TRABAJO PRÁCTICO GRUPAL - PROGRAMACIÓN C - PARTE 2

Patrones

Grupo 8 - integrantes:

- Mariano Andrés Horianski
- Fernando Daniel Rosal
- Jazmín Milagros Piriz
- Jennifer Beltrán Flores

Importante:

Los atributos y métodos necesarios para iniciar la conexión con la base de datos se encuentran en el paquete **persistencia.BasedeDatos**, en la clase *BDConexion*.

En las lineas 14, 15 y 16 se tiene:

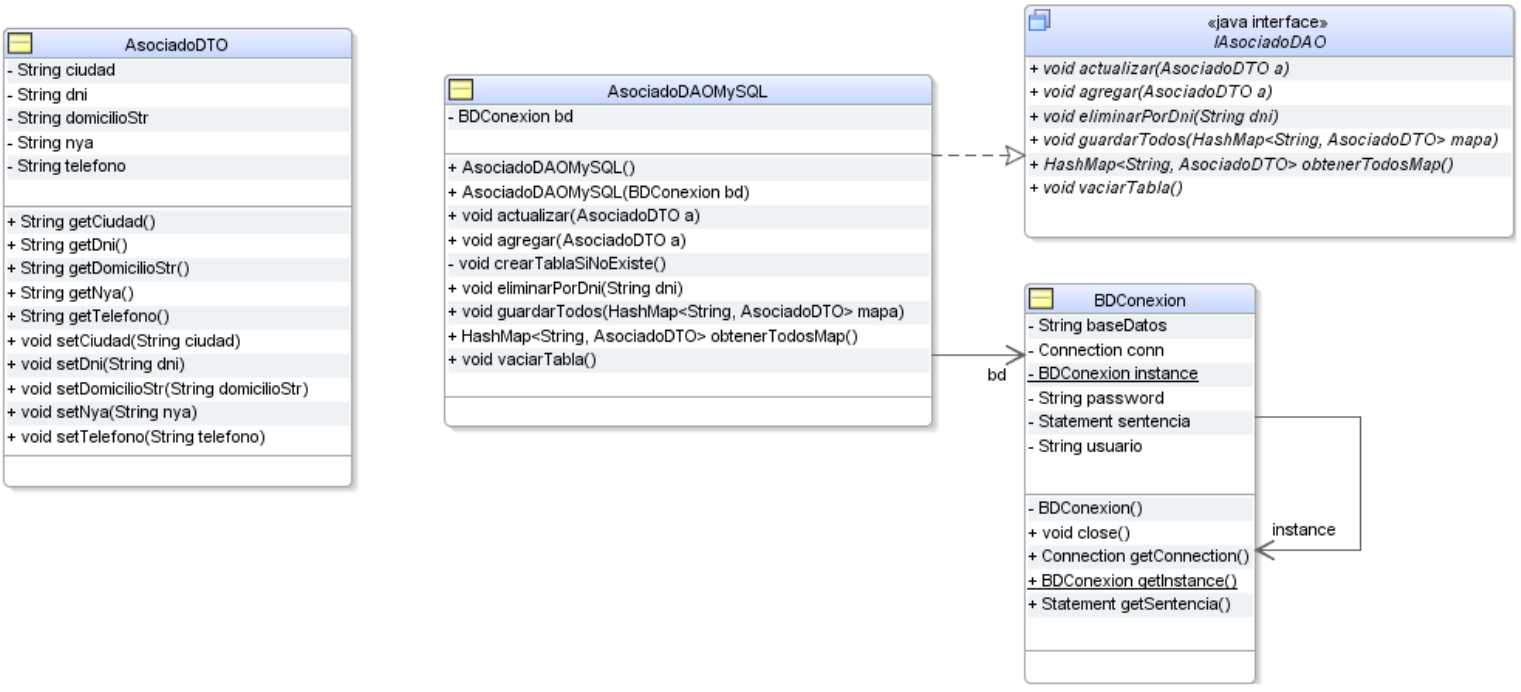
```
private String baseDatos = "jdbc:mariadb://localhost:3308/Grupo_8";
private String usuario = "progra_c";
private String password = "progra_c";
```

Mientras que en la línea 29 se carga el driver:

```
Class.forName("org.mariadb.jdbc.Driver");
```

En este informe se describirán los patrones utilizados durante la **segunda parte.**

SINGLETON: package persistencia. BasedeDatos;



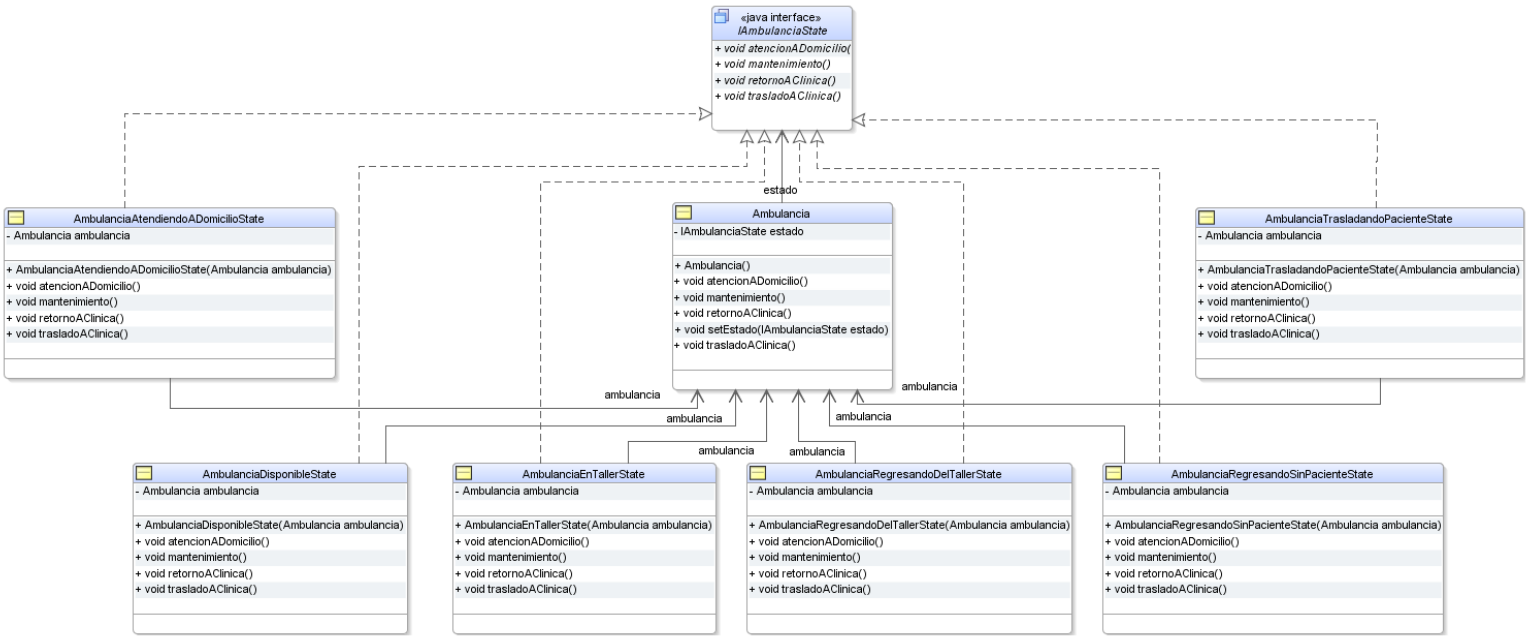
BDConexion.java:

Clase Singleton que inicializa y provee la conexión con la base de datos. Centraliza la gestión de la conexión JDBC en un único punto para que lel DAO pueda obtener la conexion sin repetir parámetros ni lógica de inicialización. Gracias al patrón Singleton asegura una única instancia de conexión accesible desde toda la aplicación.

```
public class BDConexion {

//Atributo necesario para el singleton (la instancia)
private static BDConexion instance;
//Atributos necesarios para la conexion
private Connection conn;
private String baseDatos = "jdbc:mariadb://localhost:3308/Grupo_8";
private String usuario = "progra_c";
private String password = "progra_c";


//Constructor
private BDConexion() {
      ...
      // carga el driver
       Class.forName("org.mariadb.jdbc.Driver");
      ...
      //crea la conexion con la base de datos, un statement y activa
el auto commit (cada sentencia individual se ejecuta
automaticamente)
         this.conn = DriverManager.getConnection(baseDatos, usuario,
          password);
         this.sentencia=conn.createStatement();
         this.conn.setAutoCommit(true);

    ...

}

//Patrón Singleton: si no existe, crea una instancia única de
conexión que luego retornará en cada llamado
```

```
public static synchronized BDConexion getInstance() throws
SQLException {
    if (instance == null || instance.getConnection() == null ||
instance.getConnection().isClosed()) {
        instance = new BDConexion();
    }
      return instance;

}

//Método que cierra la conexión
public void close() {
    try {
        if (conn != null && !conn.isClosed())
            conn.close();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}


}
```

STATE: **package patronState;**



IAmbulanciaState.java:

Interfaz que define el contrato común para todos los estados posibles de la ambulancia. Establece qué acciones puede realizar una ambulancia y permite que cada estado concreto decida su comportamiento propio.

```
public interface IAmbulanciaState {
    boolean puedeAtencionDomicilio();
    void solicitarAtencionDomicilio();

    boolean puedeTrasladoClinica();
    void solicitarTrasladoClinica();

    boolean puedeRetornoAutomatico();
    void retornoAutomatico();

    boolean puedeMantenimiento();
    void solicitarMantenimiento();
}
```

Es implementada por todas las clases de estado concretas de la ambulancia, las cuales tienen un comportamiento bastante homólogo, por lo que solo se va a explicar una.

AmbulanciaStateDisponible.java:

Es una de las clases concretas que implementan la interfaz anterior. Todos estos State son los responsables de cambiar el estado de la ambulancia cada vez que sea necesario y de variar el comportamiento de esta según su estado actual.

```
public class AmbulanciaStateDisponible implements IAmbulanciaState {
//referencia a la clase de contexto
     //se forma una doble referencia entre contexto y estados
     private Ambulancia ambulancia;

//constructor que utiliza la calse del contexto
public AmbulanciaStateDisponible(Ambulancia ambulancia) {
this.ambulancia = ambulancia;
}

...

//los propios métodos del estado son los responsables de
```

```
//cambiar el estado del contexto
@Override
public void solicitarAtencionDomicilio() {
ambulancia.setEstado(new AmbulanciaStateAtendiendo(ambulancia));
}

...


//hay ocasiones en las que los métodos no deben hacer nada,
//este es el caso del retorno automático cuando la ambulancia
//ya está disponible
@Override
public void retornoAutomatico() {

}

...
//a lo ultimo se implementan métodos que sirven como banderas
//útiles para la concurrencia
}
```

package negocio;

Ambulancia.java:

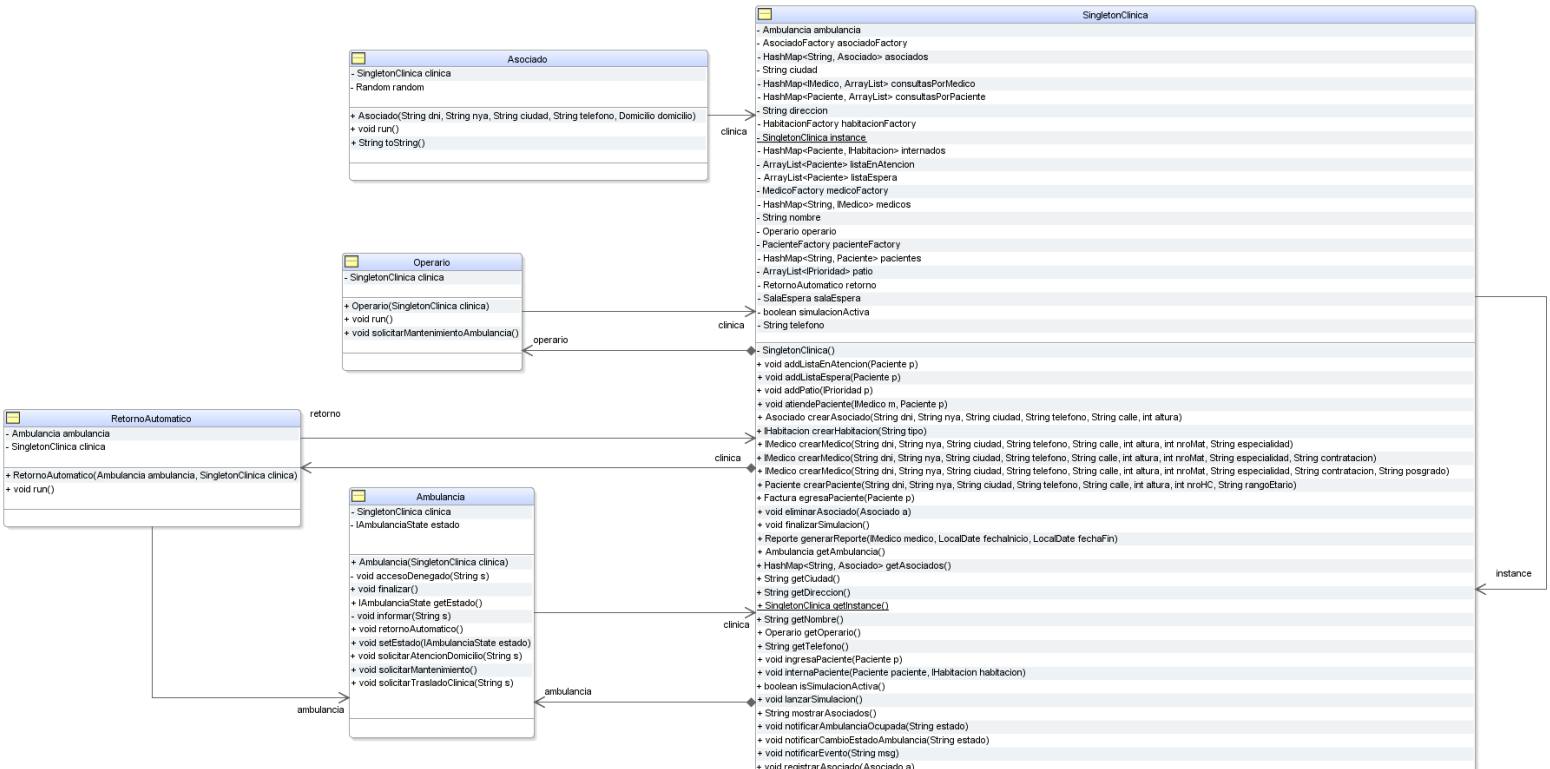
En el modelo se encuentra la clase de contexto del patrón State que representa a la ambulancia. Encapsula el estado actual y expone métodos sincronizados que representan las solicitudes externas (atención a domicilio, traslado a clínica, retorno automático, mantenimiento). Gestiona la espera de los solicitantes cuando la acción no es posible y notifica a los hilos cuando el estado cambia.

```
public class Ambulancia {
//doble referencia con el estado
private IAmbulanciaState estado;

//la clase contexto solo es responsable de inicializar
//el estado, el estado gestiona los cambios posteriores
public Ambulancia() {
this.estado = new AmbulanciaStateDisponible(this);
}
//la clase contexto igualmente implementa los métodos de
//las clases estado para delegarles la responsabilidad
```

```
public synchronized void solicitarAtencionDomicilio() {
//usa banderas en la clase de estado para definir
//la espera de los hilos en la concurrencia
while (!estado.puedeAtencionDomicilio()) {
informar(estado.toString());
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
          //finalmente delega la responsabilidad a su estado
estado.solicitarAtencionDomicilio();
//notifica a todos los hilos
notifyAll();
     }

...
//de manera homologa funcionan el resto de los métodos

//setEstado que usan las clases de estado para actualizarse
public void setEstado(IAmbulanciaState estado) {
this.estado = estado;
}
}
```

MVC:

MODELO: package negocio;



```
```

El modelo encapsula el estado de la aplicación.

Ambulancia.java:

Esta clase se explicó anteriormente.

Asociado.java:

Clase que representa a un asociado de la clínica dentro del modelo. Sus instancias funcionan como hilos participantes de la simulación, generando solicitudes de ambulancia. Hereda atributos de Persona e implementa Runnable.

```
...
@Override
public void run() {

    // El hilo del Asociado se mantiene activo mientras la
    //simulación siga activa.
    while (clinica.isSimulacionActiva()) {

        int accion = random.nextInt(2);
        // Se elige aleatoriamente qué tipo de servicio solicitar:
        // 0 = atención a domicilio, 1 = traslado a la clínica.

        switch (accion) {
            case 0:
                // El modelo avisa a la clínica que ocurrió un
                //evento.
                clinica.notificarEvento(getNya() + " pidió atención
                a domicilio");

                // Se interactúa con la ambulancia del modelo
               clinica.getAmbulancia().solicitarAtencionDomicilio(
               getNya());
```

```
                break;

      ...

         try {
             // Pausa entre solicitudes, simulando tiempo real.
             Thread.sleep(5000 + random.nextInt(2000));
         ...
    }
}
```

Operario.java:

Clase que representa al operario dentro del modelo. Solicita el mantenimiento de la ambulancia.

Implementa Runnable porque puede ejecutarse en un hilo breve, cuyo único trabajo es pedir mantenimiento y terminar.

```
...
public void solicitarMantenimientoAmbulancia() {
    // Llama al método sincronizado de la ambulancia.
    // Si la ambulancia está ocupada, este hilo puede quedar
    //bloqueado.
    clinica.getAmbulancia().solicitarMantenimiento();
}
@Override
public void run() {
    // El hilo del Operario ejecuta una única acción:
    // pedir que la ambulancia entre en mantenimiento.
    solicitarMantenimientoAmbulancia();
}
...
```

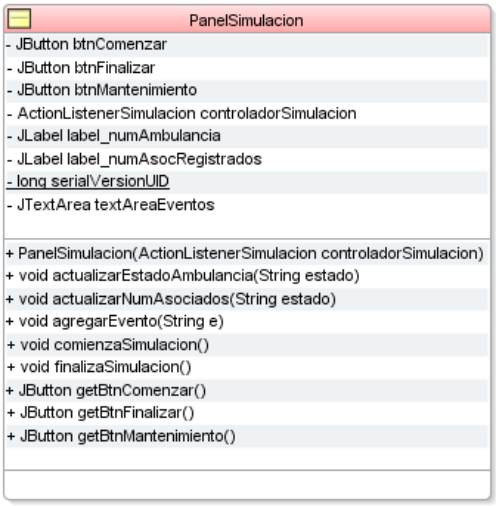
RetornoAutomatico.java:

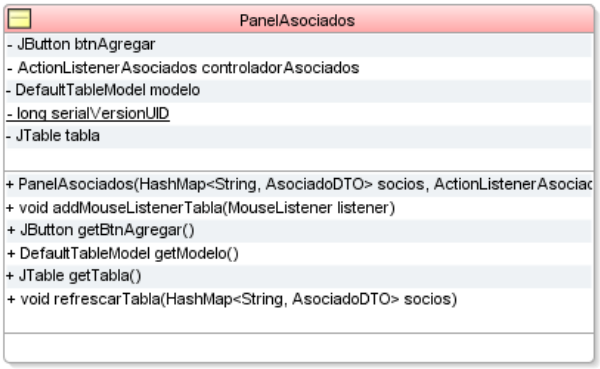
Clase que representa un hilo del sistema encargado de generar el evento de “Retorno Automático” de la ambulancia. Corre en un bucle mientras la simulación está activa y ejecuta retornoAutomatico() cada 5 segundos.

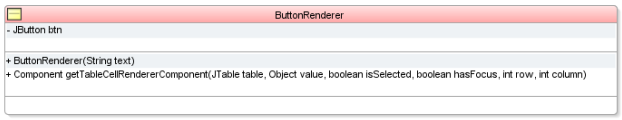
```
...
@Override
public void run() {
    while (clinica.isSimulacionActiva()) {
```

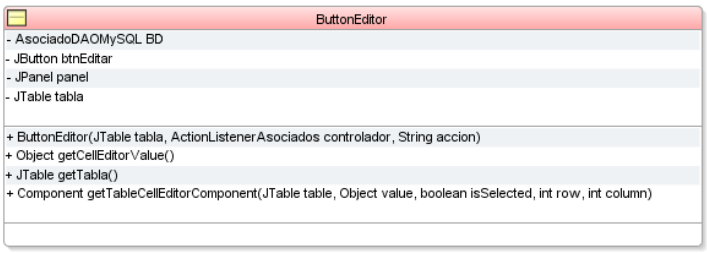
```
    // Mientras la simulación siga activa

        // Se registra el evento en el log del modelo
        clinica.notificarEvento("RETORNO: Retorno automático
         solicitado");


        // Llama al método sincronizado del modelo (puede
        //bloquearse)
        ambulancia.retornoAutomatico();

        try {
            Thread.sleep(5000);
            // Espera fija de 5 segundos entre cada evento
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

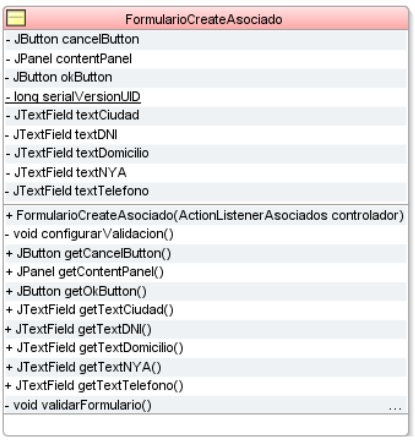
VISTA

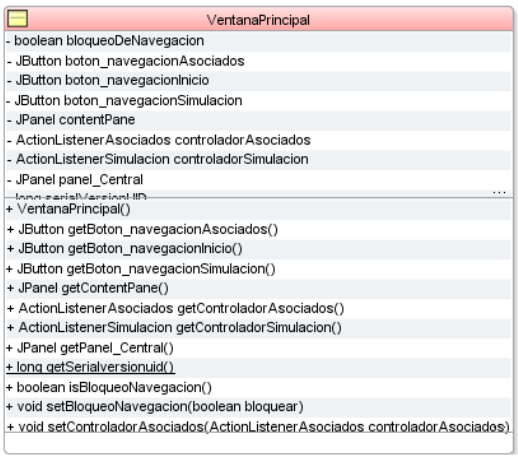


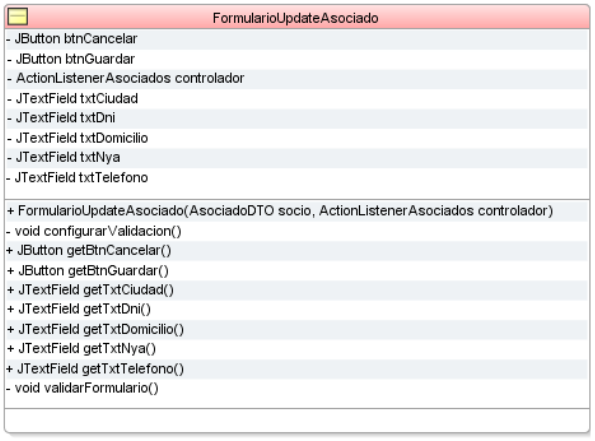












La vista es el conjunto de clases que muestran al usuario la información del modelo e interactúa con este.
Obtiene solo los datos que necesita del modelo y se actualiza cuando el modelo cambia. La vista envía los gestos del usuario al controlador.

package vista.JframePrincipal;

VentanaPrincipal.java:

VentanaPrincipal es la vista principal de la aplicación y actúa como el contenedor general de la parte gráfica. Organiza la interfaz en regiones (menú superior, barra de

herramientas, panel de navegación y panel central) y permite cambiar entre las distintas pantallas mediante un CardLayout.

Su responsabilidad principal es gestionar la estructura visual y delegar las acciones al controlador correspondiente. No contiene lógica de negocio ni modifica el modelo.

El panel central utiliza CardLayout para intercambiar vistas (Inicio, Asociados, Simulación). Cada botón de navegación dispara un ActionCommand que será recibido por los controladores configurados mediante los métodos setControladorAsociados y setControladorSimulacion.

La ventana mantiene referencias a los controladores para asignarlos dinámicamente y expone getters para que el controlador pueda acceder a elementos específicos de la vista si es necesario.

```
//tiene a ambos controladores como atributos
private ActionListenerAsociados controladorAsociados;
private ActionListenerSimulacion controladorSimulacion;
...

// Panel central con CardLayout: aquí se intercambian las distintas
//vistas
panel_Central = new JPanel(new CardLayout());
contentPane.add(panel_Central, BorderLayout.CENTER);
...

// Ejemplo de una vista añadida al CardLayout
panel_Central.add(new PanelSimulacion(controladorSimulacion),
"SIMULACION");
...

// Botón de navegación que delega acciones al controlador asignado
//previamente
boton_navegacionAsociados = new JButton("Asociados");
boton_navegacionAsociados.setActionCommand("ASOCIADOS");
...

// El controlador ya fue configurado mediante el método
//setControladorAsociados(),
// por eso aquí simplemente se lo agrega como listener
boton_navegacionAsociados.addActionListener(controladorAsociados);
...

// Método que recibe y asigna el controlador correspondiente
```

```
public void setControladorAsociados(ActionListenerAsociados
controladorAsociados) {
    this.controladorAsociados = controladorAsociados;
}
//ídem con controladorSimulacion
```

package vista.formularios;

ConfirmDialog.java:

Componente de la vista que muestra un cuadro de diálogo modal (obliga al usuario a tomar una decisión) para confirmar una acción del usuario. Presenta un mensaje y dos botones (“Sí” y “No”). Extiende JDialog (para ventanas secundarias).

Si el usuario confirma, envía un `ActionEvent` al controlador con el `actionCommand` y el `AsociadoDTO` recibido. No modifica el modelo; solo notifica al controlador y luego se cierra.

Recibe al controlador dentro de su constructor.

```
...
btnSi.addActionListener(e -> {
            listener.actionPerformed(
                new ActionEvent(socio, ActionEvent.ACTION_PERFORMED,
                 actionCommand);
            dispose(); //cerrar
});

// Acción del botón No
btnNo.addActionListener(e -> dispose());

...
```

FormularioCreateAsociado.java:

Componente de la vista que muestra un formulario modal para crear un nuevo asociado. Recibe al controlador en su constructor y resenta campos de texto para nombre, DNI, ciudad, teléfono y domicilio, junto con los botones “Guardar” y “Cancelar”. Extiende JDialog.

Al presionar “Guardar”, crea un `AsociadoDTO` con los datos ingresados y envía un `ActionEvent` al controlador con el comando `"CREATE"`. Captura la entrada del

usuario y notifica al controlador. Si se presiona “Cancelar”, el formulario se cierra sin realizar acciones.

El botón Guardar está inicialmente desactivado y se habilita automáticamente sólo cuando la validación de los campos (no vacíos, DNI y teléfono numéricos, DNI > 0) es correcta.

```
...

okButton = new JButton("Guardar");

...

okButton.addActionListener(e -> {
    AsociadoDTO socio = new AsociadoDTO();
    socio.setDni(textDNI.getText());
    socio.setNya(textNYA.getText());
    socio.setCiudad(textCiudad.getText());
    socio.setTelefono(textTelefono.getText());
    socio.setDomicilioStr(textDomicilio.getText());
    controlador.actionPerformed(new ActionEvent(socio, 0,
    "CREATE"));
    dispose();
});

cancelButton.addActionListener(e -> dispose());

...

okButton.setEnabled(false); // inicialmente desactivado

...

private void validarFormulario() {
        boolean datosValidos = true;
   ...
         okButton.setEnabled(datosValidos);
        //el botón se habilita (o no, según condición)
    }
```

FormularioUpdateAsociado.java:

Componente de la vista que muestra un formulario modal para editar un asociado existente. Tiene un atributo que referencia al controlador que se pasa por el constructor.

Recibe un `AsociadoDTO` y precarga sus datos en los campos; el DNI permanece bloqueado porque es inmutable. Presenta los campos de edición y los botones “Guardar” y “Cancelar”. Extiende JDialog.

Al presionar “Guardar”, actualiza el DTO original con los nuevos valores y envía un `ActionEvent` al controlador con el comando `"UPDATE"`. Si se presiona “Cancelar”, el formulario se cierra sin realizar acciones.

```
...

btnGuardar.addActionListener(e -> {
    socio.setNya(txtNya.getText());
    socio.setCiudad(txtCiudad.getText());
    socio.setTelefono(txtTelefono.getText());
    socio.setDomicilioStr(txtDomicilio.getText());
    controlador.actionPerformed(new ActionEvent(socio, 0,
    "UPDATE"));
    dispose();
});

btnCancelar.addActionListener(e -> dispose());

...
```

También habilita o deshabilita el botón de guardar con un `validarFormulario()`, puede ser incovado por `configurarValidacion()` ante cualquier cambio.

package vista.PanelCentral;

Panel_Inicio.java:

Componente de la vista que representa el panel inicial o pantalla de bienvenida de la aplicación. Extiende JPanel y no contiene lógica ni interacción directa con el usuario. Funciona como panel base.

PanelAsociados.java

Panel de la vista que muestra la gestión de asociados mediante una tabla y un botón para agregar nuevos. Entre sus atributos se encuentra el controlador, extiende JPanel y

organiza sus componentes con BorderLayout. Muestra los datos iniciales recibidos en un HashMap<String,AsociadoDTO>.

La tabla incluye columnas con botones “Editar” y “Eliminar”, renderizados mediante ButtonRenderer y manejados por ButtonEditor, que envían eventos al controlador con los comandos correspondientes.

```
...
tabla.getColumn("Editar").setCellRenderer(new
ButtonRenderer("Editar"));
tabla.getColumn("Editar").setCellEditor(
    new ButtonEditor(tabla, controladorAsociados, "SELECT_UPDATE")
);

tabla.getColumn("Eliminar").setCellRenderer(new
ButtonRenderer("X"));
tabla.getColumn("Eliminar").setCellEditor(
    new ButtonEditor(tabla, controladorAsociados, "SELECT_DELETE")
);
...
```

PanelSimulacion.java

Panel principal de la vista para la interfaz de simulación. Muestra el estado de la ambulancia, el número de asociados registrados, un log de eventos (JTextArea) y los botones “Comenzar”, “Mantenimiento” y “Finalizar”. Es pasivo: delega el manejo de acciones al ActionListenerSimulacion pasado en el constructor y expone métodos para actualizar la interfaz de usuario desde el controlador.

```
...
// Aquí se indica qué comando envía cada botón y se registra el
//listener externo.
btnComenzar.setActionCommand("INICIAR_SIM");
btnMantenimiento.setActionCommand("MANTENIMIENTO");
btnFinalizar.setActionCommand("FINALIZAR_SIMULACION");

btnComenzar.addActionListener(controladorSimulacion);
btnMantenimiento.addActionListener(controladorSimulacion);
btnFinalizar.addActionListener(controladorSimulacion);
...
// Log de eventos: método público para que el controlador añada
//entradas al JTextArea
public void agregarEvento(String e) {
```

```
    String tiempo =
java.time.LocalTime.now().withNano(0).toString();
    textAreaEventos.append("[" + tiempo + "] " + e + "\n");
}
...
```

También habilita y deshabilita los botones “Comenzar”, “Finalizar” y “Mantenimiento” mediante `comienzaSimulacion()` y `finalizaSimulacion()`.

package vista.RendererAsociados;

ButtonEditor.java

Clase de la vista que actúa como editor de celda para los botones dentro de la JTable de asociados.

Permite capturar el clic del usuario en una fila, reconstruir un AsociadoDTO con los datos visibles y enviar un ActionEvent al controlador (ActionListenerAsociados) con el comando correspondiente.

Integra la vista con el módulo de asociados, sirviendo como punto de disparo de las acciones "editar" o "eliminar" desde la tabla.

```
btnEditar.addActionListener(e -> {
    //Se obtiene la fila donde se hizo clic
    int fila = tabla.getSelectedRow();

    //Se reconstruye un DTO con los datos visibles en la fila
    if (fila >= 0) {
        AsociadoDTO dto = new AsociadoDTO();
        dto.setNya((String) tabla.getValueAt(fila, 0));
        dto.setDni((String) tabla.getValueAt(fila, 1));

        //Se envía un ActionEvent al controlador con el DTO y el
        //actionCommand (accion)
        controlador.actionPerformed(
            new ActionEvent(dto, ActionEvent.ACTION_PERFORMED,
            accion)
        );
        // La vista solo notifica y delega.
    }
});
```

ButtonRenderer.java

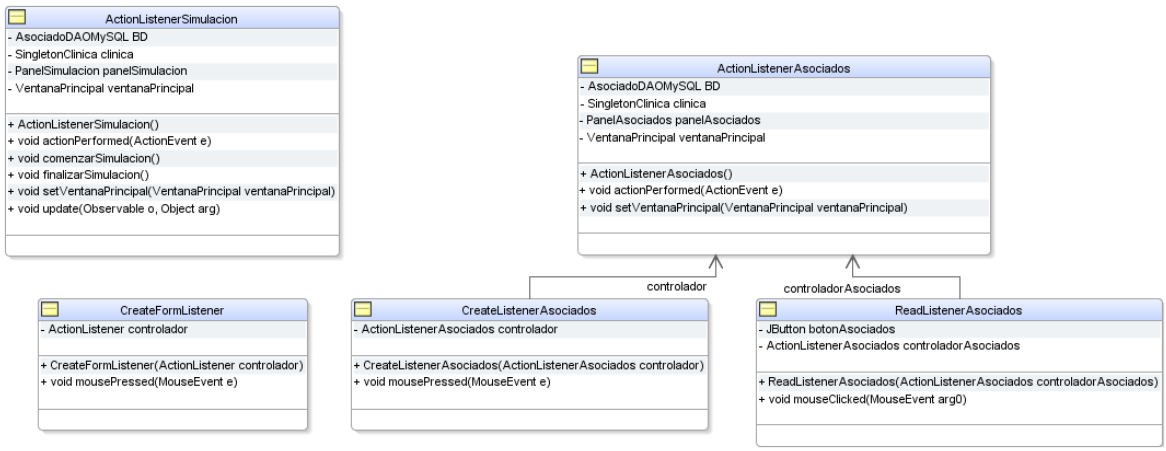
Renderizador de celdas de la vista que dibuja un botón dentro de la JTable de asociados. Define únicamente la apariencia visual del botón, no maneja clics ni acciones.

Trabaja junto al ButtonEditor: este renderer “muestra” el botón y el editor maneja la interacción.

No interactúa con el modelo ni con el controlador.

```
@Override
public Component getTableCellRendererComponent(JTable table, Object
value,
        boolean isSelected, boolean hasFocus, int row, int column) {

    // La JTable usa este panel (que contiene el JButton) solo para
    //"pintar" la celda.
    // No hay interacción: es puramente visual.
    return this;
}
```

CONTROLADOR:



package controlador.Asociados;

ActionListenerAsociados.java

Controlador responsable de gestionar las acciones del módulo de asociados. Escucha eventos de la vista (botones y formularios), ejecuta operaciones sobre la capa de persistencia (AsociadoDAOMySQL) y actualiza las vistas (por ejemplo, refrescando la tabla de PanelAsociados o mostrando formularios/diálogos). También integra nuevos Asociado en el modelo de simulación (SingletonClinica) cuando corresponde.

```
//Registro de la vista en el controlador y asignación del
//controlador a la vista
public void setVentanaPrincipal(VentanaPrincipal ventanaPrincipal) {
    this.ventanaPrincipal = ventanaPrincipal;

    // la vista guarda la referencia al controlador
    this.ventanaPrincipal.setControladorAsociados(this);

    this.ventanaPrincipal.setVisible(true);

    // Registrar listeners para navegación y listado
    this.ventanaPrincipal.getBoton_navegacionAsociados()
    .addMouseListener(new ReadListenerAsociados(this));
    this.ventanaPrincipal.getBoton_navegacionInicio().
     addActionListener(this);
}

//Método que centraliza la lógica al recibir eventos de la vista
@Override
public void actionPerformed(ActionEvent e) {
    String comando = e.getActionCommand().toUpperCase();
      ...

    //ignora acciones si la navegación está bloqueada por la
    //simulación (simulación ya activa)
    if (ventanaPrincipal.isBloqueoNavegacion()) return;

    switch(comando) {


      case "INICIO":
      // Se crea un panel simple con un mensaje de bienvenida
           // No interactúa con el modelo, solo con la vista
           JPanel panelInicio = new JPanel();
           panelInicio.add(new JLabel("Bienvenido al sistema"));
           panelCentral = ventanaPrincipal.getPanel_Central();

           // Se agrega el panel al CardLayout y se muestra
           panelCentral.add(panelInicio, "PANEL_INICIO");
           cl = (CardLayout) panelCentral.getLayout();
           cl.show(panelCentral, "PANEL_INICIO");
```

```
            panelCentral.revalidate();
            panelCentral.repaint();
            break;

        case "CREATE":
            //usuario hizo click en "guardar" en el formulario de
            //creacion

            //recibe DTO desde el FormularioCreateAsociado
            AsociadoDTO nuevoSocio = (AsociadoDTO)e.getSource();

            //guarda en base de datos
 ...
            BD.agregar(nuevoSocio);
 ...

            //actualiza la tabla mostrada en PanelAsociados
            HashMap<String, AsociadoDTO> asociados =
            BD.obtenerTodosMap();
            panelAsociados.refrescarTabla(asociados);


           // También se crea un objeto Asociado para la
           //simulación
            d = new Domicilio(nuevoSocio.getDomicilioStr(), 0);
            actual = new Asociado(nuevoSocio.getDni(),
            nuevoSocio.getNya(), nuevoSocio.getCiudad(),
            nuevoSocio.getTelefono(), d);
            ...
// Registro en el singleton de la clínica
            this.clinica.registrarAsociado(actual);
            ...

            break;

        case "READ":
            //usuario pide ver la lista de asociados
            // Obtiene todos los asociados desde la BD
            //y los muestra en la vista
            asociados = BD.obtenerTodosMap();

// Crea un panel que representa la vista de asociados y
```

```
          //lo agrega al CardLayout
            panelAsociados = new PanelAsociados(asociados, this);
            panelCentral = ventanaPrincipal.getPanel_Central();
            panelCentral.add(panelAsociados, "PANEL_ASOCIADOS");
           cl = (CardLayout) panelCentral.getLayout();
cl.show(panelCentral, "PANEL_ASOCIADOS");

           // Se agrega listener para el botón de agregar nuevo
           //asociado
panelAsociados.getBtnAgregar().addMouseListener(new
           CreateListenerAsociados(this));
break;

       case "UPDATE":
      // Actualiza un asociado existente
AsociadoDTO socio = (AsociadoDTO) e.getSource();
...
// Actualiza en la BD
BD.actualizar(socio);
...

// Actualiza la tabla en la vista con datos reciente
asociados = BD.obtenerTodosMap();
panelAsociados.refrescarTabla(asociados);

// También se actualiza el objeto en memoria
d = new Domicilio(socio.getDomicilioStr(),0);
actual = new Asociado(socio.getDni(),socio.getNya(),
           socio.getCiudad(),socio.getTelefono(),d);

break;
case "DELETE":
//Elimina un asociado
AsociadoDTO socioEliminado = (AsociadoDTO)e.getSource();
...
           BD.eliminarPorDni(socioEliminado.getDni());
...

// Refresca la tabla de la vista
asociados = BD.obtenerTodosMap();
panelAsociados.refrescarTabla(asociados);
break;
```

```
case "SELECT_UPDATE":
     // Abre formulario de actualización
     // No toca el modelo (BD), solo prepara la vista con datos
     //actuales
FormularioUpdateAsociado form = new
          FormularioUpdateAsociado((AsociadoDTO)e.getSource(),this)
          ;
form.setLocationRelativeTo(null);
      form.setVisible(true);

break;
case "SELECT_DELETE":
          //Abre confirmación de eliminación (vista)
          // No toca el modelo hasta que el usuario confirme
ConfirmDialog popUpEliminar = new ConfirmDialog(null,
"¿Está seguro que desea eliminar este asociado?",
"DELETE",
this,
(AsociadoDTO)e.getSource());
popUpEliminar.setVisible(true);
break;
}
}

}
```

CreateFormListener.java:

Este listener se encarga de capturar el evento de “guardar” desde un formulario de creación o actualización de asociados. Su rol principal es traducir la información de la vista (el formulario) en un objeto del modelo (AsociadoDTO) y luego enviarlo al controlador, que hará la operación correspondiente (crear el asociado en la BD y actualizar la vista).

```
@Override
public void mousePressed(MouseEvent e) {
    //Crea un DTO con los datos del formulario
    AsociadoDTO asociado = new AsociadoDTO();
    FormularioUpdateAsociado formulario =(FormularioUpdateAsociado)
    e.getSource();
    asociado.setCiudad(formulario.getTxtCiudad().getText());
```

```
    asociado.setDni(formulario.getTxtDni().getText());

  asociado.setDomicilioStr(formulario.getTxtDomicilio().getText());
    asociado.setNya(formulario.getTxtNya().getText());
    asociado.setTelefono(formulario.getTxtTelefono().getText());

    //Solo se responde al botón izquierdo del mouse
    if (SwingUtilities.isLeftMouseButton(e)) {
        //Crea un evento de tipo CREATE y lo envía al controlador
        ActionEvent evento = new ActionEvent(asociado, 0,"CREATE");
        controlador.actionPerformed(evento);
    }
}
```

CreateListenerAsociados.java:

Este listener se encarga de abrir un formulario para crear un nuevo asociado cuando el usuario hace clic en el botón correspondiente en la vista de asociados. No modifica datos directamente ni toca la base de datos, solo invoca la vista de creación y pasa el controlador que se encargará de manejar la acción de “guardar”.

```
@Override
public void mousePressed(MouseEvent e) {
    // Sólo responde al botón izquierdo del mouse
    if (SwingUtilities.isLeftMouseButton(e)) {
        // Crea el formulario de creación de asociado
        FormularioCreateAsociado form = new
FormularioCreateAsociado(this.controlador);

        // Centra el formulario y lo muestra
        form.setLocationRelativeTo(null);
        form.setVisible(true);
    }
}
```

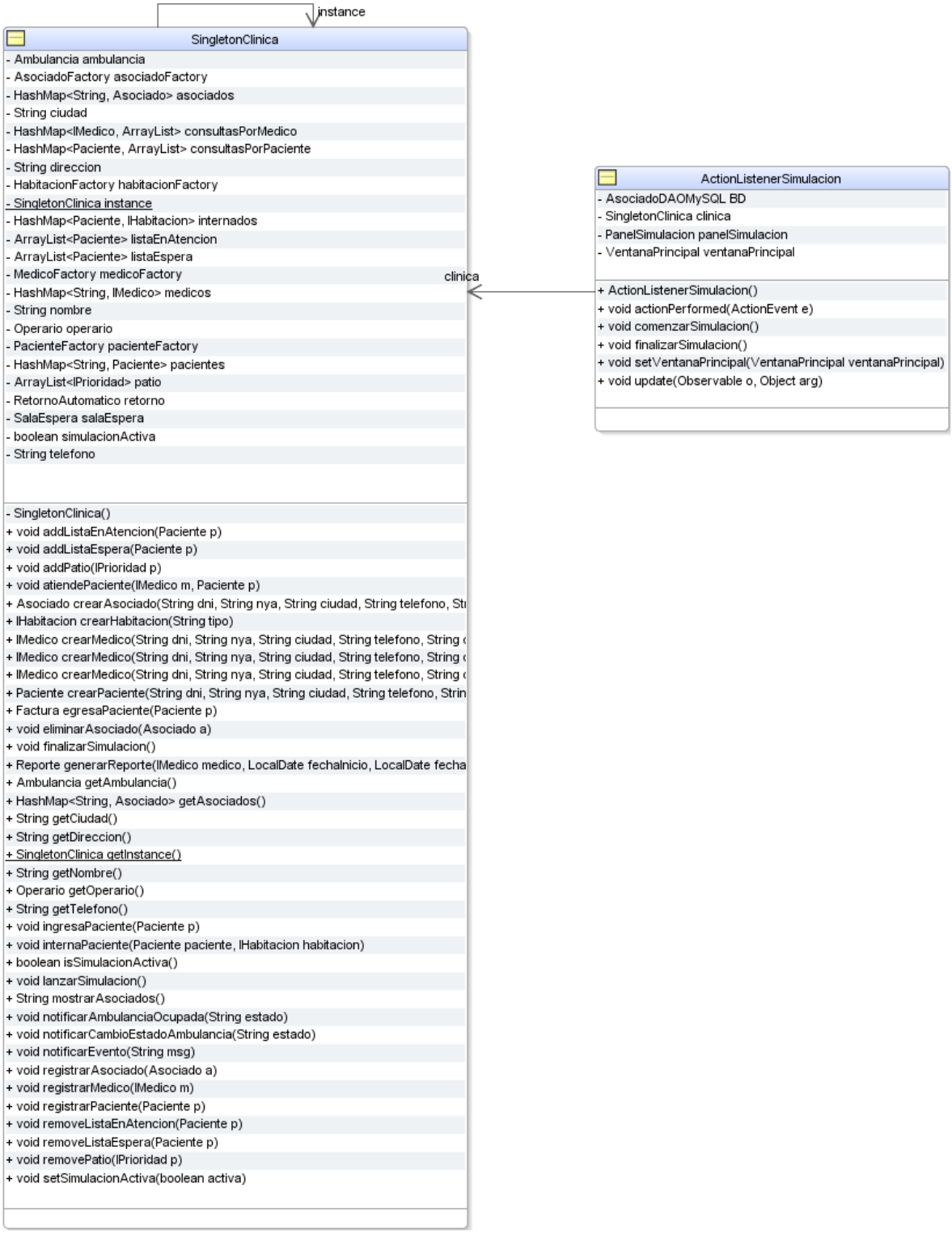
ReadListenerAsociados.java:

Este listener se encarga de detectar el clic del usuario en el botón de “Asociados” en la ventana principal y notificar al controlador que debe mostrar la lista de asociados en el panel central. No manipula la base de datos ni los datos directamente, su función es traducir el clic en un evento que el controlador pueda procesar.

```
@Override
public void mouseClicked(MouseEvent arg0) {
```

```
    // Crea un ActionEvent con comando "READ" que indica al
    //controlador que debe mostrar la lista de asociados
    ActionEvent e = new ActionEvent(arg0.getSource(), 0, "READ");

    // Notifica al controlador de asociados para que ejecute la
    //acción correspondiente
    this.controladorAsociados.actionPerformed(e);
}
```

OBSERVER/CONTROLADOR:



package controlador.Simulacion;

ActionListenerSimulacion.java:

Esta clase conecta la Vista (VentanaPrincipal y PanelSimulacion) y el Modelo (SingletonClinica), delega operaciones al modelo y mantiene la vista sincronizada mediante el patrón Observer, reflejando eventos y cambios de estado de forma automática.}

```
public ActionListenerSimulacion() {
//Obtiene la instancia del modelo
this.clinica = SingletonClinica.getInstance();
...
//Es el responsable de añadirse como observador de la
//clinica en el constructor
clinica.addObserver(this);

// Carga los asociados desde la base de datos y los registra en el
//modelo para mantener la vista actualizada
//convierte DTO a Asociado
HashMap<String,AsociadoDTO> asociados = BD.obtenerTodosMap();
for(AsociadoDTO a: asociados.values()) {
    Domicilio d = new Domicilio(a.getDomicilioStr(),0);
    Asociado actual = new
Asociado(a.getDni(),a.getNya(),a.getCiudad(),a.getTelefono(),d);
...
    clinica.registrarAsociado(actual);
}
}
```

En el método actionPerformed, gestiona comandos de la vista, delegando al modelo o modificando la interfaz. Por ejemplo, mostrar el panel de simulación, iniciar o finalizar la simulación y solicitar mantenimiento:

```
@Override
public void actionPerformed(ActionEvent e) {
...

case "INICIAR_SIM":
          if (!clinica.isSimulacionActiva())
               comenzarSimulacion();
          ventanaPrincipal.setBloqueoNavegacion(true);
          break;

     case "MANTENIMIENTO":
          if (clinica.isSimulacionActiva()) {
          Thread operario = new Thread(clinica.getOperario());
```

```
            operario.start();
            panelSimulacion.agregarEvento("OPERARIO: Solicitud de
            mantenimiento");
            }
            break;
...
}
}
```

Los métodos comenzarSimulacion y finalizarSimulacion coordinan la activación y detención de los hilos en el modelo y actualizan la vista con mensajes y estados de la ambulancia.

```
public void comenzarSimulacion() {
    panelSimulacion.agregarEvento("Simulación iniciada.\n");
...
    panelSimulacion.comienzaSimulacion();
    clinica.lanzarSimulacion(); // Modelo inicia los hilos
}

public void finalizarSimulacion() {
    panelSimulacion.agregarEvento("Simulación finalizada.\n");
    panelSimulacion.actualizarEstadoAmbulancia("  --");
...
    panelSimulacion.finalizaSimulacion();
    clinica.finalizarSimulacion(); // Modelo detiene los hilos
}
```

Como **Observer**, el método update recibe notificaciones del modelo y refleja los cambios en la vista, separando eventos de estado de la ambulancia de los eventos generales:

```
@Override
public void update(Observable o, Object arg) {
    if((SingletonClinica)o == clinica) {
        String msg = (String) arg;
        if(msg.startsWith("ESTADO:")) {
            String estado = msg.substring("ESTADO:".length());
            panelSimulacion.actualizarEstadoAmbulancia(estado);
        } else {
            panelSimulacion.agregarEvento(msg);
        }
    }
```

```
}
```

OBSERVABLE: **package clinica;**

SingletonClinica.java:

Esta clase extiende Observable, implementando el patrón Observer. Implementa los siguientes métodos para notificar a su Observer de los cambios:

```
public void notificarAmbulanciaOcupada(String estado) {
    setChanged();
    notifyObservers("RAZÓN:" + estado);
}

public void notificarCambioEstadoAmbulancia(String estado) {
    setChanged();
    notifyObservers("ESTADO: " + estado);
}

public void notificarEvento(String msg) {
    setChanged();
    notifyObservers(msg);
}
```