilio
    - Pueden solicitar ambulancia para traslado a la clínica
- Interfaz gráfica para dar de alta y baja a asociados
- Interfaz gráfica para definir simulación. Se debe poder definir:
    - Cantidad de asociados que utilizarán los servicios de clínica
    - Cantidad de solicitudes que hará cada uno
- Persistencia de la información de la clínica.
    - Persistencia binaria
    - Persistencia XML
    - Persistencia con bases de datos (a implementar a futuro)

Desarrollo de la aplicación

Nuevas funcionalidades:

- Sistema de asociados a la clínica
- Sistema de ambulancia
- Interfaz gráfica

Sistema de asociados a la clínica

La clínica cuenta con un servicio de asociados. Los mismos pueden solicitar dos servicios distintos, ambos relacionados con la utilización de la ambulancia:

- Traslado a la clínica
- Atención en domicilio

Para que una persona pueda asociarse (darse de alta), debe solicitarlo explícitamente, brindando sus datos (y obviamente pagando una módica suma de dinero).

Datos necesarios para asociarse:

- Nombre
- Apellido
- DNI
- Dirección
- Teléfono

Selección de herencia para la nueva clase Asociado

Para el modelado de la clase cliente se contemplaron dos posibles clases de las cuales heredar el comportamiento:

- Clase Persona (abstracta)
- Clase Paciente (abstracta)

Desde un punto de vista lógico un asociado es potencialmente un paciente, pero nos encontramos con el inconveniente de que dentro de los datos solicitados obligatorios no se encontraba el rango etario como requisito (el cual es utilizado para generar las clases concretas PacienteJoven, PacienteAnciano y PacienteNino a través del patrón factory). De igual manera decidimos heredar de la clase Paciente debido a que una de las acciones relacionadas a un asociado es pedir traslado a la clínica. En esta entrega no se modela el comportamiento una vez que el asociado llega a la clínica quedando como una tarea a implementarse en futuras revisiones del programa.

Características de la clase Asociado

El comportamiento que caracteriza a un asociado es solicitar una ambulancia para traslado o para atención domiciliaria.

Es factible (y de hecho, altamente probable) que varios asociados intenten obtener el servicio de ambulancia en simultáneo. De esta manera, la ambulancia se transforma en un recurso compartido en disputa por, en principio, más de un asociado (veremos más adelante que los asociados no son los únicos interesados en obtener la "atención" de la ambulancia).

Los Asociados se modelan de esta manera como "hilos" o "threads" que compiten por la atención de la ambulancia.

Debido a que los Asociados son Pacientes (gastando así la herencia), para modelar su comportamiento concurrente deberán implementar la interface Runnable.

Dentro del método Run() de cada objeto de tipo Asociado estará implementado el comportamiento relacionado con la obtención del bloqueo del recurso compartido.

Operador de la ambulancia

Se contempla que la ambulancia pueda romperse o necesitar algún tipo de mantenimiento preventivo. El encargado de enviar la ambulancia al taller será un Operador (empleado de la clínica cuyo único trabajo es mandar a la ambulancia al taller). La clase Operador heredará de la Clase Persona, siendo que todos los empleados de la clínica son personas.

Debido a que el Operador también luchará por obtener la atención de la Ambulancia, se modela al igual que un Asociado como una clase que implementa la interfaz Runnable.

Temporizador de retorno a clínica

El pasaje de la ambulancia de un estado a otro se maneja a través de hilos que le envían mensajes (o solicitudes) para ejecutar una acción determinada en caso de ser posible. Es por ello que para lograr que la ambulancia pueda pasar al estado "Disponible en Clínica" (estado 1) creamos una clase llamada Temporizador, cuyo único trabajo es solicitarle a la ambulancia que retorne a la clínica.

La clase Temporizador también luchará por la atención de la ambulancia, pero en este caso, al heredar directamente de Object, la herencia simple está aún disponible, por lo que en lugar de implementar la interface Runnnable, directamente heredará de la clase Thread.

Sistema de ambulancia

La clínica cuenta con un sistema de ambulancias (1 ambulancia) el cual puede ser únicamente utilizado por los asociados a la clínica.

Dependiendo de la solicitud que se haya hecho a la ambulancia, puede estar en distintos estados. En función del estado en el que se encuentre, al llegarle un nuevo mensaje (de alguno de los hilos que intentan acaparar su atención), la misma responderá de manera diferente a tales solicitudes. Cada vez que la ambulancia cambia de estado, debe informar a las clases interesadas.

La Ambulancia tendrá por un lado threads luchando por obtener su atención y por otro lado, su respuesta dependerá del estado en el que se encuentre cuando reciba estos mensajes.

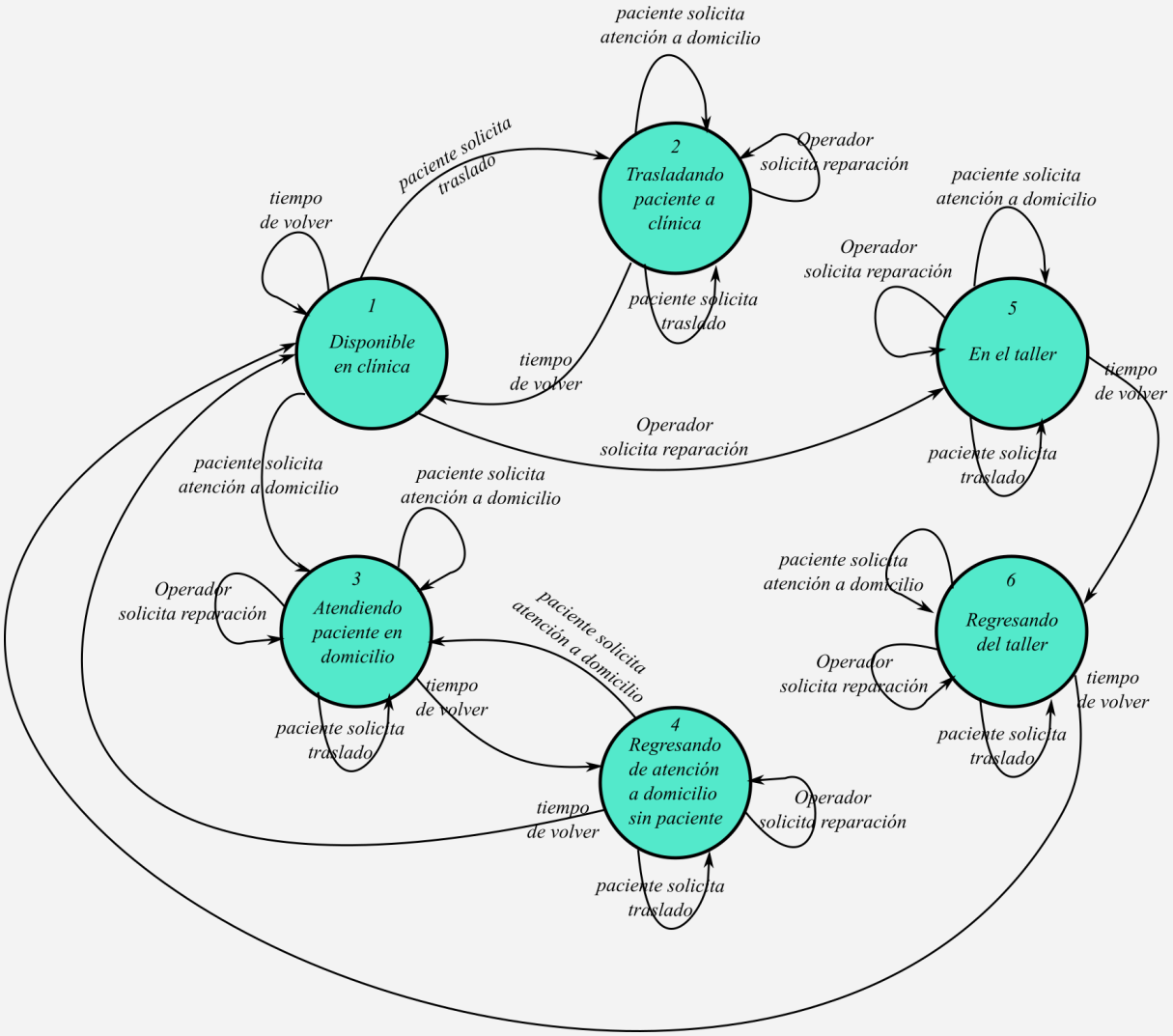
Por último, es necesario mencionar que uno de los requisitos solicitados es que se debe permitir la visualización del estado de la ambulancia en cada instante.

Teniendo en cuenta esto, para el modelado de la Clase ambulancia decidimos aplicar los siguientes patrones de diseño:

- Patrón State: para modelar la evolución de los estados de la ambulancia en función de los mensajes recibidos y el estado actual de la misma
- Patrón singleton: debido a que se dispone de una única ambulancia
- Patrón Observer Observable: para notificar de los cambios de estados a las clases interesadas (clase/s observadora/s).

Además, al ser un recurso compartido, se utilizarán métodos sincronizados para asegurar exclusión mútua de hilos. En la figura (1) se muestra un diagrama de estados que ejemplifica cómo reacciona la ambulancia a los mensajes de acuerdo al estado en el que se encuentre.

Para ver el diagrama de clases ingresar a : https://bit.ly/35ViOE3



(Para ver el diagrama de estados con mayor detalle y definición ingresar en el siguiente enlace: https://bit.ly/35WdHbQ)

Interfaz gráfica y patrón MVC

El patrón MVC permite separar la lógica de negocio de la interfaz de usuario y así repartir responsabilidades. Se divide en 3 componentes cómo indican sus siglas (modelo, ventana y controlador).

La ventana es la encargada de comunicarle al controlador que es lo que desea el usuario. El controlador en base a lo comunicado modificará el modelo si es necesario y se comunicará con la ventana nuevamente.

En nuestro caso la clase Ventana implementa una interfaz llamada IVista (modela el comportamiento de la Ventana), la clase Controlador implementa

ActionListener y WindowListener para actuar de acuerdo a lo que diga la Ventana y el modelo es la clase Clínica.

Persistencia de la información y serialización

Como requisito de implementación se tiene que la persistencia de la información se hará mediante serialización y archivos.

Cada vez que se inicia la aplicación se deben leer todos los archivos para levantar en memoria todos los datos. Cada vez que la aplicación finaliza (y en cada demanda del usuario) se deben persistir los datos para actualizar los archivos.

Para lograr la persistencia brindamos en una primera instancia la posibilidad de utilizar

- Persistencia Binaria
  - La clase a persistir y todas sus variables de instancia deben ser serializables.
  - Deben implementar la interface Serializable
- Persistencia XML
  - La clase a persistir y todas las variables de instancia que quieran ser persistidas, deben tener un constructor vacío.
  - Deben exponer las propiedades que constituyen el estado del objeto mediante getters y setters.

Información a persistir

La información que se desea persistir es toda la relacionada con el estado de la clínica en un determinado instante de tiempo.

En primer lugar, la Clínica lleva el **registro de los médicos** que trabajan en ella. Además, debemos recordar que dentro de la clínica se encuentran los siguientes módulos:

- **Módulo de ingreso:** Encargado de llevar el registro histórico de pacientes, el registro de pacientes esperando por atención en el patio y el paciente en la sala VIP (si hay alguno).
- **Módulo de atención:** encargado de llevar el registro de los pacientes en atención
- **Módulo de atención ambulatoria**: encargado de llevar el registro de los asociados a la clínica.
- **Módulo de egreso:** encargado de llevar el registro de todas las facturas realizadas a los pacientes

Para ver un diagrama de clases ingresar a: https://bit.ly/3dmbzOx

Persistencia binaria:

Como mencionamos con anterioridad, para lograr una persistencia binaria es necesario que tanto la clase madre (Clínica) como todas sus variables de instancia puedan ser serializables.

En este punto nos enfrentamos con el primer problema que es que la clínica utiliza el patrón singleton. El problema que surge con los patrones singleton es que el constructor es privado, por lo cual la instanciación se realiza a través de una variable estática que llama al constructor cuando la variable es null o retorna la instancia creada con anterioridad en cualquier otro caso (ver figura 2). Recordemos que los miembros estáticos y los marcados como transient no se persisten.

```
public class Clinica {
    private static Clinica instance = null;
    private String nombre;
    private String direccion;
    private String ciudad;
    private int telefono;
    private HashMap<Integer, IMedico> medicos = new HashMap<>();
    private ModuloIngreso moduloIngreso = new ModuloIngreso();
    private ModuloAtencion moduloAtencion = new ModuloAtencion();
    private ModuloEgreso moduloEgreso = new ModuloEgreso();
    private ModuloAtencionAmbulatoria moduloAtencionAmbulatoria = new ModuloAtencionAmbulatoria();

    private Clinica() {
        // Constructor vacio
    }

    public static Clinica getInstance() {
        if (Clinica.instance == null)
            Clinica.instance = new Clinica();
        return Clinica.instance;
    }
```

Persistencia XML:

Cómo se mencionó previamente, para persistir información utilizando XML necesitamos que la clase a persistir y sus variables de instancia ofrezcan un constructor sin argumentos, getters y setters.

Tanto la clase persona y las que extienden de ella implementan directamente la interface serializable. En tanto que para las clases con atributos estáticos utilizamos el patrón DTO para serializar, creando una clase ClinicaDTO en donde almacenamos el valor de los atributos estáticos (implementando todos los getters y setters). Posteriormente creamos una clase llamada DTOConverter con 2 métodos (ClinicaDTOFromClinica y ClinicaFromClinicaDTO) para realizar la serialización y la deserialización.

Conclusiones:

Durante el desarrollo de la segunda parte de este trabajo hemos podido explorar nuevos patrones de diseño tales como el Observer Observable, Patrón State, MVC, DAO/DTO y complementarlos con los implementados con anterioridad.

Hemos aprendido la importancia de generar código con posibilidad de ser extendido, sin la necesidad de hacer un refactor completo del mismo. Gracias a esto nos fue relativamente sencillo agregar las nuevas funcionalidades solicitadas. De igual manera debemos reconocer que con la información actual y todo lo aprendido al día de hoy, si tuviéramos que encarar nuevamente un proyecto similar, el diseño de clases hubiera sido ligeramente distinto, para asegurar una mayor escalabilidad.

En esta segunda entrega dos de las cosas más desafiantes y gratas fueron, por un lado poder generar una interfaz gráfica funcional y por el otro lado poder persistir los datos cargados en el sistema.

Por último queremos mencionar que ha sido desafiante y a la vez gratificante aprender a distribuir las tareas, siendo de vital importancia la puesta en común de criterios a la hora de generar el código.