

Facultad de Ingeniería-UNMDP

Programación “C”

**INFORME TRABAJO**  
**PRÁCTICO GRUPAL**  
**-SEGUNDA PARTE-**

*Fecha de entrega: 16/11/2025*

**Docentes a cargo:**

Leonel Guccione, Guillermo Lazzurri, Ivonne Gellon

**Integrantes:**

Salvador Munduteguy, Federico Calla Aliende, Francisco Marino,

Iván Etcheverry y Santiago Soldani

## Introducción

Este informe tiene como objetivo describir la manera en que se pensó y llevó a cabo el sistema de funcionamiento y representación de una clínica. Se explicará la organización de clases, paquetes, métodos relevantes que colaboran con el objetivo del proyecto, excepciones, problemas que se presentaron en el proceso, así como sus respectivas soluciones implementadas.

## Objetivos del Proyecto

El objetivo principal para esta segunda parte del proyecto es generar un sistema para una clínica, con el cual puedan realizar las siguientes tareas:

- Manejo del recurso compartido (ambulancia) con sus hilos
- Realizar una interfaz gráfica
- Creación y tratamiento de una base de datos (altas, bajas)
- Aplicar patrones abordados durante la cursada

## Herramientas Utilizadas

Las siguientes herramientas han sido seleccionadas para el desarrollo:

Herramienta	Descripción
Lenguaje de Programación	Java
IDE (Entorno de Desarrollo Integrado)	Eclipse
Herramientas Adicionales	Gemini, Figma, Chat gpt, "Thinking in Java 4th edition", Swing (Framework), "Desarrollo de proyectos informáticos con tecnología Java"

## Conexión base de datos

Base de datos	grupo_3
usuario	progra_c
contraseña	progra_c

## Análisis del Problema:

Para abordar esta segunda parte del Tp grupal, al igual que lo realizado previamente, se realizó un análisis del problema y bosquejo de la simulación a lograr (que entidades participaran y de qué forma). Como novedad se añade una parte de diseño, las ventanas que visualizará el usuario, encargadas de mostrar datos relevantes y resultados.

En principio se dividió el trabajo en dos partes, modelo y vista. En cuanto al modelo, se debió pensar los distintos estados pertenecientes a la ambulancia, junto con qué acciones permiten su cambio, además de que deberá manejar la concurrencia de los hilos (peleando por tomar su control). Por el lado de la vista, se notó un leve congelamiento del programa, cuando un hilo era "dormido" por el wait; por ende, se implementó un nuevo runnable, el cual hará un new hilo llamador. Ahora será este hilo llamador el que quedará congelado, sin afectar la experiencia del usuario. En conjunto, se trataron los botones, modificándose, a modo que cuando se esté ejecutando un hilo, el botón se desactiva. Mientras que cuando este finaliza su ejecución, por medio de un Observer-Observable, se volverá a activar para el usuario.

Cabe destacar la realización y tratamiento de la base de datos. En esta sección, se consideró la opción de hacerla lo más automática posible. Por ello, luego de crear la base de datos, es el mismo sistema el cual se encargará de crear la tabla de asociados, en caso de no tenerla, para cada base de datos local.

## Desarrollo del sistema

### *Clases Fundamentales:*

#### **Asociado**

Serán los hilos de la concurrencia que pelearán por tomar el control del recurso compartido (ambulancia). Poseen un código sencillo, con un método run(), que aleatoriamente seleccione si solicita traslado a clínica o atención domiciliaria. Para ello toma el RC, deja pasar un tiempo y lo libera finalmente, para ser tomado por otro.

Además es una clase que extiende de Thread, y posee una relación de composición con respecto a la clase Persona (de esta forma se solucionó el conflicto de herencia de tipo). Junto con esto, en su constructor se le asigna la ambulancia sobre la que actuar.

## **Ambulancia**

Es el recurso compartido, donde se aloja el código más “complejo” que maneja los hilos. Tiene métodos sincronizados que se encargan, ya sea, de ir a domicilio del asociado o de trasladarlo, ejecutando de principio a fin uno a la vez. Se ayuda de variables primitivas, que funcionan a modo de banderas (ej.: atendiendo, entaller), para saber si la ambulancia se encuentra disponible o no; con los métodos correspondientes se activan o desactivan.

### ***Patrones destacables:***

#### **Patrón State**

En el módulo Ambulancia se implementa el Patrón State para modelar los distintos estados operativos que puede atravesar el vehículo (Disponible, AtendiendoPaciente, TrasladandoPaciente, EnTaller, RegresandoDeAtencion, etc.). La clase Ambulancia mantiene una referencia al estado actual y delega en él todas las operaciones relacionadas con solicitudes de atención, traslados, mantenimiento y retornos automáticos.

Cada estado es responsable de definir qué acciones están permitidas y cuáles deben ser rechazadas, pudiendo además modificar el estado del objeto cuando corresponda. Esto permite eliminar condicionales complejos dentro de Ambulancia, mejorar la legibilidad, encapsular las reglas de transición y facilitar la extensión del sistema incorporando nuevos estados sin modificar la lógica existente. Gracias al Patrón State, la ambulancia cambia su comportamiento dinámicamente según su estado interno, representando de manera fiel el funcionamiento real del servicio de emergencias.

#### **Patrón DAO y DTO**

En lo que respecta a la capa de persistencia y estos patrones, se organizó de la siguiente manera. Se tiene una clase “ConexionManager”, la cual es la encargada de levantar la base de datos y conectarse a la misma. Tiene un método que permite chequear cuando se quiera el estado de la conexión. Sumado a ello, se desarrolló un método el cual detecta si se tiene creada la tabla de asociados, en caso negativo, se encarga de crearla en la base de datos local (lo que facilita el proceso didáctico).

Por otro lado, la clase “AsociadoDAO” es quien permite dar de alta y/o baja a los asociados en la base de datos. Realizando las consultas sql necesarias, para dar de alta a un asociado, en primer lugar chequea que el mismo no esté cargado previamente; esto lo hace por clave única Dni. Mientras que la baja es similar, analiza que exista el usuario deseado, para luego, en caso afirmativo, borrarlo de la base de

datos. Junto con ello, tiene la capacidad de recopilar los datos de toda la tabla, para listarlos y enseñarlos por pantalla al usuario.

Luego, se encuentra la clase Asociado DTO (Data Transfer Object) que cumple la función de transportar datos de un asociado entre las distintas capas del sistema sin exponer la lógica interna ni las estructuras complejas del dominio. Con esto se busca separar responsabilidades para mejorar el mantenimiento, escalabilidad y claridad del código. Cuando la información debe viajar hacia la capa de presentación (interfaces gráficas, formularios), la capa de persistencia (acceso a base de datos), o servicios externos, no se quiere exponer directamente la entidad de dominio ya que suelen tener lógica interna, validaciones, relaciones con otras clases, reglas de negocio o estados internos que no deben ser modificados desde afuera.

Finalmente, la clase AsociadoMapper que tiene como función principal convertir objetos del modelo de dominio en objetos DTO y viceversa. Esto permite mantener un fuerte desacoplamiento entre las capas del sistema: la lógica de negocio trabaja con entidades completas, mientras que la capa de persistencia y la interfaz gráfica solo reciben datos simples encapsulados en un AsociadoDTO. Esta separación mejora la mantenibilidad, reduce el acoplamiento entre módulos y facilita la serialización, persistencia y transporte de información dentro del sistema.

### ***Paquetes:***

Con respecto a la primera entrega, se realizó una reorganización de los paquetes del proyecto, en mayor medida por la presencia del patrón MVC, con la intención de dejar bien en claro las discrepancias entre el modelo, la vista y el controlador. Dentro de cada uno se pueden observar sub-paquetes, con la intención de mantener un mayor orden y entendimiento del código. Todos los paquetes están dentro del src (paquete que indica donde estará el código de la app).

### **Modelo.Ambulancia**

Aquí se puede encontrar todo lo referente al manejo de la concurrencia (hilos y rc). En conjunto, el patrón State, con todas sus clases de estado. Es uno de los paquetes fundamentales dentro del modelo (de esta segunda parte).

### **Persistencia**

Se declara y trabaja la capa de persistencia. Se tratan los patrones relacionados con las bases de datos y su tratamiento, como la serialización de la clase asociados.

## **Controlador.AsociadosController**

En este archivo dentro del paquete de controladores se creó un intermediario entre la ventana de Asociados (vista) y la clínica (modelo), de esta forma, a partir de las interacciones del usuario para agregar, eliminar y listar asociados, este controlador se encarga de llamar a los métodos correspondientes de la clínica. Esta clase implementa la interfaz action listener, y maneja 3 action commands provenientes de la ventana ya mencionada.

## ***Simulación***

Para resolver el problema de la simulación, se realizó un trabajo que reunió todas las partes del proyecto. Se comenzó un modelo (modelo.simulacion) el cuál encapsula los objetos que participarán en la simulación. Esto se diseñó a modo de simplificar y unificar los elementos.

Se contó con un controlador, el cuál estaba compuesto por este modelo, y a su vez, se aplicó el patrón Observer-Observable, ya que los cambios producidos en el modelo, al momento de la ejecución de la simulación, repercutirían en la vista, por lo que este patrón nos fué de mucha utilidad.

Por supuesto que tuvimos que implementar una interfaz gráfica, la cuál mostraría distinta información, y a través de la misma podríamos "comunicarnos" con el usuario. A continuación se desarrollará un poco más sobre el proceso de la simulación, numerando los puntos claves a tener en cuenta, problemáticas ocurridas y sus respectivas soluciones.

## **Modelo.simulacion**

Como hemos nombrado, esta clase, encapsula distintos atributos y objetos que serán partícipes de la simulación. Cuenta con diversos atributos que utilizamos para controlar ciertos comportamientos, pero lo más importante es que cuenta con un listado de Asociados, y gracias a esto podremos "activar/desactivar" los hilos concurrentes, y además es de suma importancia ya que estos objetos serán Observados por el controlador.

## **Controlador.SimulacionController**

El controlador se encarga de manejar las actualizaciones en pantalla y el estado de la ejecución. Es una pieza clave ya que este interpretará los distintos eventos que ocurren durante la ejecución del programa, y responderá a cada uno de ellos.

Un punto clave que nos gustaría destacar es que este controlador estaba pensado solo para una ventana, pero luego quisimos implementar una ventana nueva el cuál, de manera simultánea a la ejecución de la simulación, nos mostraría la evolución individual a lo largo de la misma de cada participante. Y para evitar cortar la simulación, crear un nuevo controlador, generando un alto acoplamiento entre los objetos, decidimos que lo mejor era, de alguna forma, reutilizar el controlador que ya teníamos, y extender los métodos implementados para poder manejar más eventos, respetando lo más posible el patrón MVC.

## **InterfazGrafica: .VentanaSimulacion & .VentanaEvolucionAsociado**

Aquí tenemos las ventanas que serán la parte visible hacia el usuario. Mediante las mismas, ambas conociendo su controlador, logramos mostrar por pantalla:

- Estado actual de la ambulancia ('Disponible', 'En el taller', 'Atendiendo' o 'Trasladando')
- Los movimientos ocurridos en tiempo real
- Las solicitudes realizadas por cada asociado participante.

Además contamos con un panel central, en el cuál podremos interactuar con la simulación, definiendo la cantidad de asociados a simular, la cantidad de solicitudes por asociado que se realizarán, y tres botones los cuales son:

<b>COMENZAR</b>	Comienza la ejecución con los parámetros dados
<b>SOLICITAR MANTENIMIENTO</b>	Comienza la ejecución del hilo del operador de la ambulancia.
<b>FINALIZAR</b>	Finaliza la simulación, sin "matar los hilos"

## **Interfaz Gráfica**

Una de las implementaciones destacables de esta segunda parte del trabajo es la interfaz gráfica.

### **Herramientas utilizadas:**

**Figma:** Figma es una plataforma de diseño colaborativo basada en la nube para crear interfaces de usuario. Fue de gran ayuda dado que permitió diagramar esta parte del proyecto, junto con su estructura y funcionalidades de una forma simple.

**Swing:** Se utilizó esta herramienta para implementar las interfaces gráficas en Java. Swing proporciona una gran cantidad de clases y librerías para desarrollar aplicaciones.

**WindowBuilder:** Windowbuilder fue de gran ayuda en la implementación de interfaces dado que proporcionó una vista en tiempo real de los cambios que hacíamos en ella.

Toda la interfaz gráfica se encuentra en el paquete "InterfazGrafica", dentro de este, la misma se divide en distintas ventanas (JFrame / JDialog)

### **Prueba**

Entry point de la app y donde se ejecuta la clase Prueba con su método estático main para comenzar a ejecutar el sistema.

### **Excepciones**

Se agregaron excepciones ante violaciones de reglas (por ejemplo, no ingresar un nombre, dni en la carga de asociados).

### **Diagrama UML**

Diagrama UML adjunto aparte.



## Conclusiones

La segunda parte de este proyecto permitió visibilizar cómo funciona un programa en el área informática (a groso modo). Desde la intervención del humano, como se van comunicando y ensamblando las partes del código para modificar/ejecutar lo solucionado, hasta almacenar y guardar información en una base de datos. Estas capas siempre están presentes de alguna manera, por lo que en este sentido el trabajo fue muy enriquecedor.

Además, la inclusión de asserts facilita en gran medida la detección y corrección de errores, en etapa de desarrollo. En proyectos a gran escala, donde una función, múltiples veces implementada, puede fallar por infinidad de causas, al incluir un assert, se puede detectar que parte del código está fallando concretamente.

En cuanto a las ventanas, agregar que ante diversas opciones sobre cómo modelarlas, se prioriza la simpleza de su implementación, a modo que la experiencia del usuario sea gratificante y práctica. A veces es mejor algo fácil de entender, que algo, tal vez visualmente más estético, pero que sus funcionalidades no estén tan claras.