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

Conexion base de datos:

paquete persistencia -> clase conexionManager -> línea 10.

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 soluciono 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.

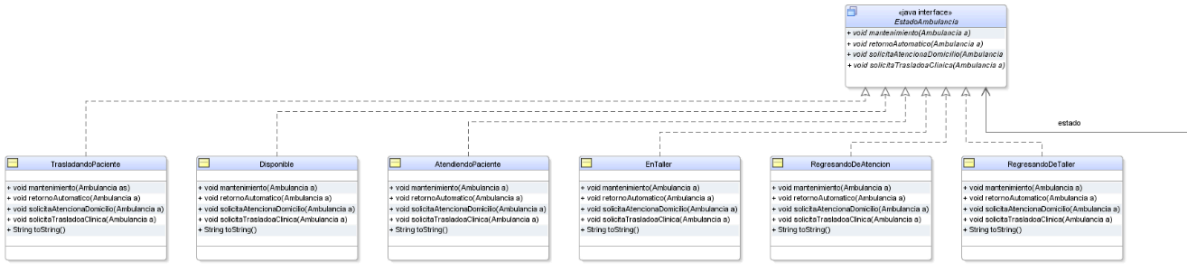
Patrones destacables:

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.

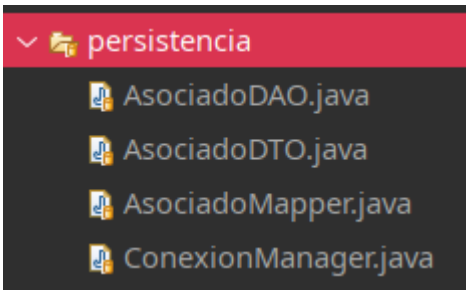
```java
public interface EstadoAmbulancia {
    void solicitaAtencionaDomicilio(Ambulancia a) throws SolicitudNoAtendidaException;
    void solicitaTrasladoaClinica(Ambulancia a) throws SolicitudNoAtendidaException;
    void retornoAutomatico(Ambulancia a) throws SolicitudNoAtendidaException;
    void mantenimiento(Ambulancia a) throws SolicitudNoAtendidaException;
}
```

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 crea la base tabla de asociados si el usuario así lo desea (opción de inicialización), borrando la tabla anterior si es que existe y creando una nueva. Además, este método agrega unos asociados de prueba para usarla sin necesidad de agregar asociados a mano.

```java
// ⭐ NUEVO MÉTODO PARA CREAR LA TABLA SI NO EXISTE
public static void inicializarEsquema() {
    try (Connection conn = getConnection(); // Abre la conexión
         Statement stmt = conn.createStatement()) { // Crea un Statement para ejecutar SQL

        stmt.execute(SQL_ELIMINACION); // Ejecuta la sentencia DDL (CREATE TABLE)
        stmt.execute(SQL_CREACION); // Ejecuta la sentencia DDL (CREATE TABLE)
        stmt.execute(SQL_ASOCIADOS); // Ejecuta sentencia para insertar Asociados
        System.out.println("✅ Esquema verificado/creado: Tabla 'Asociados' lista.");

    } catch (SQLException e) {
        // Este bloque captura errores si, por ejemplo, la base de datos 'grupo_3' no existe.
        if (e.getSQLState().startsWith("42")) {
            System.err.println("ERROR FATAL: La Base de Datos 'grupo_3' no existe o hay un problema de credenciales.");
            System.err.println("Solución: El compañero debe crear la BD vacía 'grupo_3' en su servidor MariaDB.");
        } else {
            System.err.println("ERROR al inicializar el esquema: " + e.getMessage());
        }
    }
}
```

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

```java
public void altaAsociado(Asociado asociado) throws SQLException, IllegalArgumentException {
    AsociadoDTO dto = AsociadoMapper.toDTO(asociado);
    String dni = dto.getDni();

    // 1. Verificación de duplicados
    if (existeAsociado(dni)) {
        throw new IllegalArgumentException("Ya existe un asociado con el DNI: " + dni);
    }

    // 2. Sentencia INSERT con los 5 campos de la tabla
    String sql = "INSERT INTO Asociados (dni, nombre, telefono, ciudad, domicilio) VALUES (?, ?, ?, ?, ?)";

    try (Connection conn = ConexionManager.getConnection();
         PreparedStatement ps = conn.prepareStatement(sql)) {

        ps.setString(1, dto.getDni());
        ps.setString(2, dto.getNombreyapellido());
        ps.setString(3, dto.getTelefono());
        ps.setString(4, dto.getCiudad());
        ps.setString(5, dto.getDomicilio());

        ps.executeUpdate();
        System.out.println("Asociado con DNI " + dni + " dado de alta exitosamente en BD.");
    }
}
```

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.

Patron Observer-Observable:

El módulo de simulación fue, sin dudas, el módulo que más trabajo necesito. Uno de los patrones de diseño que nos ayudaron para el desarrollo del módulo fue el patrón Observer-Observable.

El problema a resolver era actualizar el estado de la ambulancia a la vista para que se pueda ver, en tiempo real, el estado de cada asociado que solicitaba una ambulancia. Para ello, planeamos un ojo/observer para que sea notificado cada vez que una

ambulancia cambiaba su estado a alguno de los mencionados en el patrón State. Como la actualización era a la vista y estaba estrechamente relacionado con el patrón MVC, decidimos que nuestro ojo sería también el controlador de la simulación, y en consecuencia implementa el método update() y el método ActionPerformed.

```java
/**
 * Metodo del controlador, el cual es llamado cada vez que ocurre un cambio por parte de los objetos que observa.
 * recibe al observable, y un String que cierto formato el cual es analizado para saber donde actualizar
 */
@Override
public void update(Observable o, Object arg) {
    String mensaje = (String) arg;
    VentanaSimulacion ventana_sim = (VentanaSimulacion) this.vista;

    SwingUtilities.invokeLater(() -> {
        if (mensaje.startsWith("ESTADO:")) {
            String estado = mensaje.replace("ESTADO:", "").trim();
            if (estado.equalsIgnoreCase("En el taller")) {
                this.manteniendo = false;
            }

            ventana_sim.actualizarEstado(estado);

        } else if (mensaje.startsWith("LOG: ")) {
            String log = mensaje.replace("LOG: ", "").trim();
            ventana_sim.actualizarLog(log);
        }

        if (this.manteniendo) {
            ventana_sim.getBtnMant().setEnabled(false);
        } else
            if (this.modelo.enTaller())
                ventana_sim.getBtnMant().setEnabled(false);
            else
                if (this.deteniendo)
                    ventana_sim.getBtnMant().setEnabled(false);
                else
                    ventana_sim.getBtnMant().setEnabled(true);
        });
    }
```

De esta manera, cada vez que la ambulancia cambia de estado, el SimulacionController se encarga de notificarle a la vista que muestre el estado actual de la ambulancia.

Por el lado del observado (ambulancia), este nada más se encarga de cada vez que cambia de estado notificarle a todos los ojos (en este caso unos solo), que está realizando una acción diferente. El resultado final converge a que cada vez que la ambulancia cambia de estado, notifica al controlador que es al mismo tiempo el ojo, quien pasa las nuevas propiedades a la vista para que muestre el cambio al usuario.

```java
public synchronized void pedirTraslado(Asociado a) {
    while(this.atendiendo==1) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    this.atendiendo=1;
    this.setEstado(new TrasladandoPaciente());
    a.getSolicitudes().add(a.stringTraslado());
    setChanged();
    notifyObservers("LOG: "+a.toString1());
    notifyAll();
}
```

Patron MVC (modelo-vista-controlador):

Para esta parte del trabajo tuvimos que implementar una de las partes más fundamentales del desarrollo de cualquier sistema, La interfaz gráfica o interfaz de usuario.
Nosotros tenemos 3 controladores, sin embargo, para explicar nuestra implementación del patrón de diseño, voy a recurrir a la organización de las carpetas y al ejemplo de la Clínica (modelo), VentanaAsociados(vista), AsociadosController(controlador).

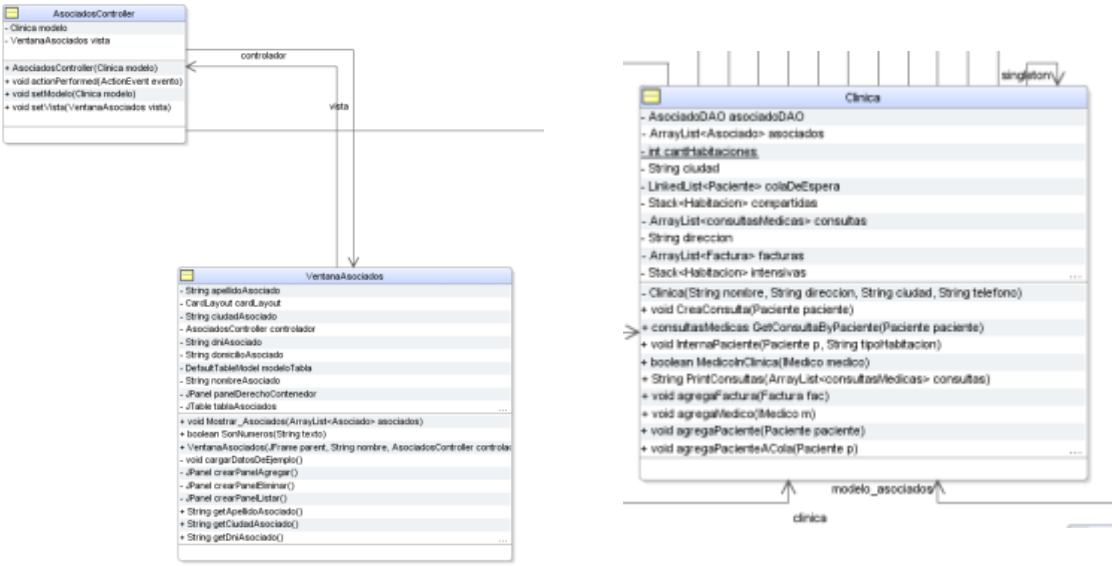
El modelo, encapsula la información de los asociados, se encarga de agregar asociados, crearlos, juntarlos en alguna colección y eliminarlos, pero no tiene idea de cuándo o con qué datos debe realizar sus tareas.

```
public void nuevoAsociado(String nombre, String apellido, String dni, String telefono, String domicilio, String ciudad) throws AsociadoInvalidoException
{
        try {

        nombre.concat(apellido);
        if(!nombre.matches(".*\\d.*")) {
            if(dni.matches("^\\d+$")) {

            Persona p = new Persona(dni,nombre,telefono,domicilio,ciudad);
            Asociado a = new Asociado(p,null);
            this.asociados.add(a);
            this.asociadoDAO.altaAsociado(a);
            }
            else {
                throw new AsociadoInvalidoException("El DNI tiene formato invalido");
            }
            }
        else {
            throw new AsociadoInvalidoException("El nombre y/o apellido tiene un formato invalido");
        }
        }
        catch(IllegalArgumentException e) {
            throw new AsociadoInvalidoException(e.getMessage());
        }
        catch(SQLException SQLe) {
            System.out.println(SQLe.getMessage());
        }
    }
```

En contraste, la clase Ventana Asociados simplemente entiende de interacciones con el usuario. algunos campos de texto que tiene que guardar, algún botón para agregar o eliminar asociados etc. sin embargo no sabe qué hacer con la información, simplemente muestra o lee información relevante para el usuario.

Aquí es donde aparece AsociadosController, que se encarga de recibir la información y/o las decisiones del usuario y decirle a la Clínica que es lo que tiene que hacer con la información que recibe, tomando decisiones, escuchando eventos y devolviendo resultados.

```
else if(evento.getActionCommand().equalsIgnoreCase("EliminarAsociado"))
{
    String dni = null;
    try {
        dni = this.vista.getDniAsociado();
        System.out.println(dni);
        this.modelo.removeAsociado(dni);
        this.vista.popUp("asociado eliminado con exito");
    }
    catch(AsociadoInvalidoException e)
    {
        this.vista.popUp(e.getMessage());
    }

}
```



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 a 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, repercutirán 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, enumerando 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 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:

COMENZAR	Comienza la ejecución con los parámetros dados
SOLICITAR MANTENIMIENTO	Comienza la ejecución del hilo del operador de la ambulancia.
FINALIZAR	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 permitio 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.