# Informe Trabajo Práctico Grupal 2° Cuatrimestre 2025 Segunda Parte


Integrantes: Gabriel Seneca
Mia Casanovas Di Sera
Octavio Laz
Mario Arriaga

# Índice

# Introducción:

En esta segunda parte del trabajo vamos a modelar el uso de una ambulancia, la cual podrá ser sometida a una simulación donde los asociados y/u operarios podrán realizar solicitudes de forma concurrente con la misma. Además, se implementarán interfaces gráficas que le permitirán al usuario interactuar con el programa. Por último también se agregara una base de datos que permitirá al sistema persistir a los asociados, así como también gestionar la alta y baja de los mismos.

Para esta nueva etapa aplicaremos diversos conceptos, tales como la programación concurrente o la implementación de patrones vistos durante la cursada, entre otros:

- MVC
- Observer/observable
- State
- Singleton
- DAO
- DTO

Con todo lo anteriormente mencionado y tomando en cuenta lo descrito en la primer parte del trabajo podremos simular la gestión de una clínica mediante el uso de un sistema desarrollado Java.

# Clases e interfaces

## Asociados:

La clase Asociado representa a una persona capaz de solicitar atención o traslados a una ambulancia. Implementa Runnable para ejecutarse en un hilo independiente, permitiendo simular múltiples asociados actuando en simultaneo. El método run() define su comportamiento: realiza solicitudes aleatorias hasta agotar su cantidad de solicitudes o hasta que la simulación se detenga. La clase mantiene datos personales heredados de Persona, una referencia a la ambulancia compartida y validaciones mediante asserts.

Atributos importantes:

- numsolicitudes: Cantidad total de solicitudes que el asociado podrá realizar durante la simulación.
- ambulancia: Referencia a la ambulancia compartida por todos los hilos, el asociado llama a sus métodos para pedir servicio.
- simulacionActiva: Bandera compartida por todos los asociados que indica si la simulación sigue en ejecución.

## Operarios:

Operario representa a un empleado encargado de solicitar mantenimiento para la ambulancia dentro de la simulación.

Corre en un hilo independiente (porque implementa Runnable), el cual es creado cuando el usuario toca el botón “solicitar mantenimiento” de la interfaz gráfica.

Atributos importantes:

- ambulancia: Referencia a la ambulancia sobre la que trabajará. Es obligatoria, porque sin ambulancia el operario no tiene sentido. Por eso en el setter hay un assert ambulancia != null.
- simulacionActiva: Dice si la simulación está activa o debe detenerse.

# Ambulancia:

La clase Ambulancia representa al recurso compartido dentro de la simulación, el cual es utilizado tanto por los Asociados como por los Operarios. También se extiende de Observable, por lo tanto notifica los cambios a las vistas cada vez que ocurre un evento. Es un monitor sincronizado ya que gestiona los hilos concurrentes para que accedan de a uno a algunos de sus métodos.

Los atributos más importantes de esta clase son:

- AmbulancaState: es el estado actual de la ambulancia.
- EstadoString: Texto que representa el estado para mostrarlo a la vista.
- Asociado: Paciente que está siendo trasladado o atendido.
- disponible, estaMantenimiento y estaRegresando: son tres atributos booleanos los cuales indican si la ambulancia esta disponible, en mantenimiento o regresando. (condiciones para que luego el hilo pueda verificar si puede trabajar con el recurso compartido o no mediante la sentencia while)

# ModeloSimulacion:

La clase ModeloSimulacion pertenece a la capa de modelo del sistema y tiene como objetivo orquestar el inicio y control básico de una simulación relacionada con asociados, una ambulancia y un operario. Su función es inicializar los hilos involucrados en la simulación, así como también implementa la lógica para finalizar la simulación.

```
public void inicia(int cantsolicitudes, int cantAsociados, ArrayList<Asociado> asociados) {
    int i=0;
    Iterator<Asociado> iterator = asociados.iterator(); // pido el iterator
    while (iterator.hasNext() && i<cantAsociados) {// mientras haya siguiente
        Asociado asociado = iterator.next();    // obtengo el siguiente
        Thread hilo = new Thread(asociado);
        asociado.setSimulacionActiva(true);
        asociado.setNumsolicitudes(cantsolicitudes);
        asociadosenHilos.add(asociado);
        hilo.start();
        i++;
    }
    Thread hiloRetorno = new Thread(retorno);
    hiloRetorno.start();
```

***"inicia(int cantSolicitudes ,int cantAsociados, ArrayList<Asociado> asociados)"***

El método activa la bandera que nos indica si la simulación está activa(simulacionActiva) y lanza el método start() de los hilos del arraylist pasado

como parámetro. El método “finalizar()” por su parte pondrá la condición simulacionActiva en false, lo que no detiene los hilos, pero si la simulación.

# IState:

Para la implementación del patrón State creamos la interfaz IState, que modela el comportamiento de los estados y también se desarrollaron clases para cada estado posible, que implementaran la interfaz IState.

A continuación, se adjunta un segmento del código mostrando el contenido de la interfaz en cuestión

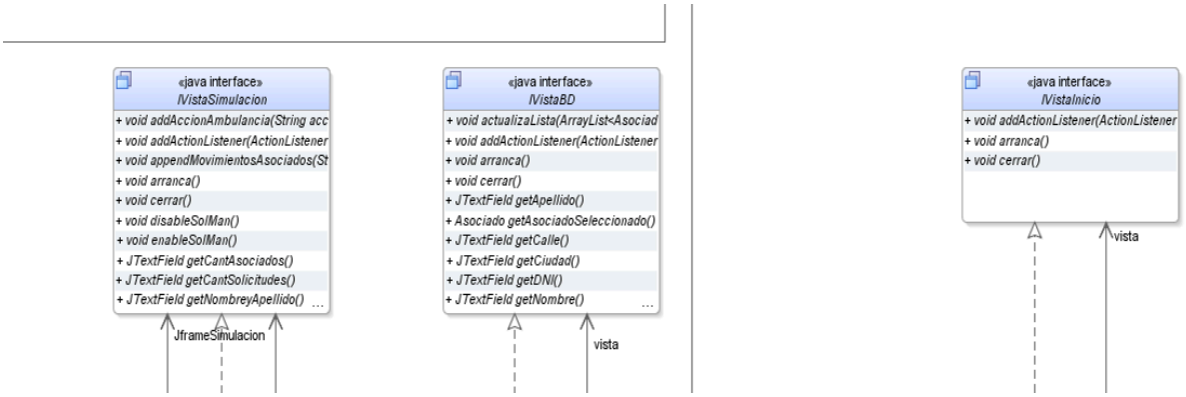
```
public interface IState {  10 usages  6 implementations  Mia Casanovas Di Sera *


    /**
     * Cambia, si es posible, el estado actual al recibir una peticion de atencion de un paciente
     */
    public void pacienteSolicitaAtencion(Asociado asociado) throws InterruptedException;  2 usages  6 implementations  new *

    /**
     * Cambia, si es posible, el estado actual al recibir una peticion de traslado de un paciente
     */
    public void pacienteSolicitaTraslado(Asociado asociado) throws InterruptedException;  no usages  6 implementations  new *

    /**
     * Cambia, si es posible, el estado actual al recibir un retorno automatico
     */
    public void retornoAutomatico() throws InterruptedException;  1 usage  6 implementations  new *

    /**
     * Cambia, si es posible, el estado actual al recibir una solicitud de mantenimiento por parte de un operario
     */
    public void solicitudMantenimiento() throws InterruptedException;  1 usage  6 implementations  new *

}
```

# IVistas:


En el patrón MVC, las interfaces de la Vista funcionan como un contrato que define qué operaciones puede realizar u ofrecer una Vista, sin importar cómo esté implementada gráficamente. Su función es desacoplar el controlador de la vista concreta, permitiendo que ambos puedan evolucionar o cambiarse sin afectar al otro. Utilizamos tres vistas: IVistaDB, IVistaInicio y IVistaSimulacion las cuales

contienen los métodos que deben implementar las ventanas, a continuación las interfaces que aplica cada ventana y sus respectivos controladores:

- IVistaBD => JFrameBD => ControladorBD.
- IVistaInicio  => JFrameInicio => ControladorInicio.
- IVistaSimulacion => JFrameSimulacion =>ControladorSimulacion.



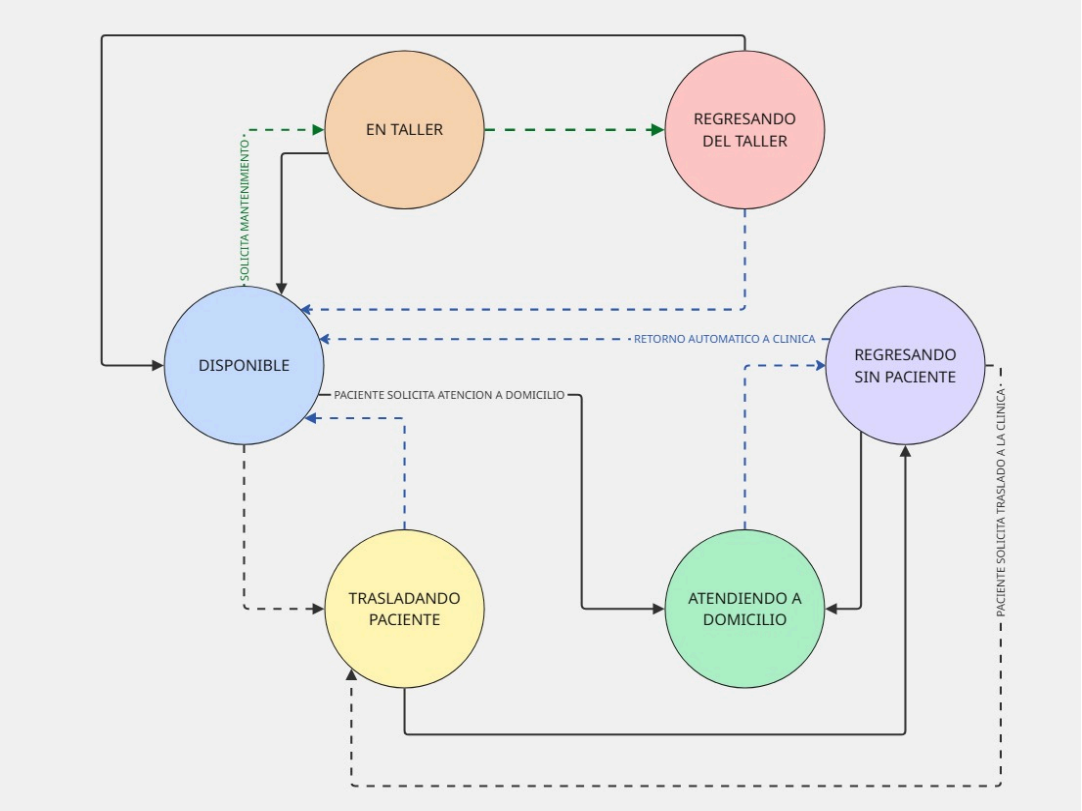
# Patrones

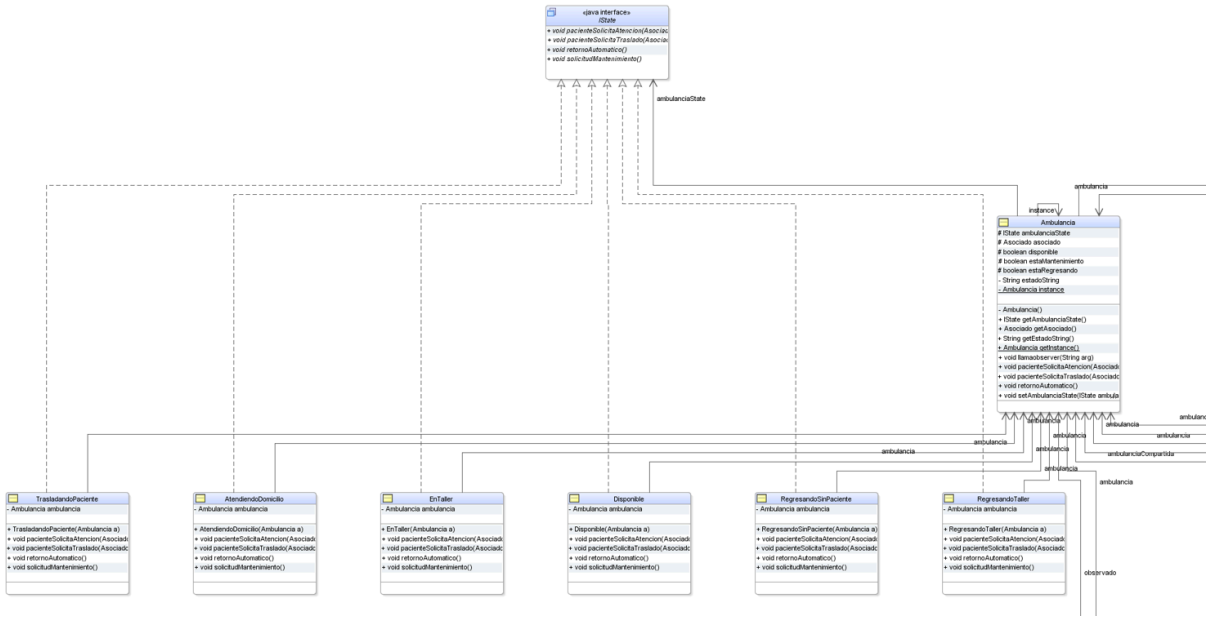
## State:

La clase ambulancia implementa el Patrón de Diseño State, permitiendo que su comportamiento cambie dinámicamente según el estado operativo en el que se encuentre.

```java
public class Ambulancia extends Observable {  Mia Casanovas Di Sera +2 *
    /** estado actual de la ambulancia*/
    protected IState ambulanciaState; //PatronState, el estado actual  8 usages
    /** string de estado actual de la ambulancia*/
    private String estadoString; //dto simple para notificar a la vista  2 usages
    /** atributo utilizado para el singleton de la ambulancia*/
    private static Ambulancia instance;  3 usages
    /** {@link Asociado } */
    protected Asociado asociado;        // El recurso "ocupado" (si no es null, está ocupada)
    /** flag que me dice si la ambulancia esta disponible o no(true== esta disponible)*/
    protected boolean disponible;      // Flag de estado (manejado por el State)  7 usages
    /** flag que me dice si la ambulancia esta en mantenimiento o no(true== esta en mantenimiento)*/
    protected boolean estaMantenimiento;  // Flag de estado (manejado por el State)  5 usages
    /** flag que me dice si la ambulancia esta en regresando o no(true== esta regresando)*/
    protected boolean estaRegresando;  2 usages
```

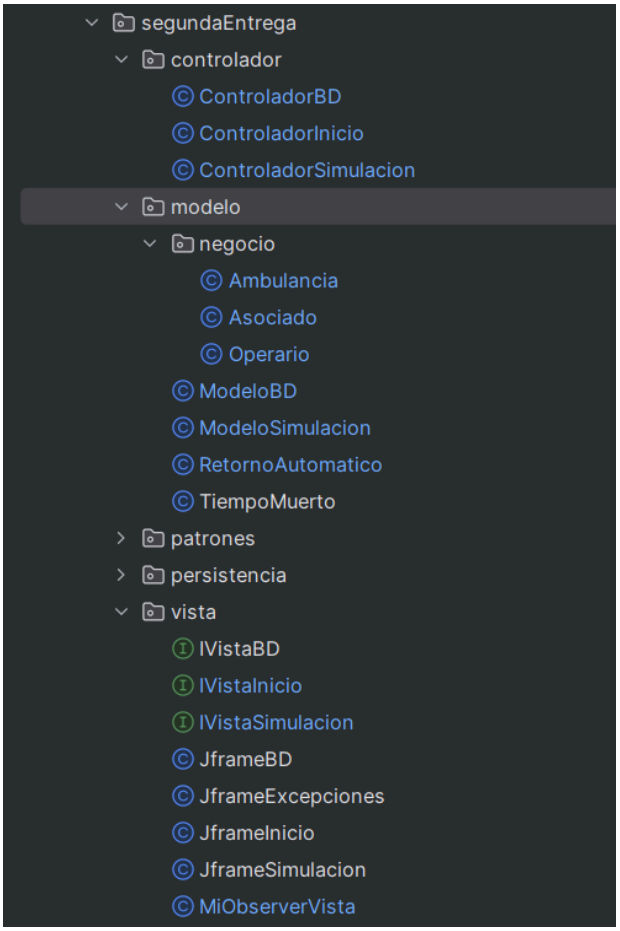
El atributo IState representa el estado actual de la ambulancia con 6 variantes. La ambulancia comienza en estado Disponible y cada estado recibe una referencia a la ambulancia para poder actualizar su contexto. En la siguiente imagen se puede ver como son y cómo varían los diferentes pasajes de estados.





# MVC:

El patrón MVC nos permite organizar el código y separar responsabilidades. Nuestro modelo contiene las clases Ambulancia, Operario y Asociado que son las que representan los datos y la lógica de negocio en esta segunda parte del trabajo. La clase Ambulancia forma parte central del modelo y representa el recurso operativo encargado de atender solicitudes de asociados dentro del sistema.




Por otro lado en la carpeta controlador, está el ControladorBD que recibe una vista que implementa la interfaz IVistaBD y un objeto del modelo Sistema.
Luego, se registra como ActionListener de la vista, permitiendo escuchar y manejar las acciones del usuario.
El ControladorBD estará a cargo de la alta y baja de asociados mediante los métodos altaAsociado() y bajaAsociado(). El primero creará un asociado con los datos ingresados por la ventana y la baja se dará seleccionando un asociado de la lista de asociados e interactuando con el botón.

Otro controlador utilizado es el ControladorInicio, el cual tiene como función principal coordinar la transición desde la pantalla de inicio hacia la ventana de simulación(JFrameSimulacion) o la ventana de la BD.

Por último tenemos el ControladorSimulacion que recibe como parámetros una vista que implementa la interfaz IVistaSimulacion y un objeto ModeloSimulacion, que representa el modelo encargado de gestionar la ambulancia, las solicitudes y el ciclo de la simulación. Además, se cuenta con una clase llamada MiObserverVista, la cual instancia a un objeto que implementa la interfaz Observer, mediante el cual se va a actualizar la ventana de la simulación en tiempo real a medida que la ambulancia(objeto observable) cambie de estado.

La Vista contiene todas las ventanas utilizadas por nuestro programa, entre ellas se encuentran las siguientes ventanas:

- JFrameSimulación: Esta ventana está dividida en diferentes secciones. Primero, se encuentra el panel de operario, el cual permite cargar el nombre del operario y solicitar mantenimiento mediante un botón dedicado (se activa solo cuando la simulación está activa). También está el panel ambulancia, que muestra el estado actual de la misma, que como explicado anteriormente, es actualizado por MiObserverVista en tiempo real cuando el estado cambia. Por otro lado, está el panel simulacion, que contiene campos para ingresar la cantidad de solicitudes y la cantidad de asociados, además de botones para iniciar y finalizar la simulación. Y por último, está el panel de notificaciones de asociados, que presenta un área de texto donde se listan las acciones y notificaciones generadas durante la simulación.

- JFrameDB: Esta ventana contiene los campos de entrada necesarios para registrar o modificar un asociado: nombre, apellido, DNI, teléfono, ciudad, calle y número de la calle. A la derecha del panel se encuentran los botones para la alta y baja de asociados, otro boton que permite cargar datos de prueba y uno para borrar todo el contenido de la base de datos. En la parte inferior se encuentra el listado de asociados en él que se seleccionará el asociado que se quiera dar de baja.

- JFrameInicio: Esta ventana inicial está compuesta por un encabezado con mensajes de bienvenida y un panel de botones que permite seleccionar entre el módulo base de datos o el módulo de simulación.

También contamos con un JframeExcepciones, justamente para mostrar las excepciones que puedan surgir.

Además desarrollamos la clase TiempoMuerto, cuya función es simular los tiempos entre las solicitudes.

```java
public class TiempoMuerto {  4 usages

    public static void esperar() throws InterruptedException{  3 usages
        Random random = new Random();
        int ms = 2000 + random.nextInt( bound: 3000); //
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            throw new InterruptedException("Error en sleep");
        }
    }
}
```

a continuación dejamos el diagrama UML del patrón MVC:

# Observer/Observable:

Se utilizó este patrón para notificar el cambio del estado de la ambulancia, lo que nos mostrará su estado actualizado en todo momento durante la simulación.

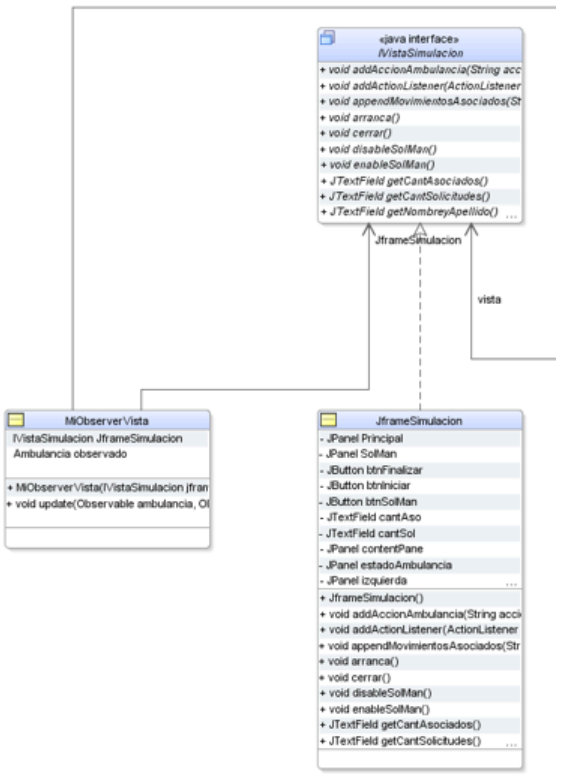
La clase Ambulancia se extiende de Observable convirtiendo a la ambulancia en un objeto observado, capaz de registrar observadores, detectar cambios y comunicar eventos relevantes. La ambulancia no necesita conocer quiénes son sus observadores ni qué hacen con la información y solo se limita a notificar.

Cada vez que la ambulancia produce un evento relevante se ejecuta el siguiente mecanismo estándar del patrón:

- this.setChanged(); => Marca internamente que el objeto experimentó un cambio.
- this.notifyObservers(mensaje); => envía un mensaje o descripción del evento a todos los observadores registrados y a su vez quita la marca de modificado.

```
public void llamaobserver(String arg){  10 usages   new *
    this.setChanged();
    this.notifyObservers(arg);
}
```

Por el lado del observer MiObserverVista implementa la interfaz observer, por lo cual implementa sus métodos. Su función principal es recibir las notificaciones provenientes de una instancia de Ambulancia(observable) y reflejar dichos cambios en la interfaz gráfica representada por IVistaSimulacion.

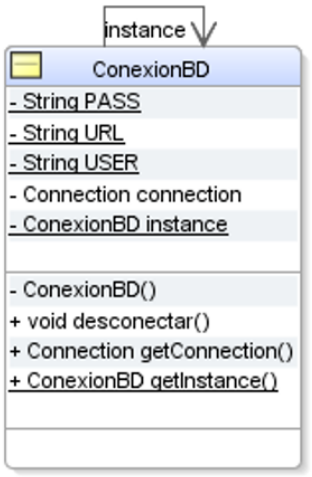



En el constructor, el observador se asocia directamente con la instancia de Ambulancia y queda registrado para recibir actualizaciones sin intervenir en la lógica interna del observable.

Dentro del método update() se verifica que la notificación provenga del objeto observado para ignorar notificaciones de otros observables manteniendo coherencia en la interacción.

```java
public class MiObserverVista implements Observer  no usages  gabynotebook +1 *
{
    IVistaSimulacion JframeSimulacion;      // COMPOSICION con la vista  3 usages
    Ambulancia observado;  4 usages



    public MiObserverVista(IVistaSimulacion jframeSimulacion,Ambulancia ambulancia) {
        JframeSimulacion = jframeSimulacion;
        observado = ambulancia;
        observado.addObserver( o: this);
    }


    @Override  gabynotebook +1
    public void update(Observable ambulancia, Object mensaje) {
        if (ambulancia==observado) {
            String estado = observado.getEstadoString();
            JframeSimulacion.setEstadoAmbulancia(estado);
            String mensajeConsola= (String) mensaje;
            JframeSimulacion.appendMovimientosAsociados(mensajeConsola);
        }
        else
            throw new IllegalArgumentException();
    }
}
```

# Singleton:




Utilizamos este patrón a la hora de crear nuestra base de datos. La clase ConexionDB lo aplica para que haya una sola instancia de la clase.

A través del método estático getInstance(), la aplicación obtiene la instancia única del Singleton. Si la instancia aún no fue creada, el método la inicializa, de lo contrario, utiliza la existente.

```
public class ConexionBD {  9 usages  Mia Casanovas Di Sera
    private static ConexionBD instance;  3 usages
    private Connection connection;  7 usages

    private static final String URL = "jdbc:mysql://localhost:3306/Grupo_x";  1 usage
    private static final String USER = "progra_c";  1 usage
    private static final String PASS = "progra_c";  1 usage

    private ConexionBD() {  1 usage  Mia Casanovas Di Sera
        try {
            Class.forName( className: "com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static ConexionBD getInstance() {  5 usages  Mia Casanovas Di Sera
        if (instance == null) {
            instance = new ConexionBD();
        }
        return instance;
    }
```

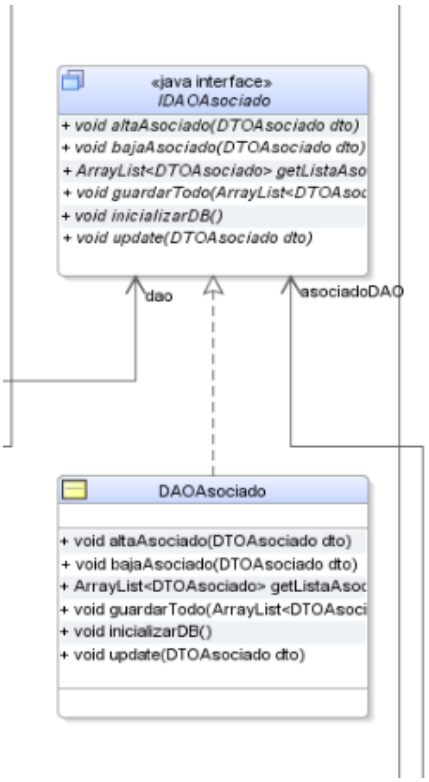
# DAO:

La clase DAOAsociado implementa el patrón DAO, cuyo propósito es encapsular todo el acceso a la base de datos, evitando que las capas superiores del sistema interactúen directamente con SQL o con objetos de conexión.

A continuación se podrá ver un fragmento de código de la implementación del patrón DAO, donde se observa una de sus funcionalidades al interactuar con una de las ventanas, la cual permite agregar(alta) a un asociado a la base de datos.

Cada método del DAO representa una operación CRUD:

- altaAsociado() => INSERT
- update() => UPDATE
- bajaAsociado() => DELETE
- getListaAsociado =>SELECT

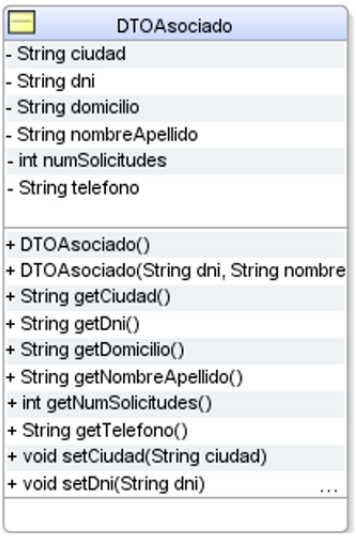



```java
@Override
public void altaAsociado(DTOAsociado dto) throws Exception {
    String sql = "INSERT INTO asociados (dni, nombre, domicilio, ciudad, telefono, numSolicitudes) VALUES (?, ?, ?, ?, ?, ?)";
    Connection con = null;
    PreparedStatement ps = null;

    try {
        con = ConexionBD.getInstance().getConnection();
        ps = con.prepareStatement(sql);

        // Mapeo de atributos del DTO a la consulta SQL
        ps.setString(1, dto.getDni());
        ps.setString(2, dto.getNombreApellido());
        ps.setString(3, dto.getDomicilio());
        ps.setString(4, dto.getCiudad());
        ps.setString(5, dto.getTelefono());
        ps.setInt(6, dto.getNumSolicitudes());

        ps.executeUpdate();
    } catch (SQLException e) {
        throw new Exception("Error al insertar asociado: " + e.getMessage());
    } finally {
        if (ps != null) ps.close();
    }
}
```

# DTO:

En cada operación, el DAO utiliza la clase DTOAsociado, este mismo permite trasladar datos entre capas, sin exponer directamente las entidades internas o estructuras complejas.

El constructor vacío permite crear un DTO e ir cargando sus valores de manera progresiva, en cambio el que tiene parámetros permite transportar un conjunto coherente de datos en una sola operación.




```java
public class DTOAsociado implements Serializable
{
    private String dni;
    private String nombreApellido;
    private String domicilio;
    private String ciudad;
    private String telefono;
    private int numSolicitudes;

    public DTOAsociado(){}

    public DTOAsociado(String dni, String nombreApellido, String domicilio, String ciudad, String telefono, int numSolicitudes) {
        this.dni = dni;
        this.nombreApellido = nombreApellido;
        this.domicilio = domicilio;
        this.ciudad = ciudad;
        this.telefono = telefono;
        this.numSolicitudes = numSolicitudes;
    }
```