

Universidad Nacional de Mar del Plata

Carrera: Ingeniería informática

Materia: Programación C

Trabajo Práctico Grupal - Segunda etapa

Profesores:

- Ivonne Gellon
- Leonel Domingo Guccione
- Carlos Guillermo Lazurri

Integrantes:

- Gribman, Ianai
- Gregorini, Dante
- Vazquez Baccaro, Mateo
- Acevedo, Valentina
- Milosevic, Natasha

1. Introducción a las clases utilizadas

Paquete Modelo Simulación

➤ *Ambulancia*

Clase concreta que extiende de Observable. Posee una referencia a su estado y métodos sincronizados para las posibles solicitudes que pueden mandar los hilos, se delega a *estado* el cambio del mismo, haciendo uso del patrón State. En esta clase también se mantiene al tanto la activación de la simulación.

➤ *Solicitante*

Clase abstracta que representa los posibles solicitantes de la ambulancia. Extiende Observable. Implementa Runnable.

➤ *Asociado, EventoRetorno y Operario*

Clases concretas que extienden de Solicitante. Sobreescriben el método run según las solicitudes que le pueden hacer a la Ambulancia, y notifican a sus observadores al momento de comenzar o finalizar dichas solicitudes.

Paquete Controladores

➤ *Controlador*

Clase abstracta que extiende WindowAdapter e implementa ActionListener. Posee referencias al modeloSimulacion, controladorAsociados y la interfaz de la vista. Esta clase recibe los eventos producidos por los botones de la vista, y delega las tareas a los controladores específicos.

➤ *ControladorAsociados*

Clase concreta. Posee una referencia a la interfaz de la base de datos y a la vista que se relaciona con la misma. Es la encargada de gestionar los cambios en la BD y la muestra de los mismos por pantalla. Por ello, posee métodos para abrir o cerrar la conexión, y agregar o eliminar asociados de la tabla.

➤ *IVistaAsociados, IVistaControlador, IVistaSimulacion*

Interfaces que definen los métodos de la vista relacionados con: la gestión de asociados, el controlador general y aquellos que incumben la simulación, respectivamente.

➤ *ManagerXMLInicializacion*

Clase concreta encargada de cargar y leer del archivo XML que contiene los datos de los asociados para la inicialización de la base de datos. Para ello, utiliza una colección de AsociadosDTO.

➤ *ManagerXMLSimulacion*

Clase concreta, destinada a cargar y leer el archivo XML que contiene la cantidad máxima y mínima de solicitudes que puede hacer un asociado a la Ambulancia.

➤ *ModuloSimulacion*

Clase concreta, que se encarga de crear los hilos, ya sea a partir de la lista de AsociadosDTO o del Evento Retorno, a la vez pide la representación de los mismos por pantalla. También produce la activación o desactivación de la simulación, en lo cual informa a la Ambulancia y la inicializa.

➤ *OjoSimulacion*

Clase concreta que extiende de Observer. Cumple la función de observar los hilos y la ambulancia, para luego gestionar la vista al ser notificado de algún cambio en sus observados.

➤ *ParametrosSimulacion*

Clase que contiene la cantidad máxima y mínima de solicitudes que puede hacer un asociado.

Paquete Patrones State

➤ *IEstado*

Interfaz del State. Tiene los métodos a implementar de las posibles solicitudes que recibe la ambulancia.

➤ *EstadoDisponible, EstadoAtiendeDomicilio, etc*

Clases concretas que implementan IEstado. Representan cada uno de los estados que puede tomar la Ambulancia. Poseen una referencia a la Ambulancia, a partir de la cual mandan la referencia al nuevo estado que debe tomar. También posee métodos booleanos para informar si de cada estado específico será posible realizar ciertas solicitudes.

Paquete Persistencia

➤ *AsociadoDTO*

Clase concreta, utilizada para transferir datos del asociado.

➤ *IBaseDeDatos*

Interfaz que representa la base de datos del sistema.

➤ *BaseDeDatosDAO*

Clase concreta, implementa IBaseDeDatos, es la encargada de crear la tabla, y gestionar la misma para leer, agregar o eliminar asociados.

➤ *ManagerXMLConfig*

Clase concreta encargada de leer y cargar el archivo XML que contiene los datos relacionados a la base de datos (host, usuario y clave).

➤ *ParametrosBaseDeDatos*

Clase concreta que encapsula los parámetros de acceso a una base de datos, para poder almacenarla y/o accederla en un archivo XML.

Paquete Util

➤ *Acciones*

Clase que contiene las constantes utilizadas para los Action Commands, tanto de la gestión como de la simulación.

➤ *Util*

Clase concreta que implementa métodos para el cálculo de los tiempos muertos utilizados en las demás clases.

Paquete Vista

➤ *ConfirmationPopUp, CustomPopUp*

Clases concretas, representan a las ventanas emergentes de la simulación.

➤ *JFramePrincipal*

Clase concreta que implementa todas las interfaces de las vistas. Representa el marco principal de la simulación. Permite alternar entre la ventana de gestión y la de simulación, así como mostrar los PopUps.

➤ *JPanelExtendido*

Clase abstracta con métodos comunes para las ventanas.

➤ *VentanaGestion*

Clase concreta que implementa KeyListener, ActionListener y ListSelectionListener. Representa a la ventana donde los usuarios manipulan a los Asociados y configuran la simulación.

➤ *VentanaSimulacion*

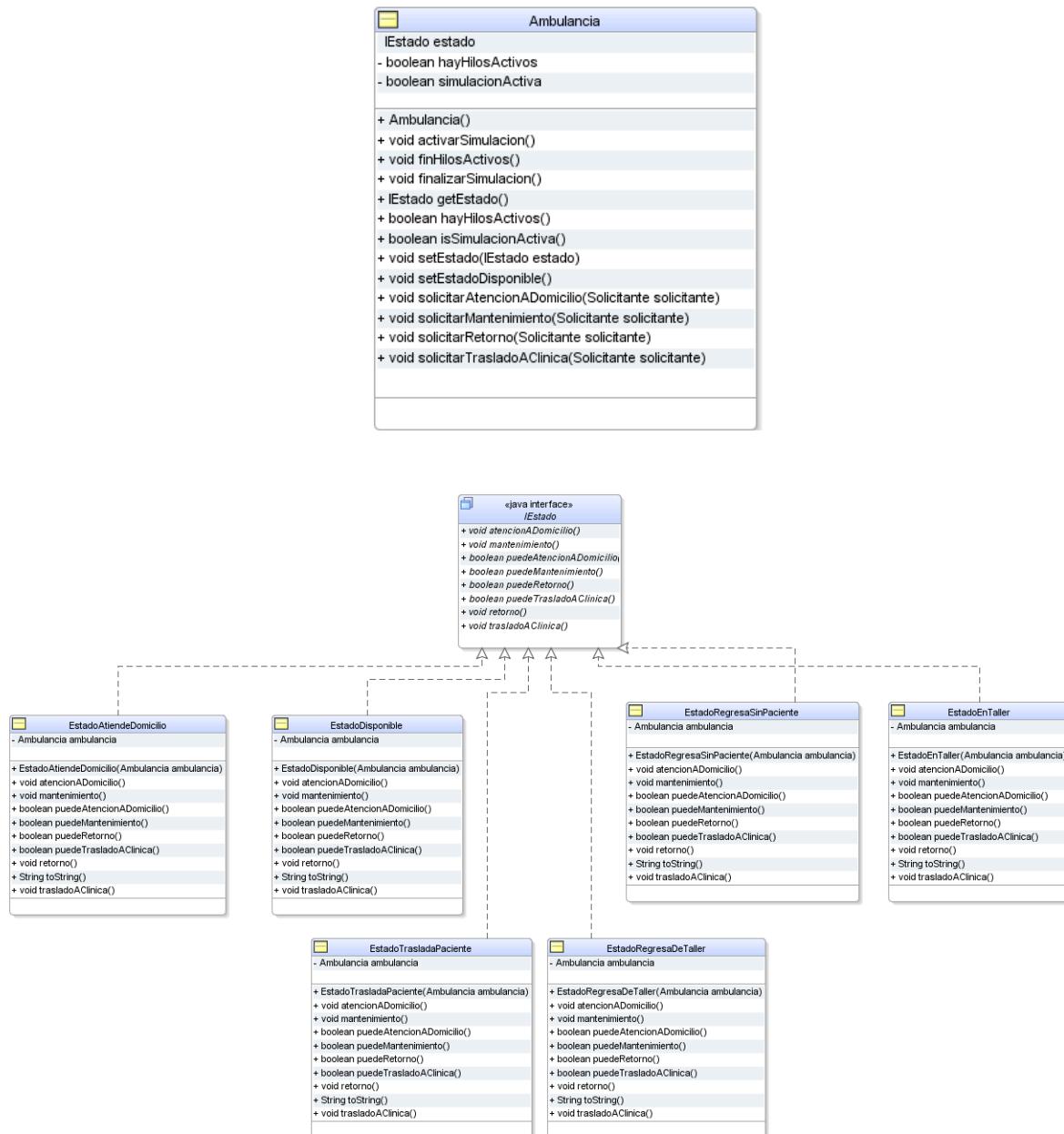
Clase concreta, representa toda la vista de la simulación, para mostrar los llamados (nuevos y atendidos), el estado de la ambulancia, etc.

2. Diseño

El proyecto aplica distintos patrones de diseño vistos en clase.

Patrón State

➤ UML Relevante



➤ *Ambulancia*

```
public class Ambulancia extends Observable {
    IEstado estado;
    public synchronized void setEstado(IEstado estado) {
        this.estado = estado;

    }

    public synchronized void
    solicitarAtencionADomicilio(Solicitante solicitante) throws
    InterruptedException {

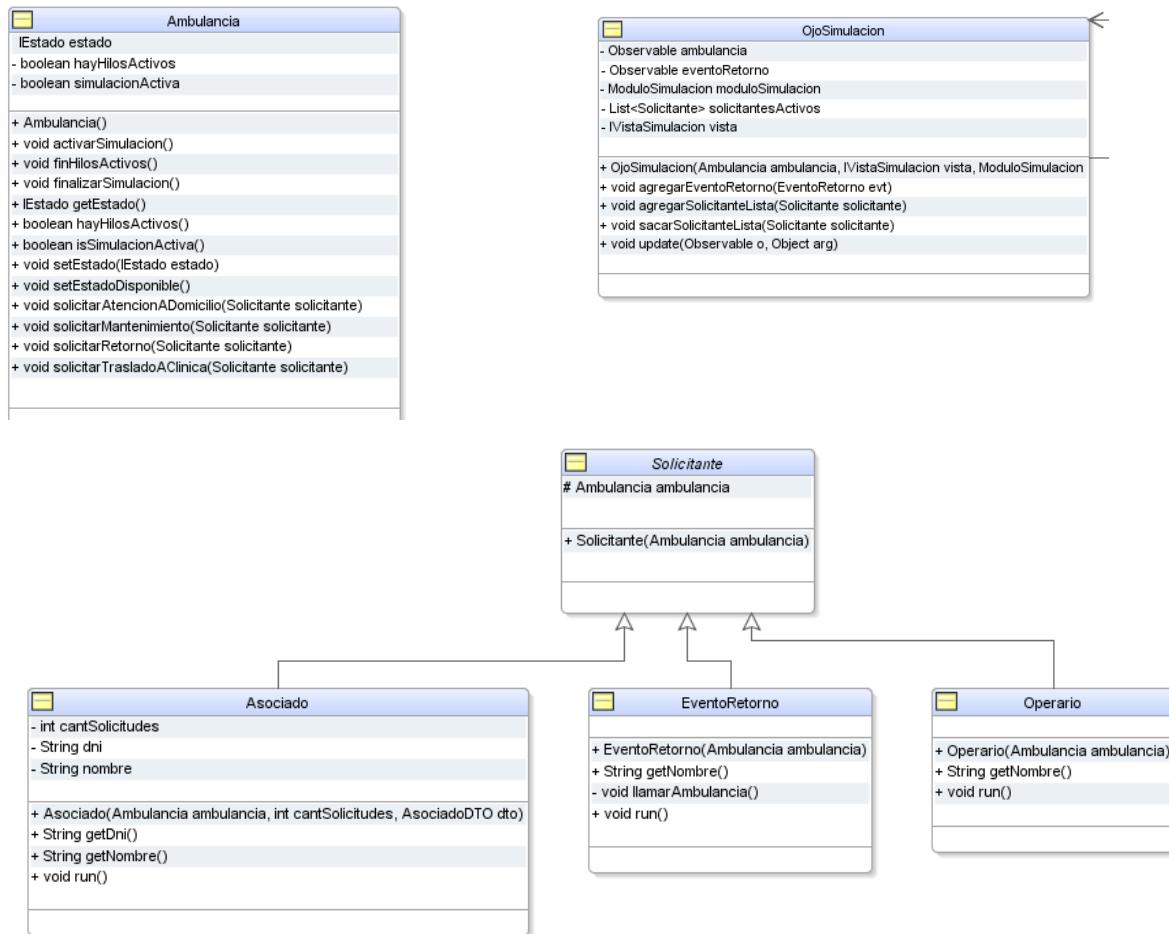
        while (!this.estado.puedeAtencionADomicilio())
            wait();

        this.estado.atencionADomicilio();
    }
}
```

El recurso compartido, la Ambulancia, actúa de contexto, conteniendo dentro de sí, mediante una relación de composición, una referencia a su estado (con variable de tipo IEstado, que es la interfaz que implementan todos los estados y define los métodos que deben implementar, que son los que conoce la Ambulancia). Al recibir una solicitud (como solicitarAtencionADomicilio), la Ambulancia delega el cambio de estado a su compuesto (su estado) gracias al polimorfismo, luego de consultar que el cambio de estado sea posible por cuestiones de concurrencia que desarrollamos a continuación. En la implementación de cada estado definimos a qué estado cambiar (o si cambiar en absoluto) al ser llamado cada método de la interfaz. Si se define qué debe cambiar, el estado llama al método público setEstado de la ambulancia, pasándole por parámetro otro estado creado en el momento, del tipo del estado al que se quiere cambiar. Esto es posible gracias a que el estado tiene también una referencia a la Ambulancia en una relación de composición, lo que implica una doble referencia que no presenta problemas si se aplica el patrón correctamente.

Patrón Observer-Observable

➤ UML Relevante



➤ Ambulancia

```

public class Ambulancia extends Observable {
    ...
    public synchronized void setEstado(IEstado estado) {
        this.estado = estado;
        setChanged();
        notifyObservers(new NotificacionSimulacion(Acciones.ESTADO, estado.toString()));
        notifyAll();
    }
    ...
}
    
```

La Ambulancia es un Observable (extiende de la clase concreta de Java Observable, que define e implementa métodos como addObserver), permitiendo que otros objetos (Observers) se suscriban a sus cambios. Lo que posibilita esto es que

cuando la Ambulancia lo considere pertinente (en nuestro caso, cuando se llama al método setEstado), ésta puede notificar a sus observadores que cambió, llamando primero al método setChanged, para modificar su flag interna que indica que cambió, y luego a notifyObservers, enviando por parámetro un objeto con información sobre el cambio. En particular, la Ambulancia envía un objeto de tipo NotificacionSimulacion que contiene dos strings, que en este caso valdrán: el string que indica que el cambio fue un cambio de estado y el string que indica a qué estado.

➤ *OjoSimulacion*

```
public class OjoSimulacion implements Observer {
    private Observable ambulancia;
    private Observable eventoRetorno;
    ...
    public OjoSimulacion(Ambulancia ambulancia, IVistaSimulacion
vista, ModuloSimulacion moduloSimulacion) {
        super();
        this.ambulancia = ambulancia;
        ambulancia.addObserver(this);
        this.vista = vista;
        this.moduloSimulacion = moduloSimulacion;
        this.solicitantesActivos = new
ArrayList<Solicitante>();
    }

    @Override
    public void update(Observable o, Object arg) {
        if (o == this.ambulancia) {
            NotificacionSimulacion notif =
(NotificationSimulacion) arg;
            vista.cambiarEstadoAmbulancia(notif.getMensaje());

        }
        else if (o == this.eventoRetorno) {...
```

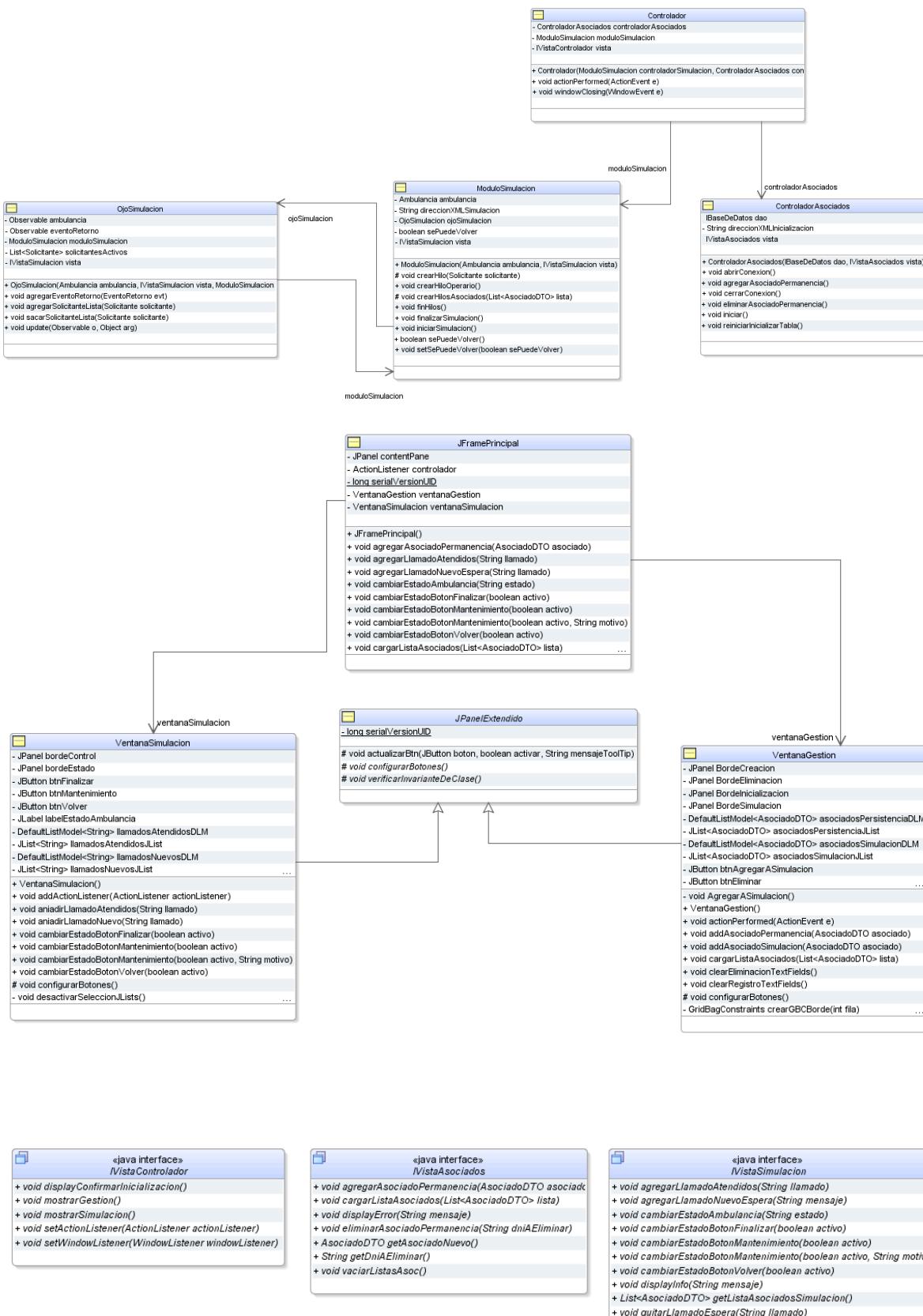
A la Ambulancia la observa un OjoSimulacion que implementa la interfaz Observer, que lo único que hace es requerir que se implemente un método update. Este método tiene dos parámetros, el primero es un Observable que identifica qué objeto envió la notificación y el segundo un Object que contiene información sobre el cambio. En general, los Observers pueden observar a varios Observables simultáneamente, diferenciándolos comparando el primer parámetro de update con una referencia que deben tener al objeto Observable instanciado. En este caso, OjoSimulacion observa varios Observables, así que tiene una referencia a la Ambulancia, y en el comienzo de update se pregunta si el Observable del parámetro es igual a la referencia de Ambulancia, en cuyo caso castea el segundo parámetro a NotificacionSimulacion porque por contrato sabe que es el tipo que Ambulancia usa para enviar la información pertinente al cambio. OjoSimulacion a su vez le avisa a la vista (a la cual conoce a través de una variable de tipo IVistaSimulacion, desacoplándolo de implementaciones particulares de la vista) qué tiene que cambiar a través de sus métodos públicos como cambiarEstadoAmbulancia.

De manera similar, el OjoSimulacion observa un EventoRetorno que notifica cuando intenta solicitar el retorno de la ambulancia, cuando dicha solicitud fue procesada, y cuando detecta que ya no quedan otros hilos activos. Ante esto, el OjoSimulacion le avisa a la vista cómo mostrar el cambio y al moduloSimulacion si tiene que cambiar algo.

También observa Solicitantes (los hilos que hacen solicitudes). Estos Solicitantes notifican ante eventos como cuando inician un pedido, cuando terminan de ser atendidos o cuando abandonan la simulación. El OjoSimulacion actualiza su lista interna de Solictantes y le avisa a la vista cómo mostrarlo (y si pasa a estar vacía la lista, le avisa al moduloSimulacion).

Patrón MVC

➤ UML Relevante



➤ Modelo

```
// En EstadoDisponible
public class EstadoDisponible implements IEstado {
    @Override
    public void atencionADomicilio() {
        ambulancia.setEstado(new
EstadoAtiendeDomicilio(this.ambulancia));
    }
    @Override
    public boolean puedeAtencionADomicilio() {
        return true;
    }
}
```

```
// En Asociado
@Override
public void run() {
    try {
        int i = 0;
        while (this.ambulancia.isSimulacionActiva() && i <
this.cantSolicitudes)
        {
            boolean traslado = Math.random() < 0.5;
            Util.tiempoMuerto();
            if (traslado) {
                this.ambulancia.solicitarTrasladoAClinica(this);
            }
            else {
                this.ambulancia.solicitarAtencionADomicilio(this);
            }
            i++;
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

El Modelo representa el núcleo de la simulación y la lógica de negocio, totalmente independiente de la interfaz gráfica. En este proyecto, el Modelo está compuesto por la clase Ambulancia (el recurso compartido que gestiona su propio estado), las clases del patrón State (que definen las reglas de negocio de qué puede

hacer la ambulancia y cuándo), y las clases Solicitante, Asociado, Operario y EventoRetorno. Estas clases son a su vez "observables" y notifican sus cambios de estado (como el inicio de una solicitud), pero no tienen conocimiento de la existencia de ninguna ventana o botón.

➤ Vista

```
// En VentanaGestion
public void setActionListener(ActionListener actionListener) {
    this.btnSimular.addActionListener(actionListener);
    this.btnRegistrar.addActionListener(actionListener);
    this.btnEliminar.addActionListener(actionListener);
    this.btnInicializar.addActionListener(actionListener);
}

// En VentanaSimulacion
public void informarCambioEstado(String estadoAmbulancia) {
    this.verificarInvarianteDeClase();
    assert estadoAmbulancia != null;
    this.labelEstadoAmbulancia.setText(estadoAmbulancia);
}
```

La Vista es la capa de presentación construida con Java Swing y está diseñada para no contener lógica de negocio. Está representada por VentanaGestion y VentanaSimulacion. VentanaGestion se encarga de mostrar los formularios de altas y bajas y las listas de asociados. VentanaSimulacion actúa como el panel de monitoreo en tiempo real, mostrando el estado de la ambulancia y las listas de llamados. La única función de la Vista es mostrar la información que le ordenan los controladores y capturar la interacción del usuario (clics), reportándolos a su ActionListener asignado. La Vista no conoce al Controlador ni al Modelo directamente. La Vista también se encarga de habilitar/deshabilitar los botones según la interacción del usuario, además muestra el motivo por el cual están deshabilitados con un toolTipText al pasarle el cursor por encima.



➤ Controlador

```
// En Controlador
@Override
public void actionPerformed(ActionEvent e) {
    switch(e.getActionCommand()) {

        case Acciones.REGISTRAR: {
            controladorAsociados.agregarAsociadoPermanencia();
            break;
        }
        ...
        case Acciones.SIMULACION: {
            controladorAsociados.cerrarConexion();
            vista.mostrarSimulacion();
            moduloSimulacion.iniciarSimulacion();
            break;
        }

        ...
    }
}
```

```
// En OjoSimulacion
@Override
public void update(Observable o, Object arg) {
    if (o == this.ambulancia) {
        NotificacionSimulacion notif = (NotificacionSimulacion)
arg;
        vista.cambiarEstadoAmbulancia(notif.getMensaje());

    }
    ...
}
```

El Controlador actúa como el mediador principal entre la Vista y el Modelo. En este diseño, Controlador.java es el ActionListener principal que recibe todos los eventos de botones de la Vista. Este controlador no ejecuta la lógica directamente, sino que delega la tarea a controladores especializados. Por ejemplo, cuando el usuario presiona "Registrar", Controlador llama a ControladorAsociados para que interactúe con el Modelo de persistencia (Base de Datos). Cuando el usuario presiona

"Simular", llama a ModuloSimulacion para que instancie y configure los hilos del Modelo (los Solicitantes y la Ambulancia).

➤ ControladorAsociados

El ControladorAsociados actúa como un controlador especializado cuya única responsabilidad es gestionar la persistencia de los asociados. No interactúa directamente con los clics del usuario, sino que recibe órdenes del Controlador principal. Su función es ser el intermediario exclusivo entre la IVistaAsociados (la VentanaGestion) y el IBaseDeDatos (el DAO del Modelo), asegurando que la Vista nunca acceda directamente a la base de datos y viceversa.

Sus métodos se encargan de lo relacionado a la creación y eliminación de asociados.

Además de la gestión de asociados, este controlador maneja la conexión con la base de datos. Otra responsabilidad del controlador es la gestión de errores: captura excepciones del Modelo y las traduce en un mensaje amigable para el usuario a través de la Vista (como mostrando una ventana de error).

➤ ModuloSimulacion

ModuloSimulacion es otro controlador especializado, al igual que ControladorAsociados. Su responsabilidad no es la persistencia de datos, sino la gestión y coordinación de la simulación activa. Recibe órdenes del Controlador principal y actúa como el punto de entrada lógico para la simulación. Tiene referencias a la Vista y el Modelo. Su primera acción es instanciar su propio controlador-observador, OjoSimulacion, pasándole las referencias del modelo y la vista, y también una referencia a sí mismo. Esto permite que OjoSimulacion le notifique de vuelta a ModuloSimulacion sobre eventos como el fin de la simulación

Gestiona la finalización de la simulación a través de los flags en la Ambulancia. El método finalizarSimulacion (llamado por el Controlador) pone simulacionActiva en false, lo que sirve como señal para que los hilos Asociado dejen de generar solicitudes (ya que su bucle run verifica ambulancia.isSimulacionActiva()). También tiene el método finHilos, que es invocado por OjoSimulacion cuando su lista de solicitantes está vacía. Este método pone hayHilosActivos en false, lo cual es la señal que el hilo EventoRetorno estaba esperando para detener su propio bucle y notificar el fin definitivo de la simulación.

Concurrencia

➤ Código relevante

```
// En Ambulancia
public synchronized void setEstado(IEstado estado) { ... }
```

```
public synchronized void solicitarAtencionADomicilio(Solicitante
solicitante) throws InterruptedException { ... }
public synchronized void solicitarTrasladoAClinica(Solicitante
solicitante) throws InterruptedException { ... }
public synchronized void solicitarRetorno(Solicitante solicitante)
throws InterruptedException { ... }
public synchronized void solicitarMantenimiento(Solicitante
solicitante) throws InterruptedException { ... }
public synchronized boolean isSimulacionActiva() { ... }
public synchronized boolean hayHilosActivos() { ... }
```

```
// En Ambulancia. Representativo de otros métodos parecidos
public synchronized void solicitarAtencionADomicilio(Solicitante
solicitante) throws InterruptedException {

    while (!this.estado.puedeAtencionADomicilio())
        wait();

    this.estado.atencionADomicilio();
}
```

```
// En Ambulancia
public synchronized void setEstado(IEstado estado) {
    this.estado = estado;
    setChanged();
    notifyObservers(new NotificacionSimulacion(Acciones.ESTADO,
estado.toString()));
    notifyAll();
}
```

```
// En Asociado, similar a Operario
@Override
public void run() {
    try {
        int i = 0;
        while (this.ambulancia.isSimulacionActiva() && i <
```

```

this.cantSolicitudes)
{
    ...
    this.ambulancia.solicitarTrasladoAClinica(this);

    ...
    i++;
}

this.notifyObservers(new
NotificacionSimulacion(Acciones.QUITAR_SOLICITANTE_ACTIVO));
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}

```

```

// En EventoRetorno
@Override
public void run() {
    try {
        // Chequea el flag synchronized
        while (this.ambulancia.hayHilosActivos())
        {
            llamarAmbulancia();
            Util.tiempoMuerto();
        }
        // ...
        // Notifica el fin real de la simulación
        this.notifyObservers(new
NotificacionSimulacion(Acciones.NO_HAY_HILOS));
    } catch (InterruptedException e) { ... }
}

```

La concurrencia del sistema se gestiona usando la clase Ambulancia como un Monitor (recurso compartido). Esta clase asegura la exclusión mutua al declarar todos sus métodos (solicitarAtencionADomicilio, setEstado, y los getters/setters de los flags) como synchronized. Esto garantiza que solo un hilo Solicitante pueda acceder al estado de la ambulancia a la vez.

La coordinación de hilos usa los métodos wait() y notifyAll() en combinación con el patrón State. Cada método de solicitud (ej. solicitarAtencionADomicilio) implementa un bucle while. La condición de este bucle es delegada al estado actual (!this.estado.puedeAtencionADomicilio()). Si el estado actual no permite la acción

(ej. EstadoEnTaller retorna false), el hilo invoca wait(). Al llamar a wait(), el hilo libera el monitor de la ambulancia y queda en espera pasiva. Esto es necesario para que otros hilos (como Operario) puedan entrar a la ambulancia y modificar el estado.

Cuando un hilo logra cambiar el estado (ej: un Operario saca la ambulancia del taller), invoca el método ambulancia.setEstado(). El método setEstado, al ser synchronized, se ejecuta de forma sincrónica y, como última acción, llama a notifyAll(). Esta llamada despierta a todos los hilos en wait(), los cuales competirán nuevamente por el monitor. Al volver a adquirirlo, cada hilo vuelve a evaluar la condición de su while contra el nuevo estado.

Persistencia

➤ Código relevante

```
// En AsociadoDTO
public class AsociadoDTO {
    private String nombre, dni;
    public AsociadoDTO(String nombre, String dni) {
        super();
        this.nombre = nombre;
        this.dni = dni;
    }
    // ... Getters y Setters ...
    public String getNombre() {
        return nombre;
    }
    public String getDni() {
        return dni;
    }
}
```

```
public interface IBaseDeDatos {
    String nombreTablaAsociados = "ASOCIADOS";
    String nombreCampoAsociadosDni = "DNI";
    String nombreCampoAsociadosNombre = "NOMBRE";
    List<AsociadoDTO> leerAsociados() throws SQLException,
    SinConexionException;
    void agregarAsociado(AsociadoDTO asociado) throws
    SQLException, SinConexionException;
    void eliminarAsociado(String dni) throws SQLException,
    SinConexionException, NoEliminadoException;
    void abrirConexion() throws SQLException;
```

```

    void cerrarConexion() throws SQLException,
SinConexionException;
    void crearTablaAsociados() throws SQLException,
SinConexionException;
    void reiniciarTablaAsociados() throws SQLException,
SinConexionException;
}

```

```

@Override
    public void eliminarAsociado(String dni) throws SQLException,
SinConexionException, NoEliminadoException {
        assert dni != null : "No se puede eliminar un asociado
con un dni null";
        if (conexion != null) {
            PreparedStatement sentencia =
conexion.prepareStatement(
                    "DELETE FROM " + nombreTablaAsociados +
" WHERE " + nombreCampoAsociadosDni + "=?");
            sentencia.setString(1, dni);
            int filaAfectada = sentencia.executeUpdate();
            if (filaAfectada == 0)
                throw new NoEliminadoException();
        } else
            throw new SinConexionException();
}

```

La persistencia del sistema se implementa desacoplando la lógica de acceso a datos mediante el patrón DAO (Data Access Object). El ControladorAsociados no interactúa directamente con la base de datos, sino que depende exclusivamente de la interfaz IBaseDeDatos. Esta interfaz define el "contrato" de las operaciones de persistencia (como leerAsociados o agregarAsociado), abstrayendo por completo los detalles de implementación de SQL. Para el transporte de datos entre capas, se utiliza el patrón DTO (Data Transfer Object). La clase AsociadoDTO actúa como un contenedor simple que encapsula los atributos nombre y dni, permitiendo que la información del asociado se transfiera desde la Vista al Controlador y al DAO sin exponer los objetos de dominio del Modelo (Asociado).

La implementación concreta, BaseDeDatosDAO, es la única clase del sistema con conocimiento directo de la tecnología de base de datos (MySQL/JDBC). Esta clase implementa la interfaz IBaseDeDatos y traduce sus métodos abstractos en sentencias SQL. Por ejemplo, al invocarse agregarAsociado, el método recibe el AsociadoDTO (enviado por el ControladorAsociados) y construye una sentencia

PreparedStatement con la consulta INSERT INTO ASOCIADOS.... Luego utiliza los getters del DTO (asociado.getDni(), asociado.getNombre()) para poblar la consulta de forma segura. Esta clase también es responsable de la gestión de la conexión (abrirConexion, cerrarConexion) y de la creación de la tabla (crearTablaAsociados), encapsulando toda la lógica de JDBC y la gestión de SQLException.