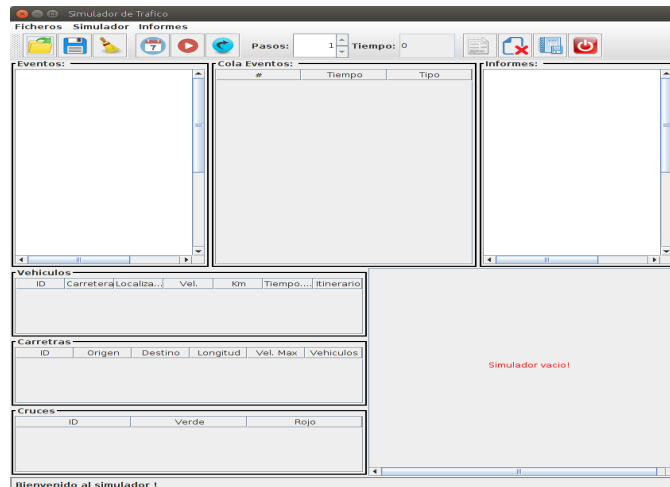


## Práctica 5: Simulador de tráfico (GUI)

Puri Arenas Sánchez  
Facultad de Informática (UCM)

### La GUI



## El diseño

■ Diseñamos una GUI para el Simulador utilizando el modelo vista-controlador.

■ El modelo (SimuladorTrafico) simplemente incorpora una lista de observadores (las vistas) para avisarles de posibles cambios.

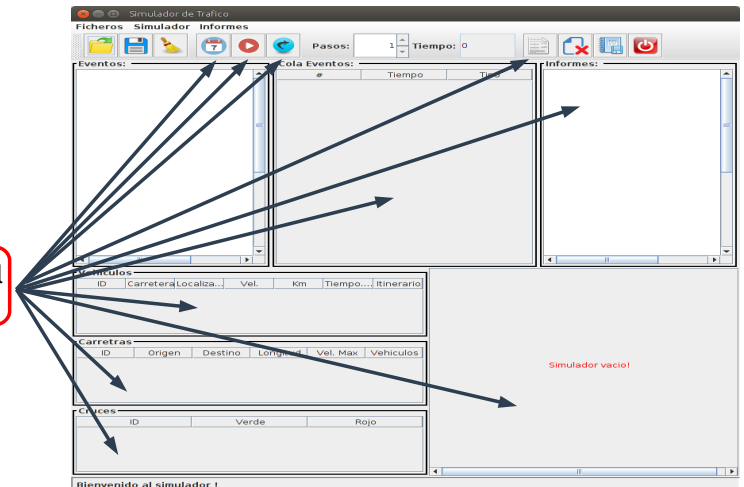
■ El controlador interactúa con las vistas. Las vistas se añaden como observadores a través del modelo. Además avisan al controlador cuando se produce algún evento en sus componentes.

■ Para diseñar la vista vamos a hacer que aquellas componentes que dependan del modelo se registren como observadoras. El resto de componentes que dependen de la ventana principal serán controladas por la propia ventana.

2

### La GUI

Dependen del modelo



## La GUI: El menú

- El **menú Ficheros** no depende del modelo:
  - **Carga Eventos**: Lee un fichero de texto y lo coloca en el área de texto “Eventos:”.
  - **Salva Eventos**: Carga el contenido del área de texto “Eventos:” en un fichero.
  - **Salva Informe**: Salga el contenido del área de texto “Informes:” en un fichero.
  - **Salir**: cierra la aplicación.
- El **menú Simulador** depende del modelo (**Ejecuta y Reinicia**).
- El **menú Informes** no depende del simulador (**Generar y Limpiar**).
- Por tanto el menú necesita el **controlador**, pero no tiene que registrarse como observador, ya que no sufre ninguna modificación tras la ejecución del modelo.

5

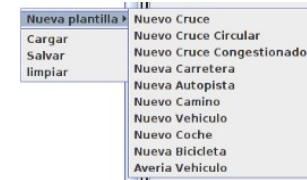
## La GUI: barra de herramientas

- La **barra de herramientas** ofrece funcionalidades similares a las del menú.
- Tiene componentes para mostrar el tiempo por el que va el simulador (**TextField**), y el número de pasos que deben ejecutarse de una vez (**Spinner**).
- El tiempo debe pasárselo el modelo.
- También el modelo debe indicar a la barra de herramientas que el simulador se ha reiniciado, para que actualice las componentes de “tiempo” y “pasos”.
- Se puede optar por:
  - Hacer que la barra de herramientas sea un *observador de modelo*.
  - Separar el **Spinner** y el **campo de texto** como componentes separadas que aparecen en la barra de herramientas, y registrar sólo estas dos componentes como observadores del modelo.

7

## La GUI: El pop-up menú

- El **pop-up menu** no depende del modelo
  - **Nueva plantilla**: Permite introducir en el área de texto “Eventos” un nuevo evento en formato texto, para lo cual suministra la plantilla correspondiente. La plantilla se escribe en el área de texto en la posición que previamente se haya seleccionado con el ratón.



- **Cargar y Salvar**: son opciones similares a las del menú Ficheros.
- **Limpiar**: vacía el área de texto correspondiente a los eventos.

6

## El modelo (Simulador de Tráfico)

- Nuestro modelo tiene como componentes principales: el paso de la simulación (**tiempo**), **los eventos**, y los **objetos de la simulación (mapa)**. Por lo tanto debe tener que ser capaz de notificar cambios en estas entidades.
- Los cambios se producen cuando:
  - La simulación recibe los eventos que le manda el controlador.
  - Cuando la simulación da un paso y se ejecutan los eventos correspondientes a ese tiempo, que crearán nuevos objetos de la simulación. Después los objetos de la simulación, almacenados en el mapa, avanzan.
  - Cuando se produce un error en la simulación.
  - Cuando la simulación se reinicia.

8

## El modelo (Simulador de Tráfico)

- Necesitamos entonces la siguiente interfaz:

```
public interface ObservadorSimuladorTrafico {
    // notifica errores
    public void errorSimulador(int tiempo, MapaCarreteras mapa,
        List<Evento> eventos, ErrorDeSimulacion e);
    // notifica el avance de los objetos de simulación
    public void avanza(int tiempo, MapaCarreteras mapa,
        List<Evento> eventos);
    // notifica que se ha generado un nuevo evento
    public void addEvento(int tiempo, MapaCarreteras mapa,
        List<Evento> eventos);
    // notifica que la simulación se ha reiniciado
    public void reinicia(int tiempo, MapaCarreteras mapa,
        List<Evento> eventos);
}
```

9

## El modelo (Simulador de Tráfico)

El modelo llevará una lista de observadores, con métodos para poder añadirlos y eliminarlos. De esta forma las vistas podrán registrarme como observadores dentro del simulador.

```
public class SimuladorTrafico implements
    Observador<ObservadorSimuladorTrafico> {
    ...
    private List<ObservadorSimuladorTrafico> observadores;
    ...
}

public interface Observador<T> {
    public void addObservador(T o);
    public void removeObservador(T o);
}
```

10

## El modelo (Simulador de Tráfico)

```
public class SimuladorTrafico implements
    Observador<ObservadorSimuladorTrafico> {
    ...
    private List<ObservadorSimuladorTrafico> observadores;
    ...
    @Override
    public void addObservador(ObservadorSimuladorTrafico o) {
        if (o != null && !this.observadores.contains(o)) {
            this.observadores.add(o);
        }
    }
    @Override
    public void removeObservador(ObservadorSimuladorTrafico o) {
        if (o != null && this.observadores.contains(o)) {
            this.observadores.remove(o);
        }
    }
}
```

11

## El modelo (Simulador de Tráfico)

- La lista de observadores se inicializa a la lista vacía en la constructora.

```
this.observadores = new ArrayList<>();
```

- Cuando se inserta un evento **e**, hay que notificar que el evento se ha insertado, o bien que se ha producido un error.

```
this.notificaNuevoEvento();
this.notificaError(e);
```

- Cuando se reinicia el simulador.

```
this.notificaReinicia();
```

- Cuando se da un paso de simulación, hay que notificar que el estado ha avanzado o bien que se ha producido un error.

```
this.notificaAvanza();
this.notificaError(e);
```

12

## El modelo (Simulador de Tráfico)

```
public void insertaEvento(Evento e) {
    if (e != null) {
        if (e.getTiempo() < this.contadorTiempo) {
            ErrorDeSimulacion err = new ErrorDeSimulacion(...);
            this.notificaError(err);
            throw err;
        } else {
            this.eventos.add(e);
            this.notificaNuevoEvento(); // se notifica a los observadores
        }
    } else {
        ErrorDeSimulacion err = new ErrorDeSimulacion(...);
        this.notificaError(err); // se notifica a los observadores
        throw err;
    }
}

private void notificaNuevoEvento() {
    for (ObservadorSimuladorTrafico o : this.observadores) {
        o.addEvento(this.contadorTiempo, this.mapa, this.eventos);
    }
}
```

13

## La clase Main

- Tiene que admitir ambos modos: Consola y gráfico.
- Puedes utilizar un tipo enumerado para distinguir el modo que se va a utilizar:

```
private enum ModoEjecucion {
    BATCH("batch"), GUI("gui");

    private String descModo;

    private ModoEjecucion(String modeDesc) {
        descModo = modeDesc;
    }

    private String getModelDesc() {
        return descModo;
    }
}
```

15

## El controlador

- Añadimos un nuevo método que permite ejecutar la simulación un número determinado de pasos.

```
public void ejecuta(int pasos) {
    this.simulador.ejecuta(pasos, this.ficheroSalida);
}
```

- Añadimos un nuevo método para reiniciar el simulador.

```
public void reinicia() { this.simulador.reinicia(); }
```

- El simulador tendrá un método reinicia, que reinicia todos sus atributos y notifica a los observadores dicha acción.

- Introducimos dos nuevos métodos que permiten añadir y eliminar observadores del simulador.

```
public void addObserver(ObservadorSimuladorTrafico o) {
    this.simulador.addObserver(o);
}
...
```

14

## La clase Main

- Debes ampliar el método “ParseaArgumentos” para que acepte ahora los dos modos: **batch** y **gui**. Para ello añade un nuevo método estático “parseaOpcionModo(CommandLine)”, que se implementa de forma similar al resto de métodos.
- También el método “construyeOpciones()” debe soportar ahora las nuevas opciones.
- Estos son ejemplos de las opciones que debe aceptar la práctica:

```
-i resources/examples/events/basic/ex1.ini
-i resources/examples/events/advanced/ex1.ini -t 100
-m batch -i resources/examples/events/basic/ex1.ini -t 20
-m batch -i resources/examples/events/advanced/ex1.ini
-m gui -i resources/examples/events/basic/ex1.ini
-m gui -i resources/examples/events/advanced/ex1.ini
--help
```

16

## La clase Main

Para iniciar el modo gráfico puedes usar el método:

```
private static void iniciaModoGrafico() throws FileNotFoundException,
    InvocationTargetException, InterruptedException {
    SimuladorTrafico sim = new SimuladorTrafico();
    OutputStream os = Main.ficheroSalida == null ?
        System.out : new FileOutputStream(new File(Main.ficheroSalida));
    Controlador ctrl = new Controlador(sim, Main.limiteTiempo, null, os);

    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            try {
                new VentanaPrincipal(Main.ficheroEntrada, ctrl);
            } catch (FileNotFoundException e) {e.printStackTrace();}
        }
    });
}
```

17

## La GUI: Ventana principal

- La registraremos como observadora, pero simplemente para que *controle los posibles errores* e informe al usuario de dichos errores.
- La ventana principal podría no ser observadora si se hace un tratamiento de excepciones detallado, de forma que cada componente pueda reaccionar ante el error de forma correcta. En nuestro diseño tendremos varios observadores, concretamente aquellas componentes que dependan del modelo.

```
public class VentanaPrincipal extends JFrame
    implements ObservadorSimuladorTrafico {

    private Controlador controlador;
    ...
}
```

18

## Ventana Principal: Atributos

```
public static Border bordePorDefecto =
    BorderFactory.createLineBorder(Color.black, 2);
// SUPERIOR PANEL
static private final String[] columnIdEventos =
    { "#", "Tiempo", "Tipo" };

private PanelAreaTexto panelEditorEventos;
private PanelAreaTexto panelInformes;
private PanelTabla<Evento> panelColaEventos;

// MENU AND TOOL BAR
private JFileChooser fc;
private ToolBar toolbar;

// GRAPHIC PANEL
private ComponenteMapa componenteMapa;

// STATUS BAR (INFO AT THE BOTTOM OF THE WINDOW)
private PanelBarraEstado panelBarraEstado;
```

19

## Ventana Principal: Atributos

```
// INFERIOR PANEL
static private final String[] columnIdVehiculo = { "ID", "Carretera",
    "Localizacion", "Vel.", "Km", "Tiempo. Ave.", "Itinerario" };
static private final String[] columnIdCarretera = { "ID", "Origen",
    "Destino", "Longitud", "Vel. Max", "Vehiculos" };
static private final String[] columnIdCruce = { "ID", "Verde", "Rojo" };

private PanelTabla<Vehiculo> panelVehiculos;
private PanelTabla<Carretera> panelCarreteras;
private PanelTabla<CruceGenerico<?>> panelCruces;

// REPORT DIALOG
private DialogoInformes dialogoInformes; // opcional

// MODEL PART - VIEW CONTROLLER MODEL
private File ficheroActual;
private Controlador controlador;
```

20

## Ventana principal: constructora

```
public VentanaPrincipal(String ficheroEntrada, Controlador
ctrl)
    throws FileNotFoundException {
    super("Simulador de Trafico");
    this.controlador = ctrl;
    this.ficheroActual = ficheroEntrada != null ?
        new File(ficheroEntrada) : null;
    this.initGUI();
    // añadimos la ventana principal como observadora
    ctrl.addObserver(this);
}
```

21

## Ventana principal: constructora

```
private void initGUI() {
    ...
    // PANEL QUE CONTIENE EL RESTO DE COMPONENTES
    // (Lo dividimos en dos paneles (superior e inferior)
    JPanel panelCentral = this.createPanelCentral();
    panelPrincipal.add(panelCentral, BorderLayout.CENTER);
    ...
}
```

El panel central contendrá dos paneles: El superior, compuesto de dos áreas de texto y una tabla, y el inferior, compuesto por tres tablas y una zona de gráficos.

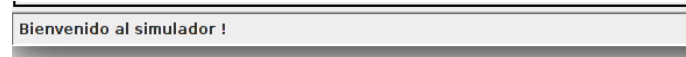
```
private JPanel createPanelCentral() {
    JPanel panelCentral = new JPanel();
    // para colocar el panel superior e inferior
    panelCentral.setLayout(new GridLayout(2,1));
    return panelCentral;
}
```

23

## Ventana principal: constructora

```
private void initGUI() {
    this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    this.addWindowListener(new WindowListener() {
        // al salir pide confirmación
    });
    JPanel panelPrincipal = this.creaPanelPrincipal();
    this.setContentPane(panelPrincipal);

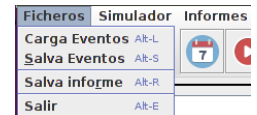
    // BARRA DE ESTADO INFERIOR
    // (contiene una JLabel para mostrar el estado del simulador)
    this.addBarraEstado(panelPrincipal);
    ...
}
```



22

## Ventana principal: constructora

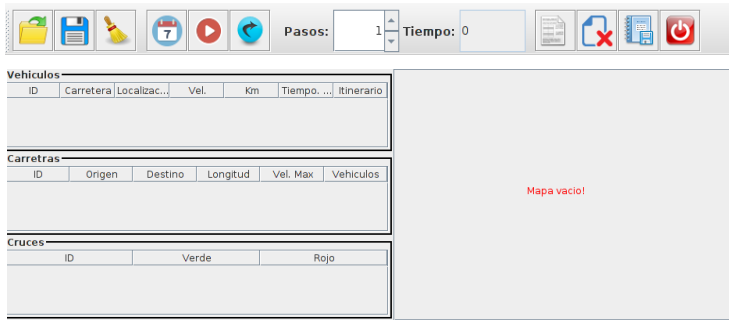
```
private void initGUI() {
    ...
    // PANEL SUPERIOR
    this.createPanelSuperior(panelCentral);
    // MENU
    BarraMenu menubar = new BarraMenu(this, this.controlador);
    this.setJMenuBar(menubar);
    ...
}
```



24

## Ventana principal: constructora

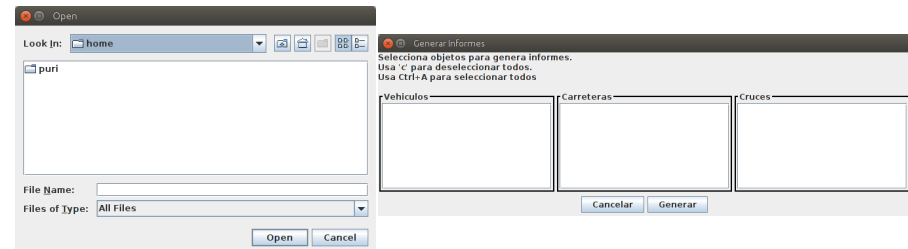
```
private void initGUI() {
    ...
    // PANEL INFERIOR
    this.createPanelInferior(panelCentral);
    // BARRA DE HERRAMIENTAS
    this.addToolBar(panelPrincipal);
    ...
}
```



25

## Ventana principal: constructora

```
private void initGUI() {
    ...
    // FILE CHOOSEER
    this.fc = new JFileChooser();
    // REPORT DIALOG (OPCIONAL)
    this.dialogoInformes = new DialogoInformes(this, this.controlador);
    this.pack();
    this.setVisible(true);
}
```



26

## Panel superior

Creamos un panel con layout “BoxLayout” alineado en el eje de las X. Contendrá:

- **panelEditorEventos:** Contiene un area de texto para mostrar los eventos. Puede crearse con el “texto” de un fichero previamente cargado. **No es observador.**  

```
this.panelEditorEventos = new PanelEditorEventos(titulo,texto,true,this);
```
- **panelColaEventos:** Contendrá una tabla de eventos. **Es observador.**  

```
this.panelColaEventos = new PanelTabla<Evento>("Cola Eventos: ",
    new ModeloTablaEventos(VentanaPrincipal.columnIdEventos,
        this.controlador));
```
- **panelInformes:** Contiene un area de texto para mostrar los informes. **Es observador.**  

```
this.panelInformes = new PanelInformes("Informes: ",false,
    this.controlador);
```

27

## Panel superior

- La booleana en los paneles de edición indica si *el area de texto es editable*.
- Para leer un fichero, en caso de que el comando lo indique:

```
String texto = "";
try {
    texto = this.leeFichero(this.ficheroActual);
} catch (FileNotFoundException e) {
    this.ficheroActual = null;
    this.muestraDialogoError("Error durante la lectura del
    fichero: " + e.getMessage());
}
```

Esto donde lo pongo?

En caso de error debe usarse:  
**JOptionPane.showMessageDialog**  
 para mostrar al usuario información sobre dicho error.

28

## Paneles con área de texto

Tenemos una clase **PanelAreaTexto**, de la que heredan **PanelEditorEventos** y **PanelInformes**.

```
abstract public class PanelAreaTexto extends JPanel {
    protected JTextArea areatexto;

    public PanelAreaTexto(String titulo, boolean editable) {
        this.setLayout(new GridLayout(1,1));
        this.aretexto = new JTextArea(40, 30);
        this.aretexto.setEditable(editable);
        this.add(new JScrollPane(aretexto,...));
        this.setBorde(titulo);
    }
    public void setBorde(String titulo){
        this.setBorder(BorderFactory.createTitledBorder(...));
    }
    public String getTexto() {...}
    public void setTexto(String texto) {...}
    public void limpiar() {...}
    public void inserta(String valor) {
        this.aretexto.insert(valor, this.aretexto.getCaretPosition());
    }
}
```

29

## PanelEditorEventos

```
public class PanelEditorEventos extends PanelAreaTexto {

    public PanelEditorEventos(String titulo, String texto,
        boolean editable, VentanaPrincipal mainWindow) {
        super(titulo,editable);
        this.setTexto(texto);
        // OPCIONAL
        PopUpMenu popUp = new PopUpMenu(mainWindow);
        this.aretexto.add(popUp);
        this.aretexto.addMouseListener(new MouseListener() {
            ...
            @Override
            public void mousePressed(MouseEvent e) {
                if (e.isPopupTrigger() && areatexto.isEnabled())
                    popUp.show(e.getComponent(), e.getX(), e.getY());
            }
            @Override
            public void mouseReleased(MouseEvent e) {...}
        });
    }
}
```

31

## PanelInformes

// OBSERVADOR DEL MODELO

```
public class PanelInformes extends PanelAreaTexto implements
    ObservadorSimuladorTrafico {

    public PanelInformes(String titulo, boolean editable, Controlador ctrl) {
        super(titulo, editable);
        ctrl.addObserver(this); // se añade como observador
    }
    ...
}
```

30

## Tablas

■ Aparecen cuatro tablas de características similares.

■ Para implementarlas utilizaremos un genérico: **PanelTabla<T>**, que se instanciará con **Vehiculo**, **Carretera**, **CruceGenerico<?>** y **Evento**. Esta clase crea un panel y coloca una tabla sobre él, con el *modelo* que se pase en su constructora.

■ Como tenemos cuatro tablas, tendremos cuatro modelos de tabla. Para modularizar la implementación, definiremos una clase genérica **ModeloTabla<T>**, de la que heredarán las clases **ModeloTablaCarreteras**, **ModeloTablaCruces**, **ModeloTablaEventos** y **ModeloTablaVehiculos**.

```
public class PanelTabla<T> extends JPanel {
    private ModeloTabla<T> modelo;

    public PanelTabla(String bordeId, ModeloTabla<T> modelo){
        this.setLayout(new GridLayout(1,1));
        this.setBorder(...);
        this.modelo = modelo;
        JTable tabla = new JTable(this.modelo);
        this.add(new JScrollPane(tabla, ...))
    }
}
```

32



## El modelo para las tablas

```
// EL MODELO SE REGISTRA COMO OBSERVADOR
public abstract class ModeloTabla<T> extends DefaultTableModel
    implements ObservadorSimuladorTrafico {
    protected String[] columnIds;
    protected List<T> lista;

    public ModeloTabla(String[] columnIdEventos, Controlador ctrl) {
        this.lista = null;
        ...
        ctrl.addObserver(this);
    }
    @Override
    public String getColumnName(int col) { return this.columnIds[col]; }
    @Override
    public int getColumnCount() {return this.columnIds.length;}
    @Override
    public int getRowCount() {return this.lista == null ? 0 :
        this.lista.size();}
}
```

33

## Panel inferior

- El panel inferior será un **JPanel** con **BoxLayout** en el eje X.
- Contendrá un *panel para meter las tres tablas*, y un *panel para el gráfico*.
- El panel para las tablas (puedes usar **GridLayout(3,1)**) contendrá a su vez otros tres paneles:

```
this.panelVehiculos = new PanelTabla<Vehiculo>("Vehiculos",
    new ModeloTablaVehiculos(VentanaPrincipal.columnIdVehiculo,
        this.controlador));

this.panelCarreteras = new PanelTabla<Carretera>("Carretras",
    new ModeloTablaCarreteras(VentanaPrincipal.columnIdCarretera,
        this.controlador));

this.panelCruces = new PanelTabla<CruceGenerico<?>>("Cruces",
    new ModeloTablaCruces(VentanaPrincipal.columnIdCruce,
        this.controlador));
```

35

## ModeloTablaEvento

```
public class ModeloTablaEventos extends ModeloTabla<Evento> {

    public ModeloTablaEventos(String[] columnIdEventos, Controlador ctrl) { ... }
    @Override // necesario para que se visualicen los datos
    public Object getValueAt(int indiceFil, int indiceCol) {
        Object s = null;
        switch (indiceCol) {
            case 0: s = indiceFil; break;
            case 1: s = this.lista.get(indiceFil).getTiempo(); break;
            case 2: s = this.lista.get(indiceFil).toString(); break;
            Default: assert (false);
        }
        return s;
    }
    ...
    @Override
    public void avanza(int tiempo, MapaCarreteras mapa, List<Evento> eventos) {
        this.lista = eventos; this.fireTableStructureChanged();
    }
    @Override
    public void addEvento(int tiempo, MapaCarreteras mapa, List<Evento> eventos)...
    @Override
    public void reinicia(int tiempo, MapaCarreteras mapa, List<Evento> eventos) ...
}
```

34

## Panel inferior

- Para los gráficos se definirá una componente gráfica (**ComponenteMapa**) que será también un observador. *La componente se suministra con la práctica.*
- Para crear la componente:

```
this.componenteMapa = new ComponenteMapa(this.controlador);
// añadir un JScrollPane al panel inferior donde se coloca la
// componente.
panelInferior.add(new JScrollPane(componenteMapa,...));
```

36

## Barra de estado

La barra de estado será una clase **PanelBarraEstado**. Creamos este panel con:

```
private void addBarraEstado(JPanel panelPrincipal) {
    this.panelBarraEstado = new PanelBarraEstado("Bienvenido al
        simulador !",this.controlador);
    // se añade al panel principal (el que contiene al panel
    // superior y al inferior)
    panelPrincipal.add(this.panelBarraEstado, BorderLayout.PAGE_END);
}
```

El **PanelBarraEstado** es un observador, y contiene una **JLabel** para ir mostrando el estado de la ejecución.

37

## El menú

La clase **BarraMenu** implementa el menú. El menú necesita el controlador para interactuar con el simulador de tráfico, y también la ventana principal para acciones como “salvar informes”, “cargar eventos”, etc. En la ventana principal tendremos:

```
BarraMenu menubar = new BarraMenu(this,this.controlador);
this.setJMenuBar(menubar);
```

Además necesitamos un **FileChooser** para poder seleccionar archivos donde salvar o cargar información.

```
this.fc = new JFileChooser();
```

39

## Barra de estado

```
public class PanelBarraEstado extends JPanel
    implements ObservadorSimuladorTrafico {
    private JLabel infoEjecucion;

    public PanelBarraEstado(String mensaje, Controlador controlador) {
        this.setLayout(new FlowLayout(FlowLayout.LEFT));
        this.infoEjecucion = new JLabel(mensaje);
        this.add(this.infoEjecucion);
        this.setBorder(BorderFactory.createBevelBorder(1));
        controlador.addObserver(this);
    }
    public void setMensaje(String mensaje) {...} // la ventana principal se
        // comunica con el panel
    ...
    @Override
    public void avanza(int tiempo, MapaCarreteras mapa, List<Evento> eventos) {
        this.infoEjecucion.setText("Paso: " + tiempo + " del Simulador");
    }
    @Override
    public void addEvento(int tiempo, MapaCarreteras mapa, List<Evento> eventos) {
        this.infoEjecucion.setText("Evento añadido al simulador");
    }
    @Override
    public void reinicia(int tiempo, MapaCarreteras mapa, List<Evento> eventos)...
}
```

38

## La clase BarraMenú

```
public class BarraMenu extends JMenuBar {
```

```
    public BarraMenu(VentanaPrincipal mainWindow, Controlador controlador) {
        super();
        // MANEJO DE FICHEROS
        JMenu menuFicheros = new JMenu("Ficheros");
        this.add(menuFicheros);
        this.creaMenuFicheros(menuFicheros,mainWindow);
        // SIMULADOR
        JMenu menuSimulador = new JMenu("Simulador");
        this.add(menuSimulador);
        this.creaMenuSimulador(menuSimulador,controlador,mainWindow);
        // INFORMES
        JMenu menuReport = new JMenu("Informes");
        this.add(menuReport);
        this.creaMenuInformes(menuReport,mainWindow);
    }
}
```

40

## Menú para los ficheros

```
private void creaMenuFicheros(JMenu menu, VentanaPrincipal mainWindow) {
    JMenuItem cargar = new JMenuItem("Carga Eventos");
    cargar.setMnemonic(KeyEvent.VK_L);
    cargar.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,
        ActionEvent.ALT_MASK));
    cargar.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            mainWindow.cargaFichero();
        }
    });
    ...
    menu.add(cargar);
    menu.add(salvar);
    menu.addSeparator();
    menu.add(salvarInformes);
    menu.addSeparator();
    menu.add(salir);
}
```

41

## Cargar un fichero

El fichero se carga desde la ventana principal, usando FileChooser.

```
public void cargaFichero() {
    int returnVal = this.fc.showOpenDialog(null);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
        File fichero = this.fc.getSelectedFile();
        try {
            String s = leeFichero(fichero);
            this.controlador.reinicia();
            this.ficheroActual = fichero;
            this.panelEditorEventos.setText(s);
            this.panelEditorEventos.setBorde(this.ficheroActual.getName());
            this.panelBarraEstado.setMensaje("Fichero " + fichero.getName() +
                " de eventos cargado into the editor");
        }
        catch (FileNotFoundException e) {
            this.muestraDialogoError("Error durante la lectura del fichero: " +
                e.getMessage());
        }
    }
}
```

42

## Menú para el simulador

```
private void creaMenuSimulador(JMenu menuSimulador, Controlador controlador,
    VentanaPrincipal mainWindow) {
    JMenuItem ejecuta = new JMenuItem("Ejecuta");
    ejecuta.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            int pasos = mainWindow.getSteps();
            controlador.ejecuta(pasos);
        }
    });
    JMenuItem reinicia = new JMenuItem("Reinicia");
    ...
    menuSimulador.add(ejecuta);
    menuSimulador.add(reinicia);
}
```

43

## Menú para los informes

```
private void creaMenuInformes(JMenu menuReport,
    VentanaPrincipal mainWindow) {
    JMenuItem generaInformes = new JMenuItem("Generar");
    generaInformes.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            // OPCIONAL
            mainWindow.generaInformes();
        }
    });
    menuReport.add(generaInformes);
    JMenuItem limpiaAreaInformes = new JMenuItem("Clear");
    ...
    menuReport.add(limpiaAreaInformes);
}
```

44

## Barra de herramientas

■ Tiene una funcionalidad similar a la del Menú. En la ventana principal, método `initGUI()`, aparecerá la llamada `this.addBarraEstado(panelPrincipal)`. Este método crea la barra de herramientas y la añade al panel principal, al inicio de la página.

■ Como tiene opciones que dependen de la ventana principal y del simulador, esta componente será un observador.

```
this.toolbar = new ToolBar(this, this.controlador);
panelPrincipal.add(this.toolbar, BorderLayout.PAGE_START);
```

45

## Barra de Herramientas

```
public class ToolBar extends JToolBar
    implements ObservadorSimuladorTrafico {

    private JSpinner steps;
    private JTextField time;

    public ToolBar(VentanaPrincipal mainWindow, Controlador controlador){
        super();
        controlador.addObserver(this);
        ...
    }
}
```

46

## Barra de Herramientas: Cargar eventos

```
JButton botonCargar = new JButton();
botonCargar.setToolTipText("Carga un fichero de ventos");
botonCargar.setIcon(new
    ImageIcon(Utils.loadImage("resources/icons/open.png")));
botonCargar.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        mainWindow.cargaFichero();
    }
});
this.add(botonCargar);
```

47

## Barra de Herramientas (check-in)

```
JButton botonCheckIn = new JButton();
botonCheckIn.setToolTipText("Carga los eventos al simulador");
botonCheckIn.setIcon(new
    ImageIcon(Utils.loadImage("resources/icons/events.png")));
botonCheckIn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            controlador.reinicia();
            byte[] contenido = mainWindow.getTextoEditorEventos().getBytes();
            controlador.cargaEventos(new ByteArrayInputStream(contenido));
        } catch (ErrorDeSimulacion err) { }
        mainWindow.setMensaje("Eventos cargados al simulador!");
    }
});
this.add(botonCheckIn);
```

48

## Barra de Herramientas: Spinner

```
this.add(new JLabel(" Pasos: "));
this.steps = new JSpinner(new SpinnerNumberModel(5, 1, 1000, 1));
this.steps.setToolTipText("pasos a ejecutar: 1-1000");
this.steps.setMaximumSize(new Dimension(70, 70));
this.steps.setMinimumSize(new Dimension(70, 70));
this.steps.setValue(1);
this.add(steps);
```

49

## Barra de Herramientas: Informes (Opcional)

```
// OPCIONAL
JButton botonGeneraReports = new JButton();
botonGeneraReports.setToolTipText("Generata informes");
botonGeneraReports.setIcon(new
    ImageIcon(Utils.loadImage("resources/icons/report.png")));
botonGeneraReports.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        mainWindow.generaInformes();
    }
});
this.add(botonGeneraReports);
```

51

## Barra de herramientas: Tiempo

```
this.add(new JLabel(" Tiempo: "));
this.time = new JTextField("0", 5);
this.time.setToolTipText("Tiempo actual");
this.time.setMaximumSize(new Dimension(70, 70));
this.time.setMinimumSize(new Dimension(70, 70));
this.time.setEditable(false);
this.add(this.time);
```

50

## Reaccionando como observador

```
@Override
public void errorSimulador(int tiempo, MapaCarreteras mapa,
    List<Evento> eventos, ErrorDeSimulacion e) {}

@Override
public void avanza(int tiempo, MapaCarreteras mapa,
    List<Evento> eventos) {
    this.time.setText(""+tiempo);
}

@Override
public void addEvento(int tiempo, MapaCarreteras mapa,
    List<Evento> eventos) {}

@Override
public void reinicia(int tiempo, MapaCarreteras mapa,
    List<Evento> eventos) {
    this.steps.setValue(1);
    this.time.setText("0");
}
```

52

## Pop-Up menú

Aparece en el **PanelEditorEventos**, donde opcionalmente podemos crearlo en su constructora.

```
PopupMenu popUp = new PopupMenu(mainWindow);
this.aretexto.add(popUp);
this.aretexto.addMouseListener(new MouseListener() {
    ...
    @Override
    public void mousePressed(MouseEvent e) {
        if (e.isPopupTrigger() && areatexto.isEnabled()) {
            popUp.show(e.getComponent(), e.getX(), e.getY());
        }
    }
    ...
});
```

Esto lo he puesto  
en panel Editor  
Eventos

53

## Pop-Up menú

■ Las opciones de **Cargar**, **Salvar** y **Limpiar** son similares a las del menú.

■ La opción más interesante es la de “**Nueva Plantilla**”.

■ En la clase **ParserEventos** tenemos todos los constructores de eventos almacenados en un array. Por cada uno de esos constructores de eventos, tenemos que crear una opción dentro de “**Nueva Plantilla**”.

■ Lo más fácil es recorrer dicho array invocando al método “**toString**” de cada constructor de eventos.

```
public PopupMenu(VentanaPrincipal mainWindow) {
    ...
    JMenu plantillas = new JMenu("Nueva plantilla");
    this.add(plantillas);
    // añadir las opciones con sus listeners
```

54

## Pop-Up menú

```
for (ConstructorEventos ce : ParserEventos.getConstructoresEventos()) {
    JMenuItem mi = new JMenuItem(ce.toString());
    mi.addActionListener(new ActionListener() {
        ...
        @Override
        public void actionPerformed(ActionEvent e) {
            mainWindow.inserta(ce.template() + System.lineSeparator());
        }
    });
    plantillas.add(mi);
}
```

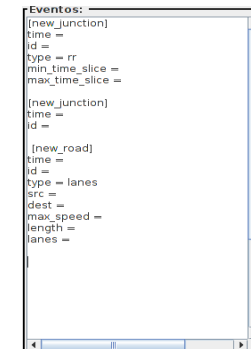
// **String template()** es un método público que debe definirse en la  
// clase **ConstructorEventos**, y que debe generar la plantilla  
// correspondiente en función de los campos, y teniendo en cuenta  
// los posibles valores por defecto.

??  
Esto donde lo  
pongo??

55

## ConstructorEventos

Esta clase, además de los atributos de la Práctica 4 (**etiqueta** y **claves**), tendrá un nuevo atributo: **String[] valoresPorDefecto**. Este atributo almacena los valores por defecto de un constructor de eventos. Para el caso de las plantillas, vamos a suponer que el único valor por defecto que vamos a tener es el tipo del objeto en cuestión. El resto de campos deberá rellenarlos el usuario en el área correspondiente al editor de eventos.



56

## ConstructorEventoNuevoCruceCircular

- Para esta clase, el atributo valores por defecto se inicializa a:

```
this.valoresPorDefecto = new String[] { "", "", "rr", "", "" };
```

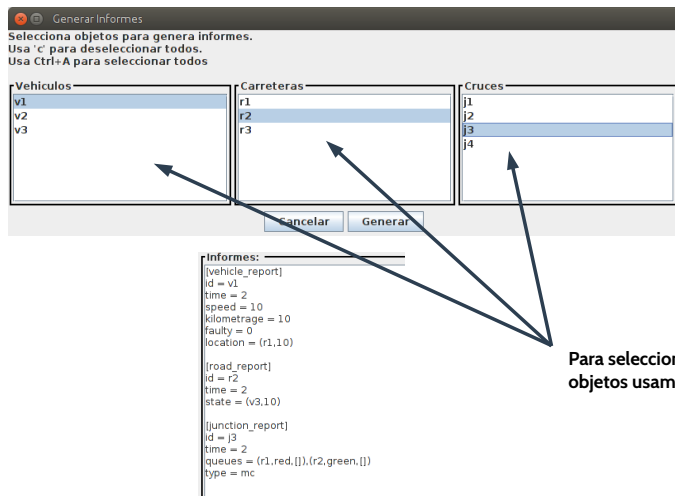
- y el método "toString" será:

```
@Override
public String toString() {
    return "Nuevo Cruce Circular";
}
```

- El resto de clases deben hacer el mismo proceso. Los objetos básicos (parte I) de la práctica, no tienen valores por defecto.

57

## Generar informes



59

## Generar informes

- Tanto el menú como la barra de herramientas tienen una opción para generar informes.

- Dicha opción abre un **diálogo** con los objetos de la simulación actual.

- Se seleccionan con el ratón los objetos sobre los cuales generar el informe, y al pulsar el botón generar, el área correspondiente a los informes se limpia, y aparecen los informes asociados a los objetos seleccionados.

- Para implementar esta opción necesitamos un nuevo Frame, del tipo **JDialog**. Este Frame estará activo en la ventana principal, y sólo se mostrará cuando se seleccione la opción de generar informes.

58

## Generar informes

- En la ventana principal, método `initGUI()`, tendremos:

```
this.dialogoInformes = new DialogoInformes(this, this.controlador);
```

- Esta componente será una observadora, ya que puede estar visible u oculta, pero debe tener *siempre actualizados los objetos de la simulación* presentes en cada paso.
- El **JDialog** contiene tres paneles, y cada uno de estos paneles contendrá un **JList** para mostrar la información sobre los correspondientes objetos de la simulación y poder seleccionarlos.
- Por tanto vamos a tener una clase genérica para estos paneles **PanelObjSim<T>**, que será la que contenga el correspondiente objeto **JList<T>** objList.

60

## El modelo

```
public class ListModel<T> extends DefaultListModel<T> {  
    private List<T> lista;  
  
    ListModel() { this.lista = null; }  
  
    public void setList(List<T> lista) {  
        this.lista = lista;  
        fireContentsChanged(this, 0, this.lista.size());  
    }  
  
    @Override  
    public T getElementAt(int index) { ... }  
  
    @Override  
    public int getSize() {  
        return this.lista == null ? 0 : this.lista.size();  
    }  
}
```

61

## Panel de Objetos de Simulación

```
public class PanelObjSim<T> extends JPanel {  
  
    private ListModel<T> listModel;  
    private JList<T> objList;  
  
    public PanelObjSim(String titulo) {  
        ...  
        this.listModel = new ListModel<T>();  
        this.objList = new JList<T>(this.listModel);  
        addCleanSelectionListener(objList);  
        this.add(new JScrollPane(objList,  
            JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,  
            JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED), BorderLayout.CENTER);  
    }  
}
```

62

## Panel de Objetos de Simulación

```
private void addCleanSelectionListener(JList<?> list) {  
    list.addKeyListener(new KeyListener() {  
        // limpiar la seleccion de items pulsando "c"  
        @Override  
        public void keyTyped(KeyEvent e) {  
            if (e.getKeyChar() == DialogoInformes.TECLALIMPIAR)  
                list.clearSelection();  
        }  
    });  
}  
  
public List<T> getSelectedItems() {  
    List<T> l = new ArrayList<>();  
    for (int i : this.objList.getSelectedIndices()) {  
        l.add(listModel.getElementAt(i));  
    }  
    return l;  
}  
  
public void setList(List<T> lista) { this.listModel.setList(lista); }
```

63

## Diálogo de Informes

```
public class DialogoInformes extends JDialog  
    implements ObservadorSimuladorTrafico {  
  
    private PanelBotones panelBotones;  
    private PanelObjSim<Vehiculo> panelVehiculos;  
    private PanelObjSim<Carretera> panelCarreteras;  
    private PanelObjSim<CruceGenerico<?>> panelCruces;  
    ...  
  
    private void initGUI() {  
        ...  
        this.panelVehiculos = new PanelObjSim<Vehiculo>("Vehiculos");  
        this.panelCarreteras = new PanelObjSim<Carretera>("Carreteras");  
        this.panelCruces = new PanelObjSim<CruceGenerico<?>>("Cruces");  
        this.panelBotones = new PanelBotones(this);  
        InformationPanel panelInfo = new InformationPanel();  
        panelPrincipal.add(panelInfo, BorderLayout.PAGE_START);  
        ...  
    }  
}
```

64



## Diálogo de Informes

```
public void mostrar() { this.setVisible(true); }

private void setMapa(MapaCarreteras mapa) {
    this.panelVehiculos.setList(mapa.getVehiculos());
    this.panelCarreteras.setList(mapa.getCarreteras());
    this.panelCruces.setList(mapa.getCruces());
}

public List<Vehiculo> getVehiculosSeleccionados() {
    return this.panelVehiculos.getSelectedItems();
}

public List<Carretera> getCarreterasSeleccionadas() {
    return this.panelCarreteras.getSelectedItems();
}

public List<CruceGenerico<?>> getCrucesSeleccionados() {
    return this.panelCruces.getSelectedItems();
}
```

65

## Diálogo de informes

```
...

@Override
public void avanza(int tiempo, MapaCarreteras mapa, List<Evento> eventos) {
    this.setMapa(mapa);
}

@Override
public void addEvento(int tiempo, MapaCarreteras mapa, List<Evento> eventos) {
    this.setMapa(mapa);
}

@Override
public void reinicia(int tiempo, MapaCarreteras mapa, List<Evento> eventos) {
    this.setMapa(mapa);
}
```

66