UNIVERSIDAD NACIONAL de MAR DEL PLATA

# Trabajo Práctico Grupal 2025

## Segunda Etapa

- **Asignatura:** Programación C
- **Integrantes del grupo:**
  - Aldana Juan
  - Distefano Gian Luca
  - Giordano Valentino
  - Proia Agustin
  - Yanosky Santiago
- **Fecha de Entrega:** 16/11/2025

## Reestructuración del Proyecto

En la primera instancia de esta segunda etapa del TPG, nos concentramos en reestructurar los distintos módulos del proyecto, siguiendo las especificaciones dadas por la cátedra en el enunciado de esta segunda parte.
Por otro lado y enfocándonos más en la parte programable, en esta segunda parte nos vimos con la necesidad de aplicar una interfaz gráfica, de tal manera que la experiencia para el usuario sea óptima, con una interfaz gráfica clara, comprensible y fácil de usar, esta interfaz gráfica se implementó utilizando a su vez el patrón **MVC**, para separar de manera clara las distintas capas del programa, más adelante se va a desarrollar en profundidad el uso de este patrón.
Relacionando con la primera parte, a nuestro sistema original se le añadieron:

- Un nuevo módulo (**Ambulancia**).
- La conexión a la Base de Datos (**Persistencia**).
- Los participantes de la simulación (**Asociados y un Operario**).
- Los observadores de los nuevos participantes.

Todos estos nuevos participantes del modelo son grandes protagonistas de los distintos patrones que aplicamos durante el proyecto.

## Patrón Facade

Siguiendo la implementación del patrón **Facade** en la primera entrega, en esta entrega agregamos el Módulo Simulación al sistema, que va a ser el encargado de llevar a cabo todas las funciones relacionadas a la simulación.
Este módulo es el encargado de iniciar los distintos hilos de los **Asociados** y del **Operario** en caso de que el usuario solicite enviar la ambulancia al taller. También es la encargada en caso de que se quiera finalizar la simulación y/o se termine debido a que ya no haya más solicitudes.

## Patron MVC - Model View Controller

Este patrón lo aplicamos para separar las distintas capas del programa, el Modelo (**Sistema**), la Vista (**VistaConfig** y **VistaSimulacion**) y el **Controlador**. La aplicación de este patrón fue para poder comunicar a la Vista con el Modelo sin la necesidad de que estos 2 se conozcan, esto a través del Controlador, quien conoce a los 2.
(De esta manera disminuimos su *Acoplamiento*).
El **Controlador** de nuestro programa se encarga de recibir acciones realizadas por el usuario a través de la **Vista** y según esas acciones comunicarse y realizar cambios en el **Modelo**. Esto incluye al manejo de los cambios realizados al **Modelo** durante la simulación, como las solicitudes de los **Asociados** a la **Ambulancia** y la conexión entre las interacciones del usuario que realiza utilizando la **Vista** con la operación correspondiente que se hace en el **Modelo**.
La parte clave de este patrón es que sólo el **Controlador** conozca a ambos (**Modelo** y **Vista**). De esta manera, el **Controlador** es capaz de "escuchar" a las acciones del usuario realizadas en la **Vista** (como la interacción con los botones y los diferentes campos de texto) sin que esta se "entere". (La **Vista** simplemente envía los mensajes pero no sabe quién los recibe, de esta manera es más independiente). Esto se logra a través del método **setActionListener** que debe implementar la vista pero que luego es utilizado por el **Controlador** para asignarse a sí mismo como un "Listener".

A continuación vamos a ver algunas partes del código implementado en la clase **Controlador** y la clase **VistaSimulacion**:

```
public Controlador(IVista vistaSim, VistaConfig vistaConfig, Sistema modelo) {  1 usage  Valentino *
    super();
    this.modelo = modelo;
    this.vistaSim = vistaSim;
    this.modelo.crearObservadores( controlador: this);
    vistaSim.setActionListener(this);
    vistaSim.setWindowListener(this);
    this.vistaConfig = vistaConfig;
    vistaConfig.setActionListener(this);
    vistaSim.setVisible(false);
    vistaConfig.setVisible(true);
}
```

```java
@Override  Valentino
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals("Iniciar Simulacion")) {
        int numAsociados = vistaConfig.getNumAsociados();
        int numSolicitudes = vistaConfig.getNumSolicitudes();
        modelo.configurarSimulacion(numAsociados, numSolicitudes);
        vistaConfig.setVisible(false);
        vistaSim.setVisible(true);
        vistaSim.setearListas(modelo.getAsociados());
        vistaSim.setearOpearios(modelo.getOperario());
    } else if (command.equals("Finalizar")) {
        vistaSim.desactivarTaller();
        modelo.finalizarSimulacion();
    } else if(command.equals("Mantenimiento")){
        vistaSim.desactivarTaller(); //Para evitar multiples hilos de mantenimiento.
        modelo.mandarAMantenimiento();
    } else if(command.equals("Reiniciar")){

        vistaSim.iniciarSimulacion();
        vistaSim.limpiarTextAreas();
        vistaSim.limpiarCampos();
        vistaSim.setearListas(modelo.getAsociados());
        vistaSim.setearOpearios(modelo.getOperario());
```

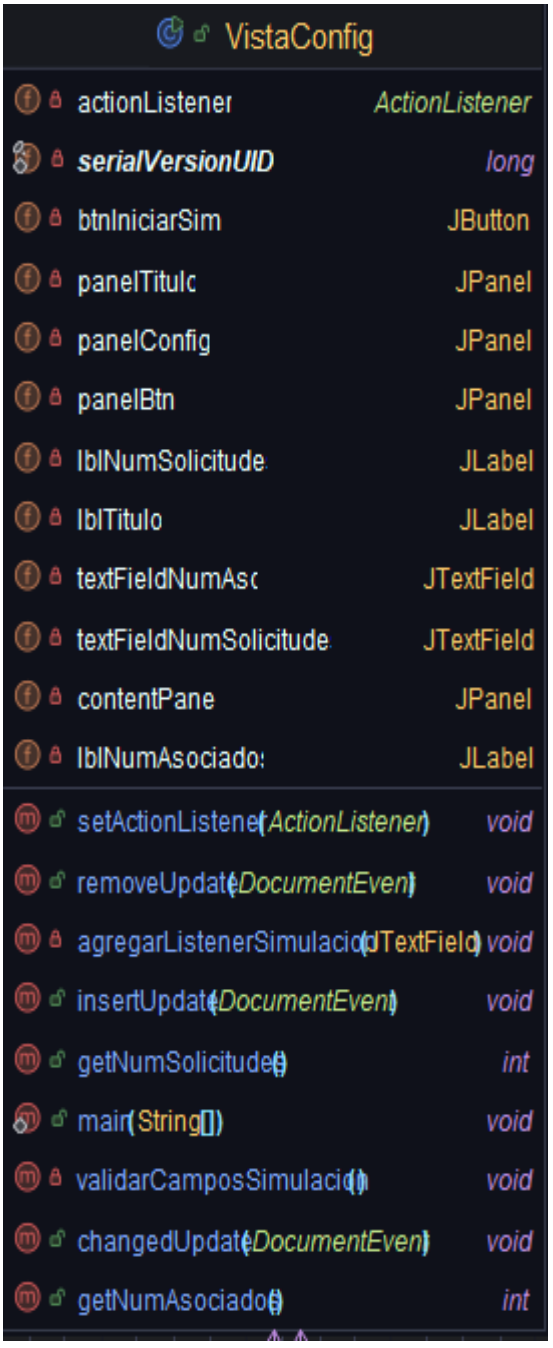
```java
@Override  1 usage  Valentino
public void setActionListener(ActionListener actionListener) {
    this.actionListener = actionListener;
    this.btnAlta.addActionListener(this.actionListener);
    this.btnBaja.addActionListener(this.actionListener);
    this.btnModificacion.addActionListener(this.actionListener);
    this.btnReiniciarSim.addActionListener(this.actionListener);
    this.btnFinalizarSim.addActionListener(this.actionListener);
    this.btnReiniciarDB.addActionListener(this.actionListener);
    this.btnMantenimiento.addActionListener(this.actionListener);
}
```

En la primer imagen tenemos el constructor del **Controlador** donde se inicializan los *atributos*, se establece la relación entre el **Controlador** y la **Vista** y se inicia la ventana de configuración (de la Simulación), esta es la ventana donde el usuario debe ingresar la *cantidad de asociados* que participan en la Simulación y la *cantidad de solicitudes* de cada uno.
En la segunda imagen observamos el método más importante del controlador, el **actionPerformed**, este se encarga de recibir las acciones que realiza el usuario y según el tipo de acción que reciba el Controlador hace una cosa u otra.
En la tercera imagen observamos una parte del código de la **Vista**, donde se le asigna a los botones qué controlador es quien los va a escuchar, este es el primer método que se llama en el constructor del **Controlador**, *vistaSim.setActionListener(this)* donde el **Controlador** se asigna como el *ActionListener*.

Los cambios que se producen en el **Modelo**, no se lo comunica el **Modelo** directamente al **Controlador** sino que hace uso de un "**Observador"**, aquí es donde entra otro patrón aplicado en el proyecto, el patrón **Observer-Observable**.

Este diagrama UML ilustra parcialmente la implementación del patrón **MVC**. Se observa cómo el **Controlador** tiene conocimiento de la **Vista** y el **Modelo** para coordinarlos.

Por el contrario, la **Vista** solo detecta que un *listener* (que en este caso es el Controlador) reacciona a sus eventos, como la pulsación de un botón o la entrada de texto. Nota: Los diagramas UML completos del Modelo y de la Vista de Simulación se han omitido por brevedad, pero están disponibles para consulta en el repositorio del proyecto.




## Patron Observer-Observable

Este patrón lo utilizamos para poder reflejar en la **Vista** los distintos cambios que producen los **Asociados**, el **Operario** y los distintos cambios en el estado de la **Ambulancia**.

- **Asociados**: Los Asociados, durante la simulación, cada vez que realizan algún tipo de solicitud a la Ambulancia les notifican a sus Observadores que han *realizado una solicitud*, ya sea "Atencion a Domicilio" o "Traslado a clínica", cuando el Observador recibe la notificación, se encarga de comunicarle al Controlador que hubo un cambio en el modelo (un Asociado realizó una solicitud), el Controlador, conocedor de la Vista, le comunica a la Vista que actualice la pantalla y muestre por esta misma el nombre del asociado y la solicitud que este realizó.
- **Operario**: Cada vez que el usuario solicita mantenimiento el Controlador le comunica al Modelo que debe crear un nuevo hilo de ejecución de tipo **ThreadOperario**, este Thread se encarga de solicitar mantenimiento a la Ambulancia. El Operario al igual que los Asociados posee un Observador, el momento en que el operario solicita con éxito el mantenimiento a la ambulancia le notifica a su Observador que ha enviado con éxito a la ambulancia al taller, luego este Observador se comunica con el Controlador para que le avise a la Vista que indique por pantalla que la ambulancia se encuentra en el taller, de igual manera se se repetirá este proceso cuando la Ambulancia abandone el taller.
- **Ambulancia**: Esta clase es la mejor a la hora de aplicar el patrón Observer-Observable, ya que la Ambulancia durante la simulación está constantemente cambiando de estado y, por ende, comunicándoselo a su Observador que, a su vez, lo comunica al Controlador para que el controlador le avise a la Vista que la ambulancia sufrió un cambio de estado y, por consiguiente, debe cambiar el estado de la ambulancia reflejado en la vista.

Para ejemplificar este patrón vamos a mostrar un poco de código de la **Ambulancia**, ya que la aplicación de este patrón en las 3 clases es muy parecido y su diferencia es mínima:

```java
@Override  10 usages  Valentino
public void notificarObservadores(String evento) {
    for (IObservador observador : this.observadores) {
        observador.update( obj: this, evento);
    }
}
```

```java
public void setEstado(StateAmbulancia estado) {  11 usages  JuanEstebanAldana +1
    this.estado = estado;
    setChanged();
    notificarObservadores( evento: "Cambio de estado");
}
```

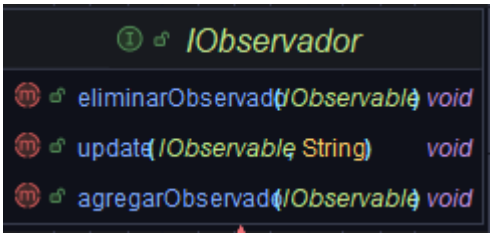
```java
@Override
public void update(IObservable obj, String evento) {

    assert ambulancias.contains(obj)==true : "El objeto observado no es la ambulancia asociada a este observador.";

    Ambulancia ambulancia = (Ambulancia) obj;
    if (controladorVista != null) {
        if (evento.equals("Termino la simulacion"))
            controladorVista.finalizarSim("Simulacion Finalizada", ambulancia.isSimulacion());
        else
            controladorVista.actualizarEstadoAmbulancia(ambulancia.getEstado().getNombre(), ambulancia.isSimulacion());
    }
}
```

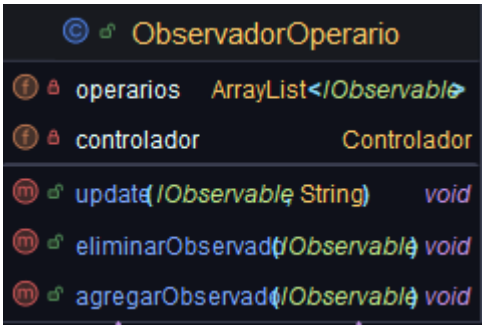
```java
public void actualizarEstadoAmbulancia(String estado, Boolean enSimulacion) {  1 usage  Valentino
    vistaSim.actualizarEstadoAmb("");
    if(estado.equals("Regresando del taller") && enSimulacion){
        vistaSim.activarTaller();
    }

    vistaSim.actualizarEstadoAmb(estado);
}

public void finalizarSim(String estado, Boolean enSimulacion) {  1 usage  Valentino
    if (!enSimulacion){
        System.out.println("Finalizo la simulacion");
        estado = "Simulacion finalizada.";
        vistaSim.finalizarSimulacion();
        vistaSim.actualizarEstadoAmb(estado);
    }
}
```

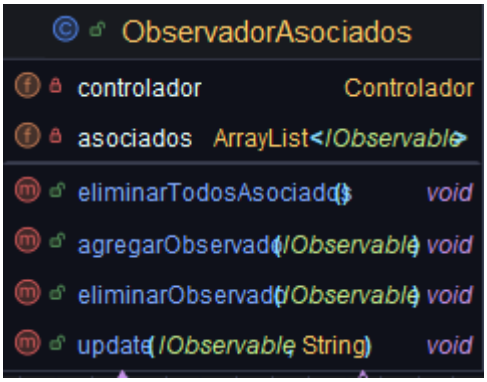
Acá tenemos cuatro imágenes que muestran un poco de la implementación del Observador, tenemos código de la clase **Ambulancia** (Imagen 1 y 2), clase **ObservadorAmbulancia** (Imagen 3) y el **Controlador** (Imagen 4) .

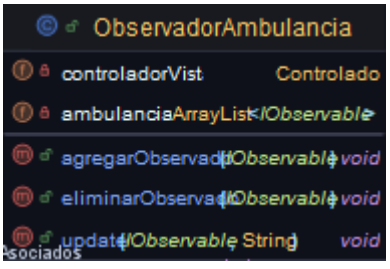
En las primeras dos imágenes observamos los dos métodos que desencadenan la actualización de la vista, la **Ambulancia** en el momento en el que se actualiza su estado (Imagen 1), y la manera que notifica a sus Observadores (Imagen 2), luego sus Observadores en este caso **ObservadorAmbulancia**, según el *tipo de notificación* que reciba (un cambio de estado o una notificación de que se terminaron las solicitudes), va a comunicarle una cosa u otra al **Controlador**, en última instancia el **Controlador** se encarga de avisar los cambios en el modelo a la **Vista** para que esta se actualice mostrándole los cambios al usuario.

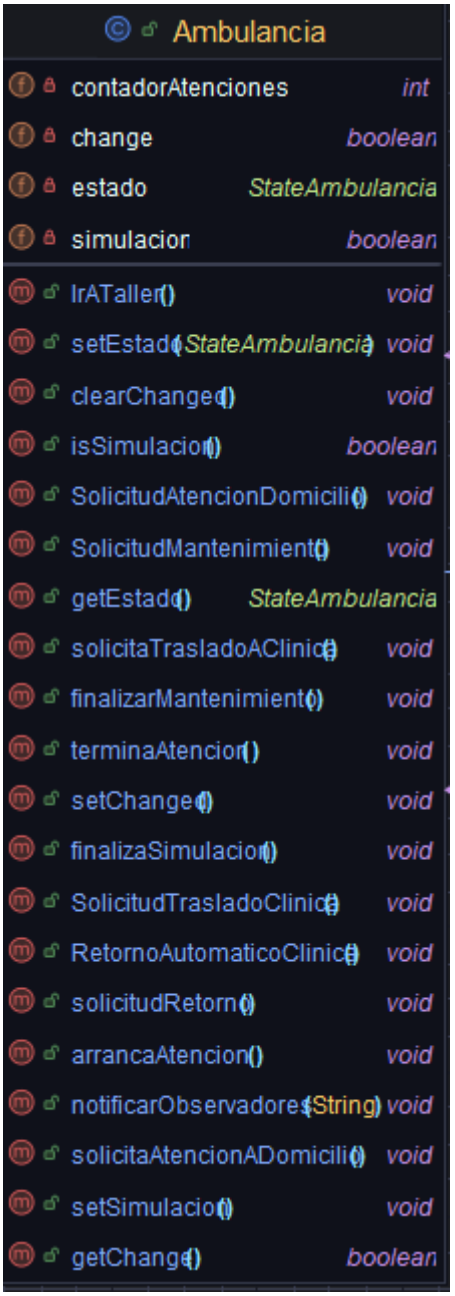
Partes del diagrama UML del patrón **Observer - Observable**:



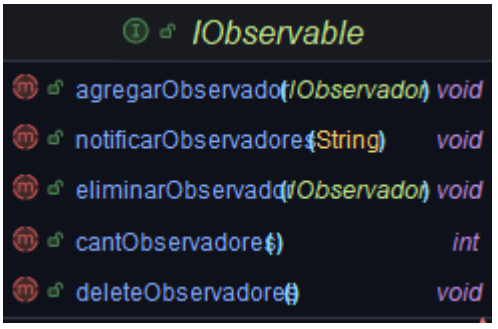



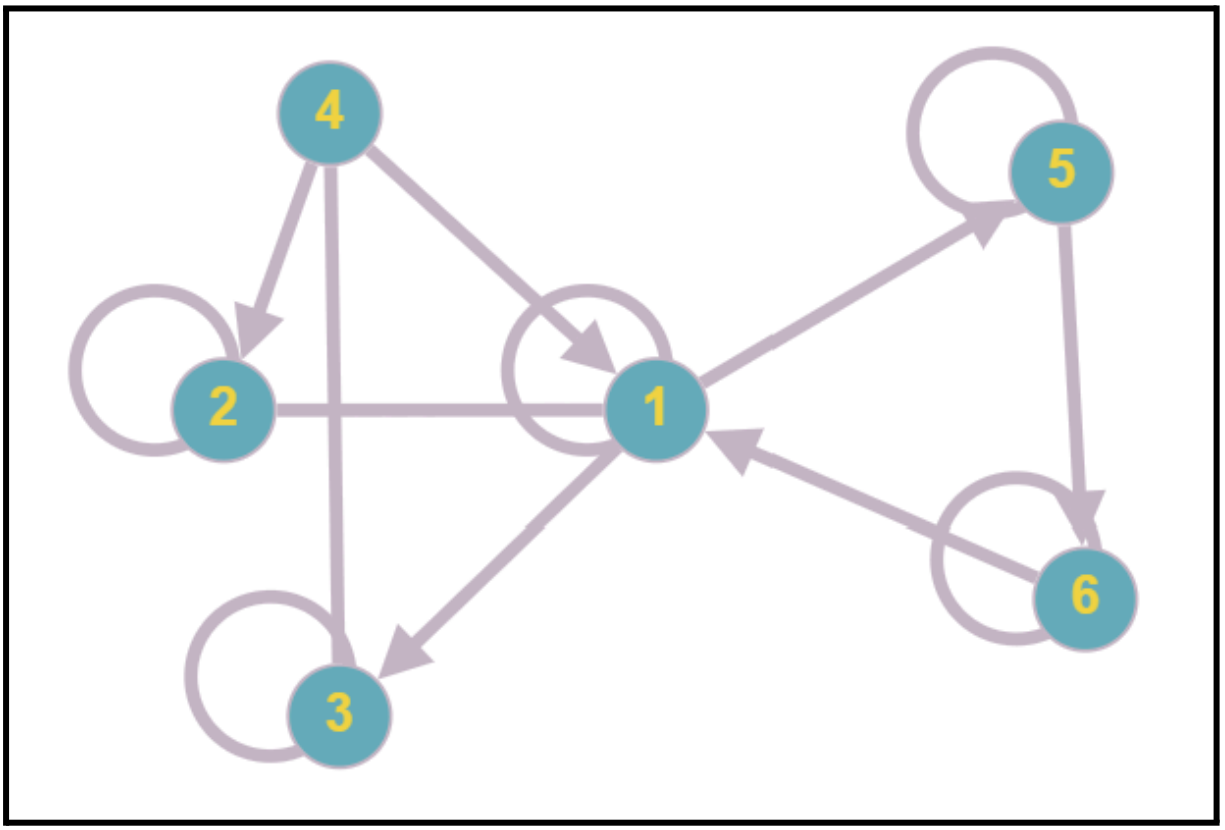


## Patrón State

Aplicamos el patrón **State** para modelar el uso de la ambulancia. Dicho patrón se usa para implementar un objeto que se comporta de distinta manera en base al estado en que se encuentra. En nuestra simulación la **Ambulancia** es un objeto que se comporta de distintas maneras dependiendo de la acción que esté realizando. Para esto se parte de una clase **Ambulancia**, la misma contiene los atributos generales de la Ambulancia y un *atributo estado*, al que se le asigna un objeto que implementa la interfaz ***StateAmbulancia*** . Esta *interfaz*, contiene los métodos capaces de modificar el comportamiento y de realizar cambios de estado. Por ende será el propio estado, dependiendo de la implementación que reciban los métodos en el mismo quien realizará el cambio y/o las acciones pertinentes. Dicha implementación varía en los diferentes estados según sea o no posible realizar el cambio solicitado.

A continuación adjuntamos un grafo que ilustra cuáles estados son o no accesibles desde otros, independientemente de la solicitud que genera dicho cambio. Las líneas que no finalizan con una flecha implican que ambos estados son accesibles desde el otro:



Estados:

1- Disponible

2- Trasladando Paciente

3- Atendiendo a Domicilio

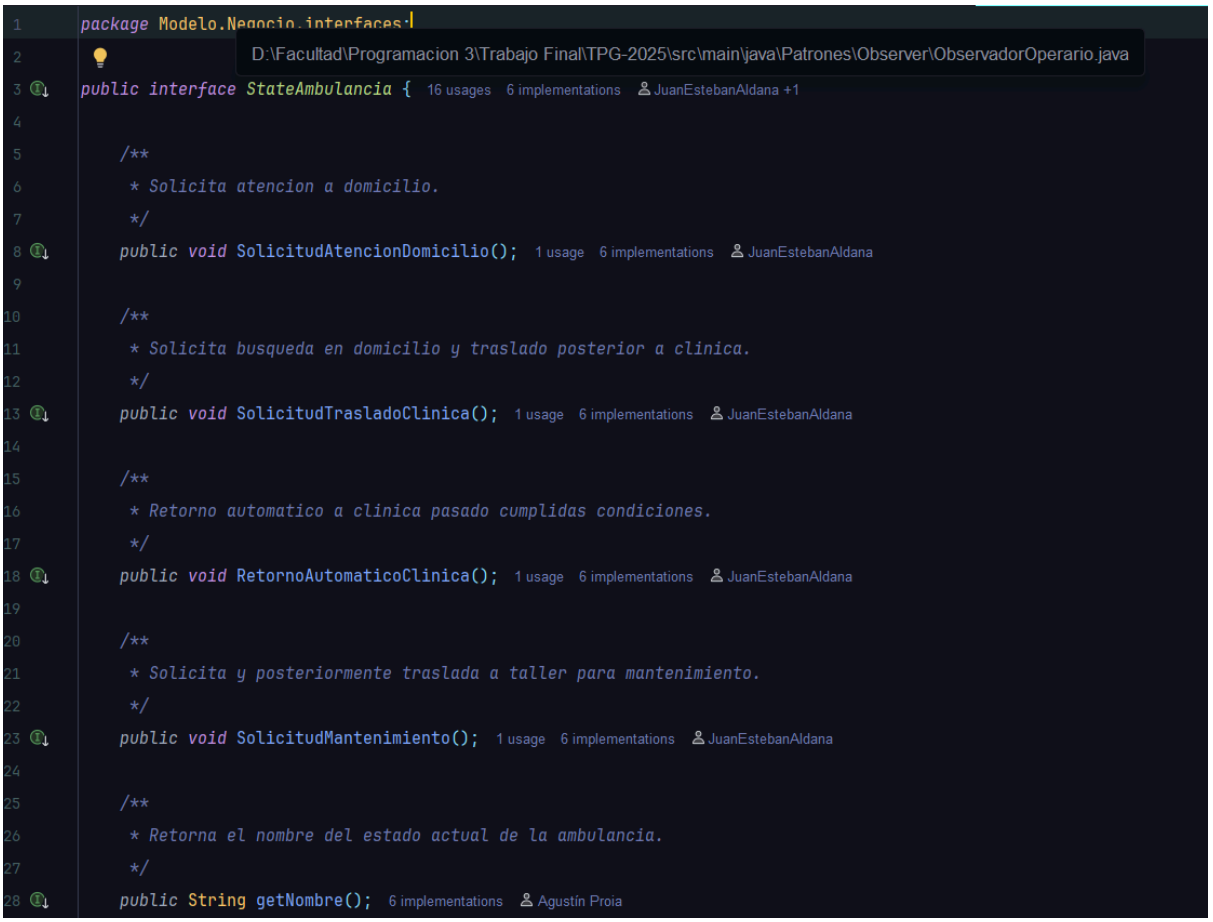
4- Regresando sin Paciente

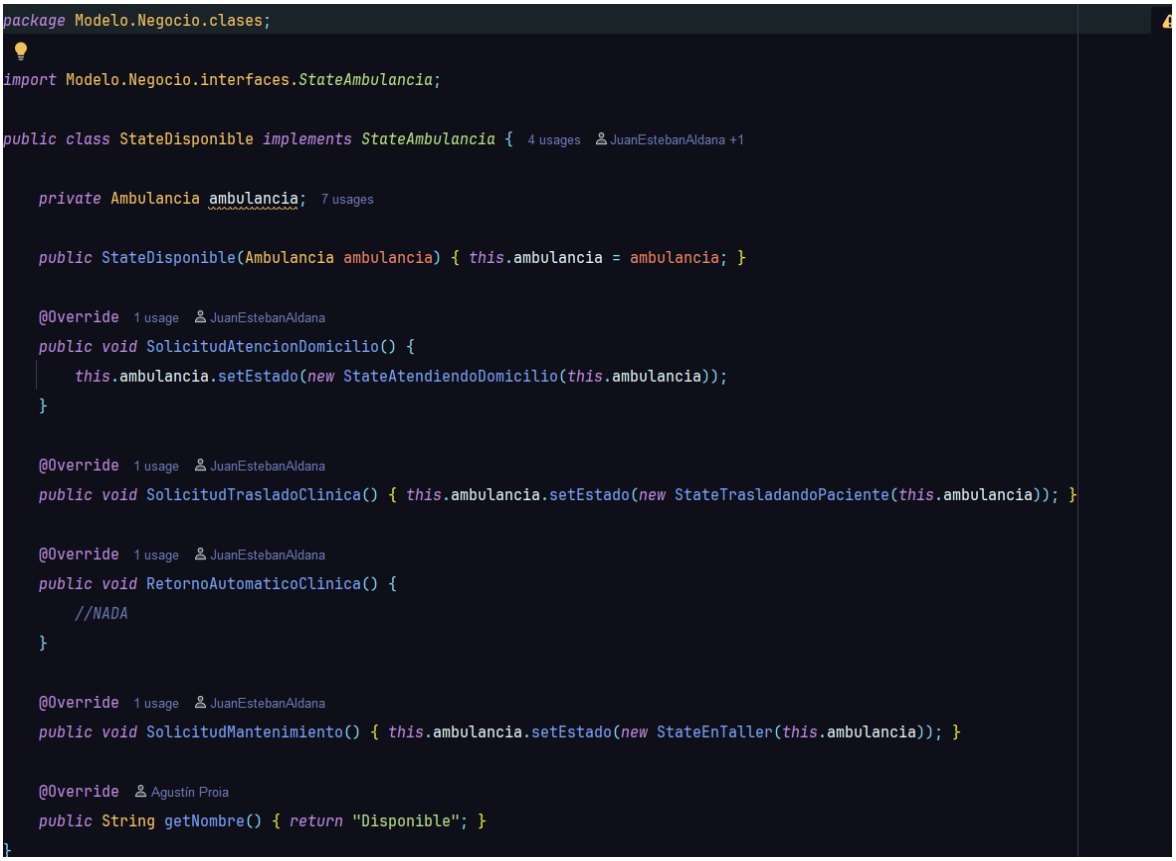
5- En Taller

6- Regresando del Taller

Los **Estados** son Clases concretas, de la cual, como mencionamos la **Ambulancia** tendrá una instancia en forma de una variable de tipo *interfaz* (**StateAmbulancia**). Cuando se realiza la solicitud del cambio de estado, la **Ambulancia** delega esta tarea al propio estado. Implementar el estado como un atributo de la ambulancia obliga a instanciar un nuevo estado cada vez que se realiza un cambio.

A modo ejemplificador añadimos código pertenecientes a la *interfaz* **StateAmbulancia** y a un estado concreto. En el mismo se observa específicamente como no todos los estados son accesibles desde el estado “actual”.

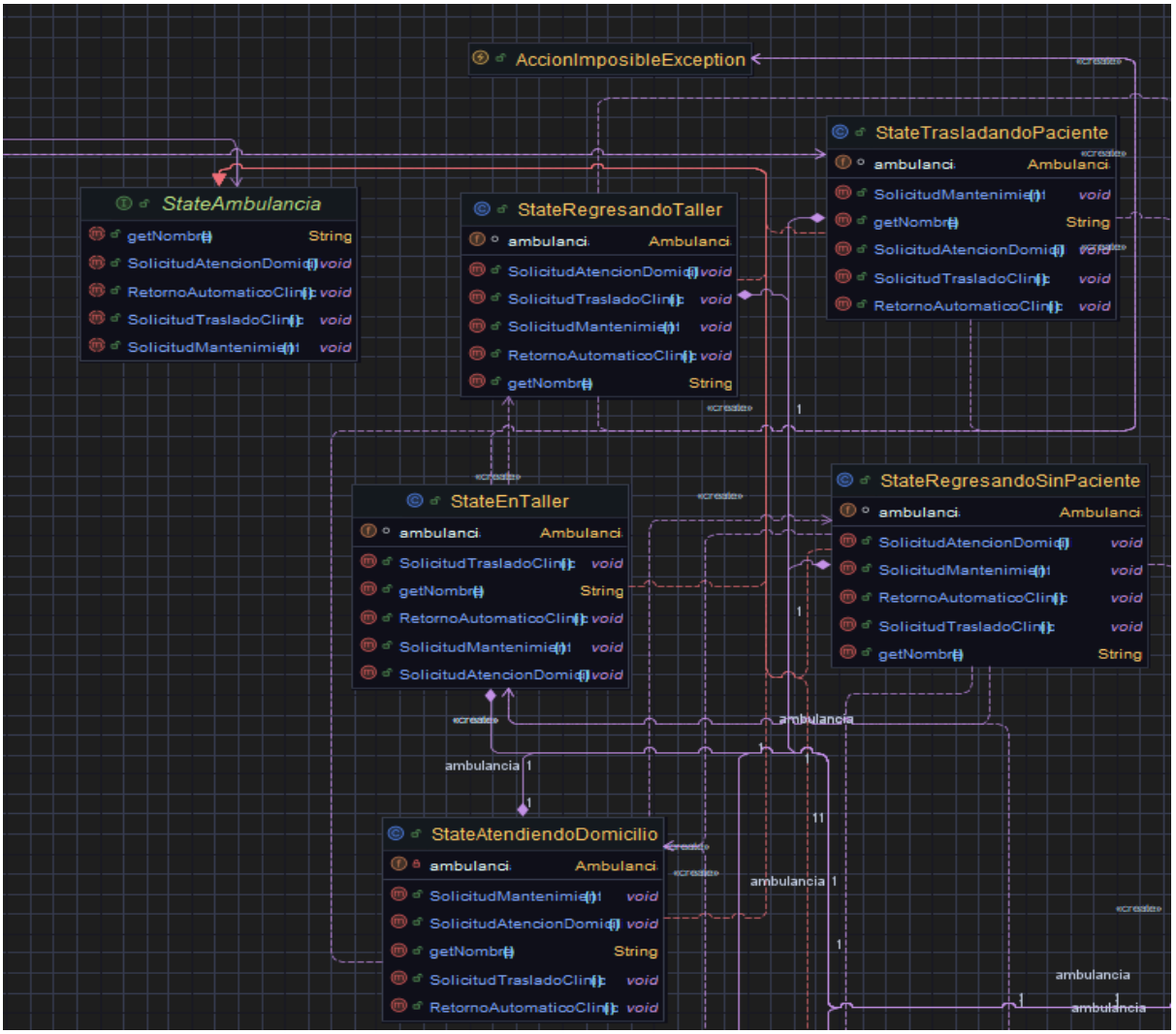




```java
package Modelo.Negocio.clases;

import Modelo.Negocio.interfaces.StateAmbulancia;

public class StateDisponible implements StateAmbulancia {  4 usages  JuanEstebanAldana +1

    private Ambulancia ambulancia;  7 usages

    public StateDisponible(Ambulancia ambulancia) { this.ambulancia = ambulancia; }

    @Override  1 usage  JuanEstebanAldana
    public void SolicitudAtencionDomicilio() {
        this.ambulancia.setEstado(new StateAtendiendoDomicilio(this.ambulancia));
    }

    @Override  1 usage  JuanEstebanAldana
    public void SolicitudTrasladoClinica() { this.ambulancia.setEstado(new StateTrasladandoPaciente(this.ambulancia)); }

    @Override  1 usage  JuanEstebanAldana
    public void RetornoAutomaticoClinica() {
        //NADA
    }

    @Override  1 usage  JuanEstebanAldana
    public void SolicitudMantenimiento() { this.ambulancia.setEstado(new StateEnTaller(this.ambulancia)); }

    @Override  Agustín Proia
    public String getNombre() { return "Disponible"; }
}
```

En estas 2 imágenes vemos cómo se implementa el patrón. En la primera imagen tenemos la interfaz mencionada previamente ***StateAmbulancia***, que según el "***State"*** que se esté desarrollando realizará una u otra cosa.
En la segunda imagen tenemos la implementación del **StateDisponible**, que es el estado "base" de la Ambulancia, aquí podemos ver cómo es que cambia el estado de la ambulancia según el método que se llama de esta además de los diferentes comportamientos "base" que puede realizar estando en este estado.

Partes del diagrama UML del patrón **State**:




## Concurrencia

Ya que existen múltiples personas que pueden solicitar hacer uso de la Ambulancia, es necesario entonces arbitrar un sistema en que, las solicitudes de cambio de estado no queden olvidadas (No sean descartadas). Esto se modela haciendo uso de programación concurrente. Existe en este caso una única **Ambulancia**, esta será entonces el *recurso compartido*. Partiendo de que todos los tipos de uso son excluyentes, no se pueden hacer dos cosas a la vez con la Ambulancia, se modela cada solicitud de acción como un hilo de ejecución, mediante el cual las personas (Asociados y el Operario) acceden al uso de la Ambulancia. La idea detrás de los hilos es no dejar "morir" la solicitud que se realizó. Es decir, si la ambulancia no puede atender la solicitud en el momento que se realiza, debe hacerlo posteriormente, cuando las condiciones estén dadas para realizar lo pedido. De estar dadas las condiciones proceden a utilizar el recurso, inhabilitando su uso para el resto de los hilos. Para nuestro caso particular, solo existen condiciones referentes a la acción que está siendo realizada al momento de la solicitud, aunque estas podrían no ser únicas en un futuro, por ejemplo, si se implementa un atributo en la ambulancia que indique la cantidad de nafta que posee y la misma es cero, ninguna acción podría ser realizada, independientemente de si el estado actual de la ambulancia lo permitiera. En caso de estar inhabilitado el *recurso compartido*, siendo utilizado por otro hilo, el hilo que está realizando la solicitud procederá a esperar. Al finalizar el uso del *recurso compartido*, el hilo notificará a los hilos que están esperando hacer uso del recurso, habilitándolos de nuevo para competir por el mismo. El siguiente uso del recurso estará dado puramente por la velocidad con que los hilos alcancen a solicitarlo, no existe ningún tipo de prioridad.

Los hilos extienden de **Thread** para realizar específicamente lo requerido por cada caso, contienen un método **run**, que es el que se llevará a cabo una vez el hilo comience su ejecución con la llamada **Thread.start()**. Dicho método es el que simula el comportamiento de los **Asociados** y del **Operario**, solicitando a la ambulancia distintas peticiones como pueden ser: Atención a domicilio, traslados a clínica, enviar al taller, etc. Siempre y cuando se esté ejecutando la simulación y el usuario no le haya dado a finalizar.

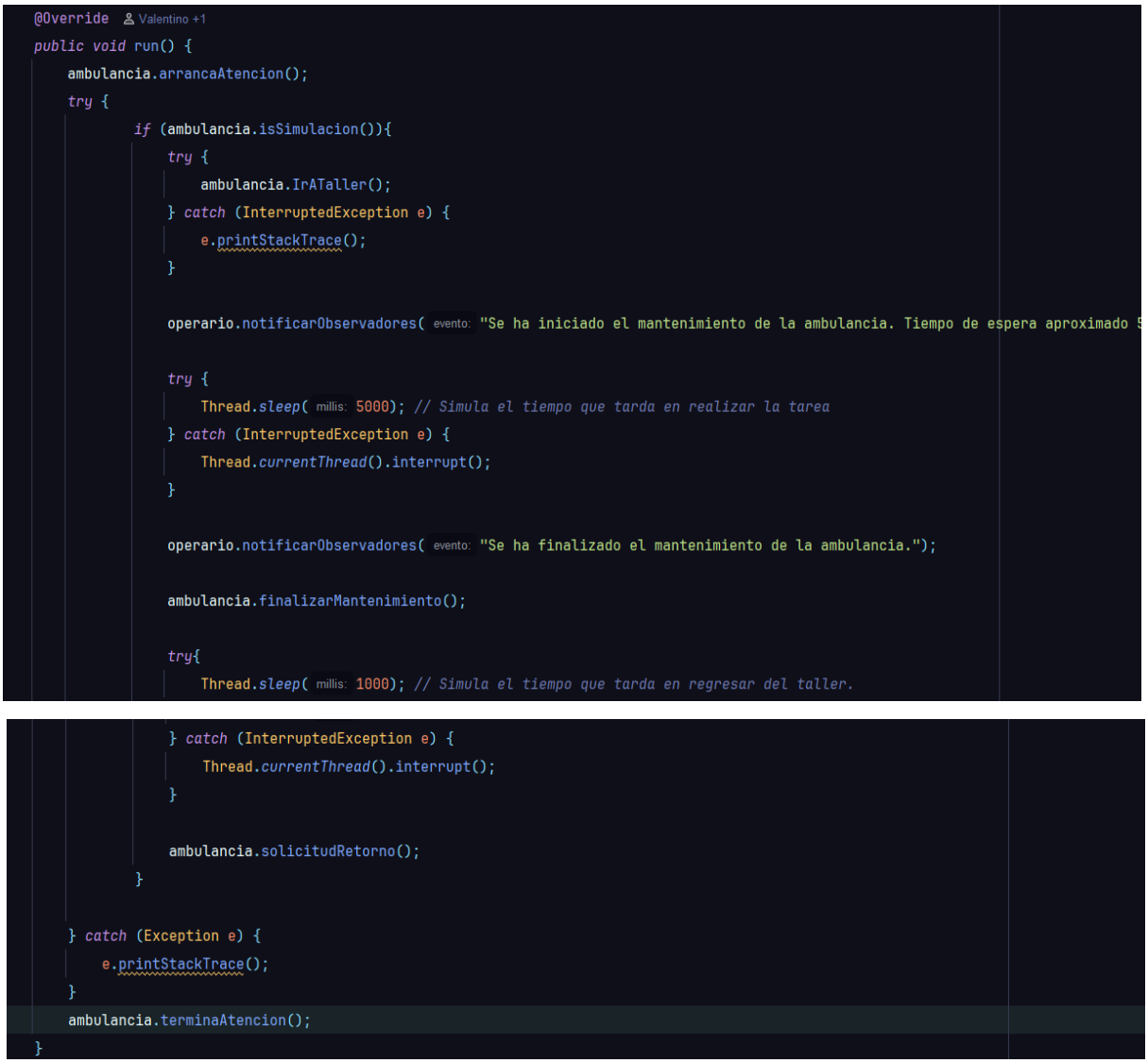
```
public class ThreadAsociado extends Thread {  1 usage  Agustín Proia +1
    @Override  Agustín Proia +1
    public void run() {
        try {
            int n = solicitudes;
            Random random = new Random();
            ambulancia.arrancaAtencion();
            int i=0;
            while(i<n && ambulancia.isSimulacion()) {

                int solicitud = random.nextInt( bound: 2) + 1;

                int tiempoDeEspera = random.nextInt( bound: 5000) + 1000;
                try {
                    Thread.sleep(tiempoDeEspera);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                if(solicitud == 1)
                    try {
                        ambulancia.solicitaAtencionADomicilio();
                        if (ambulancia.isSimulacion())
                            asociado.notificarObservadores( evento: "El asociado: " + asociado.getNombre() + " ha solicitado atención a domici
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                else
```

```
                else
                    try {
                        ambulancia.solicitaTrasladoAClinica();
                        if (ambulancia.isSimulacion())
                            asociado.notificarObservadores( evento: "El asociado: " + asociado.getNombre() + " ha solicitado traslado a cli
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                if (ambulancia.isSimulacion()){
                    tiempoDeEspera = random.nextInt( bound: 5000) + 1000;
                    try {
                        Thread.sleep(tiempoDeEspera);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                ambulancia.solicitudRetorno();

                i++;
            }
            System.out.println("ThreadAsociado finalizado");
            if (ambulancia.isSimulacion()){
                ambulancia.terminaAtencion();
            }
```

Aquí tenemos 2 imágenes que muestran el método **run** de la clase **ThreadAsociado**, que es la clase utilizada para representar solicitudes de un **Asociado** de manera concurrente. Hay ciertas características del código para destacar:

- El uso de **Thread.sleep()** es clave para la simulación. Se simula lo que tardaría realmente la **Ambulancia**. También se utiliza para simular el hecho de que un **Asociado** no está constantemente llamando a la ambulancia y evitar casos en el que todas las solicitudes provengan del mismo **Asociado** repetidas veces.
- Además del *número de solicitudes* por **Asociado**, verificamos que la **Ambulancia** siga en simulación ya que el usuario desde la interfaz gráfica puede finalizar la simulación. (Si se deshabilita la Ambulancia, se finaliza la Simulación) (Esto nos permite "finalizar" la simulación sin detener los hilos abruptamente).

De una forma muy parecida se implementó la clase **ThreadOperario**, sólo que su método **run** varía ligeramente respecto a lo visto en **ThreadAsociado**.







Como vemos en las imágenes y debido a como está implementado el **run** en **ThreadOperario** no se utiliza un ciclo while con un número fijo de solicitudes ya que este

Thread se crea y se ejecuta cada vez que el usuario lo quiera, presionando el boton “Mantenimiento” en la vista.

## Base de Datos, Patrón Singleton

Para esta segunda parte se requería realizar la persistencia de cierta información, a fin de facilitar el uso del programa. Para esto se utilizó una base de datos. Resulta entonces necesario garantizar un uso apropiado de la base. Para evitar generar inconsistencias, es conveniente realizar una única conexión a la base de datos (No se interactúa con múltiples bases de datos a la  vez), además es necesario garantizar que toda la información esté actualizada al momento de utilizarse. Para esto, se aplica el patrón **Singleton**. El mismo garantiza la existencia de una única instancia de una clase. Para este caso, se aplica este patrón a la clase **BaseDeDatos**. El patrón consta en generar un método **getInstance**, que oficiara de “Constructor”, será el método que será invocado para realizar la conexión y se asegurará que sólo una instancia de **BaseDeDatos** pueda ser inicializada a la vez.

```
public static BaseDeDatos getInstance(){  Valentino
    if (instance == null){
        instance = new BaseDeDatos();
    }
    return instance;
}
```

Como se observa, este método será el responsable de evaluar si corresponde o no generar una instancia de la base de datos, de ser así, será quien llame al verdadero constructor privado (es una formalidad ya que en este caso el Constructor únicamente crea al Objeto).

```
private BaseDeDatos()  1 usage  GianLuDiste +1
{
```

En esta clase se encuentran los campos correspondientes al acceso a la base de datos, en caso de tener que cambiar los *drivers*, *usuario*, *contraseña* y/o el *servidor* se debe hacer en esta clase.

```java
public void conectar() throws SQLException  1 usage  Valentino +1 *
{
    String url = "jdbc:mysql://127.0.0.1:3306/";
    String bd = "grupo_9";
    String usuario = "root";
    String password = "root";

    // 1. Conexión al servidor (sin BD) para asegurar que la BD exista
    Connection serverCon = null;
    try {
        serverCon = DriverManager.getConnection(url, usuario, password);
        java.sql.Statement s = serverCon.createStatement();

        // 2. Comando clave: Crea la BD solo si no existe
        s.execute( sql: "CREATE DATABASE IF NOT EXISTS " + bd);
        s.close();
    } finally {
        // Cerramos esta conexión temporal al servidor
        if (serverCon != null) {
            serverCon.close();
        }
    }
    // 3. Ahora sí, nos conectamos a nuestra base de datos "grupo_9"
    //    (que sabemos que ya existe)
    this.con = DriverManager.getConnection( url: url + bd, usuario, password);
    //this.con = DriverManager.getConnection(url + bd, usuario, password);
}
```

Como se ve en la imagen, todo lo relacionado a la conexión con la base de datos se realiza en un único método **conectar**, esto nos facilita el cambio de *datos de acceso* si fuese necesario.

## Nuevas clases/Interfaces

A continuación vamos a dar una explicación breve de las clases implementadas en esta segunda etapa del proyecto:

- **Controlador**: Su propósito principal es actuar como el "intermediario" entre la interfaz de usuario (la **Vista**) y la lógica de negocio (el **Modelo**). Extiende de *WindowAdapter* para el manejo cuando se cierre la ventana e implementa *ActionListener* para responder a las acciones del usuario en la vista.

- **Módulo Simulación**: Como comentamos en un principio este es el módulo perteneciente al **Sistema** que se encarga principalmente de lo relacionado a iniciar, manejar y finalizar la simulación del programa.

- **Ambulancia**: Esta clase es uno de los elementos principales de la simulación, a continuación alguna de sus responsabilidades:
    1. Es el recurso compartido (la única ambulancia) por el que compiten múltiples hilos (**ThreadAsociado** y **ThreadOperario** para el mantenimiento).
    2. Es un **Observable**: Hereda de **Observable** e implementa *IObservable*. Tanto **Asociado** como **Operario**, implementan *IObservable* ya que le deben notificar a sus observadores que sufrieron cambios/realizaron acciones.
    3. Posee un atributo de tipo **StateAmbulancia** que es parte fundamental de la aplicación del patrón **State** comentado previamente.

- **Asociado**: A diferencia de **Ambulancia**, el Asociado sólo implementa la interfaz *IObservable* y no extiende Observable ya que sino se perdía la línea de herencia con **Persona**. Asociado le va a notificar a su observador cada vez que haga uso de la ambulancia, sea para pedir transporte a la clínica o para atención a domicilio.

- **Operario**: Mismo caso que **Asociado** para el tema de la implementación de *IObservable*. El operario le avisa a su observador cada vez que envía a la ambulancia a mantenimiento.

- **ThreadOperario/ThreadAsociado**: Ambas funcionan igual aunque con ligeras diferencias, extienden de **Thread** lo que permite sobreescribir el método **run**, dicho método es el que se va a ejecutar a la hora de hacer el llamado **Thread.start()** en la simulación y empezar la ejecución de los hilos aprovechando la concurrencia.

- **ObservadorAsociados/ObservadorAmbulancia/ObservadorOperario:** Estas tres clases son las encargadas de observar distintos objetos de clases a las que hace referencia su nombre, por ello cada implementación necesita de un atributo que sea un ArrayList de su respectivo tipo (aunque en nuestra clínica solo haya una ambulancia y un único operario se decidió usar un arreglo para que el sistema esté abierto a su extensión). También cuentan con una referencia al **Controlador**, esto con el fin de que en caso de recibir notificaciones de los objetos los cuales se están observando, se le avise al controlador y este le notifique a la vista para que actualice el estado de los objetos (similar al **Patrón MVC**). Estas tres clases, por obvias

razones, tienen métodos para modificar los **Observables** y el método **update** que es el encargado de avisar al **Controlador** los cambios.

- **BaseDeDatos** (DAO): Como bien indica su nombre es la clase que se encarga de realizar determinadas tareas, fijadas por la interfaz *IPersistencia*, en la base de datos. Esta clase utiliza el patrón **Singleton** para únicamente tener una instancia de la base de datos. Es la encargada de conectarse y desconectarse de la BD, crear las tablas de la misma, reiniciarla si fuera necesario, como también de inicializarla con datos por defecto, realiza consultas a la BD para recuperar los **Asociados** y tiene la capacidad de dar de alta y/o baja a un determinado **Asociado** si el usuario así lo desea.

- **AsociadoExistenteException/AsociadoInexistenteException:** Son las dos excepciones que aparecen en **altaBD** y **bajaBD** respectivamente, son lanzadas cuando se quiere agregar a la BD un asociado que ya está en la lista (la clave para fijarse si es el mismo asociado o no es el DNI) o cuando se quiere eliminar un asociado que no está en la BD.

## Ejecución del programa

**Para ejecutar la simulación:**

1. Dirígete a la clase **PruebaVistas**.
2. Ejecuta el método main que se encuentra allí.
3. Se abrirá una ventana para la **configuración** de la simulación.
4. Una vez que termines de configurar e inicies la simulación, se lanzará la **ventana principal**.