
Práctica 5: Interfaz gráfica para el simulador físico

Fecha de entrega: 23 de abril de 2018 a las 09:00

Objetivo: Diseño orientado a objetos, Modelo-Vista-Controlador, interfaces gráficas de usuario con Swing.

1. Control de copias

Durante el curso se realizará control de copias de todas las prácticas, comparando las entregas de todos los grupos de TP. Se considera copia la reproducción total o parcial del código de otros alumnos o cualquier código extraído de Internet o de cualquier otra fuente, salvo aquellas autorizadas explícitamente por el profesor.

En caso de detección de copia se informará al Comité de Actuación ante Copias que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres medidas siguientes:

- Calificación de cero en la convocatoria de TP a la que corresponde la práctica o examen.
- Calificación de cero en todas las convocatorias de TP del curso actual 2018/2019.
- Apertura de expediente académico ante la *Inspección de Servicios de la Universidad*.

2. Instrucciones Generales

Las siguientes instrucciones **son estrictas**, es decir, debes seguirlas obligatoriamente.

1. Lee el enunciado completo de la práctica antes de empezar a escribir código.
2. Haz una copia de la práctica 4 antes de hacer cambios en ella para la práctica 5.

3. Crea un nuevo paquete `simulator.view` para colocar en él todas las clases necesarias para las vistas.
4. Es necesario usar exactamente la misma estructura de paquetes y los mismos nombres de clases que aparecen en el enunciado.
5. No está permitido el uso de ninguna herramienta para la generación automática de interfaces gráficas de usuario.
6. Cuando entregues la práctica sube un fichero **zip** del proyecto que incluya todos los subdirectorios excepto el subdirectorio **bin**. **Otros formatos (por ejemplo 7zip, rar, etc.) no están permitidos.**

3. Descripción general de la práctica

En esta práctica vas a desarrollar una interfaz gráfica de usuario (GUI) para el simulador físico siguiendo el patrón de diseño modelo-vista-controlador. En la figura 1 puedes ver la GUI que hay que construir. Está compuesta por una ventana principal que contiene cuatro componentes: (1) un panel de control para interactuar con el simulador; (2) una tabla que muestra el estado de todos los cuerpos; (3) un visor (*viewer*) en el que aparecen dibujados los cuerpos en cada paso de la simulación; y (4) una barra de estado en la que aparece más información, que detallaremos después.

Las secciones 4 y 5 describen los cambios que hay que hacer en el modelo y en el controlador, respectivamente; la sección 6 describe la funcionalidad y los detalles para la implementación de la GUI; y la sección 7 describe los cambios que hay que hacer en la clase `Main`.

4. Cambios en el modelo

Esta sección describe los cambios que hay que hacer en el modelo para usar el patrón de diseño MVC y para añadir alguna funcionalidad extra.

4.1. Nueva funcionalidad

Añade los siguientes métodos a la clase `PhysicsSimulator` (si es que no los tienes ya):

- `public void reset()`: vacía la lista de cuerpos y pone el tiempo a 0,0.
- `public setDeltaTime(double dt)`: cambia el *tiempo real por paso* (delta-time de aquí en adelante) a `dt`. Si `dt` tiene un valor no válido lanza una excepción de tipo `IllegalArgumentException`.
- `public void setGravityLaws(GravityLaws gravityLaws)`: cambia las leyes de gravedad del simulador a `gravityLaws`. Lanza una `IllegalArgumentException` si el valor no es válido, es decir, si es `null`.

Además, añade un método `toString()` a las clases `NewtonUniversalGravitation`, `FallingToCenterGravity` y `NoGravity` que devuelva una breve descripción de las leyes de gravedad correspondientes.

4.2. Uso de MVC en Model

Esta sección explica cómo modificar el modelo para usar el patrón de diseño MVC, es decir, para permitir que los observadores reciban del modelo notificaciones de determinados eventos.

La interfaz SimulatorObserver

Representamos a los observadores a través de la siguiente interfaz, que incluye varios tipos de notificaciones (y que hay que colocar en el paquete `simulator.model`):

```
public interface SimulatorObserver {
    public void onRegister(List<Body> bodies, double time, double dt, String gLawsDesc);
    public void onReset(List<Body> bodies, double time, double dt, String gLawsDesc);
    public void onBodyAdded(List<Body> bodies, Body b);
    public void onAdvance(List<Body> bodies, double time);
    public void onDeltaTimeChanged(double dt);
    public void onGravityLawChanged(String gLawsDesc);
}
```

Los nombres de los métodos dan información sobre el significado de los eventos que notifican. En cuando a los parámetros: `bodies` es la lista de cuerpos actual; `b` es un cuerpo, `time` es el tiempo actual del simulador (el delta-time); `dt` es el tiempo por paso actual del simulador; `gLawsDesc` es un string que describe las leyes de gravedad actuales (que se obtiene invocando al método `toString()` de la ley de gravedad actual).

Nuevos observadores

Añade a `PhysicsSimulator` un campo que sea una lista de observadores, inicialmente vacía, y añade el siguiente método para registrar un nuevo observador:

- `public void addObserver(SimulatorObserver o)`: añade `o` a la lista de observadores, si es que no está ya en ella.

Envío de notificaciones

Cambia la clase `PhysicsSimulator` para enviar notificaciones como se describe a continuación:

- Al final del método `addObserver` envía una notificación `onRegister` **solo al observador que se acaba de registrar** para pasarle el estado actual del simulador.
- Al final del método `reset` envía una notificación `onReset` a **todos los observadores**.
- Al final del método `addBody` envía una notificación `onBodyAdded` a **todos los observadores**.
- Al final del método `advance` envía una notificación `onAdvance` a **todos los observadores**.
- Al final del método `setDeltaTime` envía una notificación `onDeltaTimeChanged` a **todos los observadores**.
- Al final del método `setGravityLaws` envía una notificación `onGravityLawsChanged` a **todos los observadores**.

5. Preparación de Controller

También es necesario añadir nueva funcionalidad al controlador. En primer lugar, cambia la constructora de modo que reciba un nuevo parámetro de tipo `Factory<GravityLaws>` que se almacene en el campo correspondiente. Además, añade los siguientes métodos:

- `public void reset()`: invoca al método `reset` del simulador.
- `public void setDeltaTime(double dt)`: invoca al método `setDeltaTime` del simulador.
- `public void addObserver(SimulatorObserver o)`: invoca al método `addObserver` del simulador.
- `public void run(int n)`: ejecuta `n` pasos del simulador sin escribir nada en consola.
- `public Factory<GravityLaws> getGravityLawsFactory()` devuelve la factoría de leyes de gravedad.
- `public void setGravityLaws(JSONObject info)`: usa la factoría de leyes de gravedad actual para crear un nuevo objeto de tipo `GravityLaws` a partir de `info` y cambia las leyes de la gravedad del simulador por él.

6. La interfaz gráfica de usuario

La figura 1 muestra el aspecto general de la GUI. Está compuesta por una ventana principal, compuesta a su vez por cuatro componentes: (1) un panel de control para que el usuario interactúe con el simulador; (2) una tabla que muestra el estado de todos los cuerpos; (3) un visor (*viewer*) para mostrar gráficamente el estado del universo en cada paso de la simulación; (4) una barra de estado en la que se muestra más información, que detallaremos después.

Representaremos la ventana principal mediante una clase que extiende a `JFrame`, y el resto de componentes mediante clases que extienden a `JPanel` (o `JComponent`). Esto permite manejar todas las componentes como componentes Swing que se alojan en la ventana principal, lo que a su vez permite reemplazar una implementación por otra sin necesidad de hacer modificaciones profundas en el código.

6.1. Panel de control

El panel de control es responsable de la interacción usuario-simulador. Lo representaremos mediante la clase `ControlPanel`:

```
public class ControlPanel extends JPanel implements SimulatorObserver {
    // ...
    private Controller _ctrl;
    private boolean _stopped;

    ControlPanel(Controller ctrl) {
        _ctrl = ctrl;
        _stopped = true;
        initGUI();
        _ctrl.addObserver(this);
    }

    private void initGUI() {
```

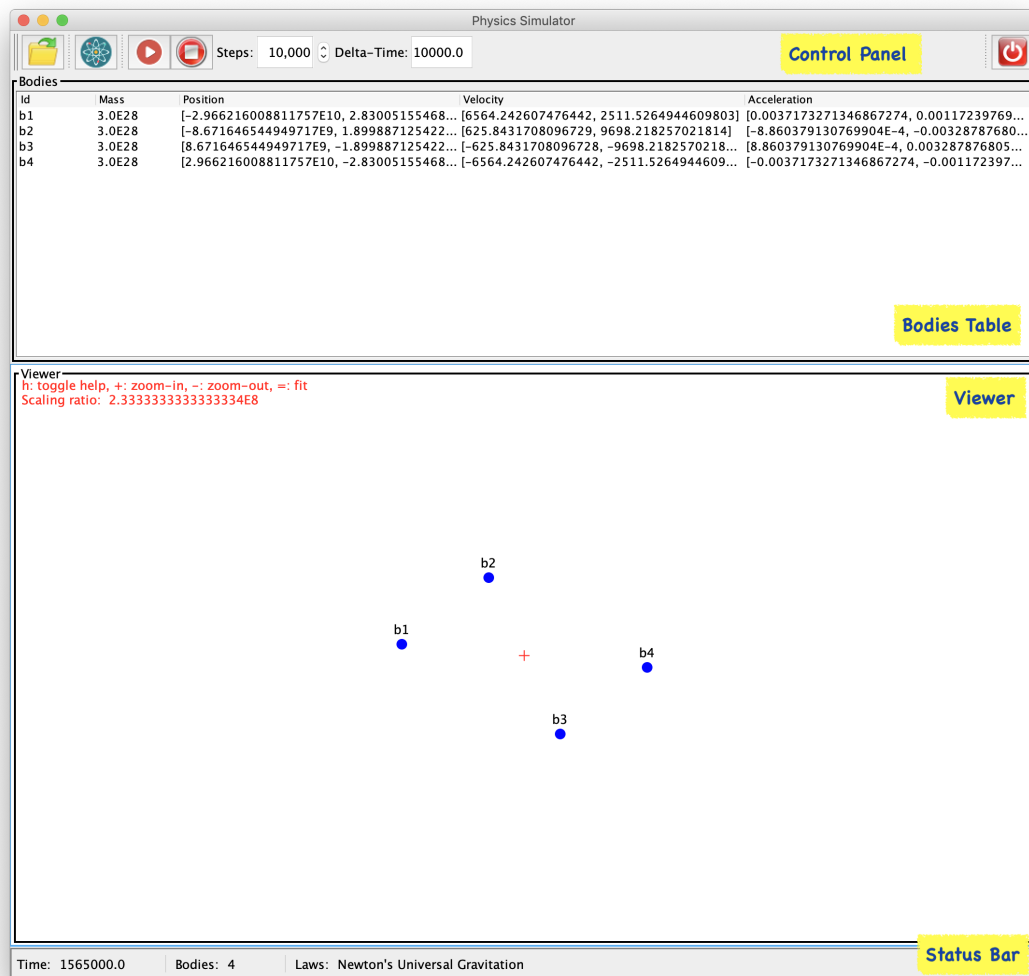


Figura 1: Interfaz gráfica de usuario. Los rectángulos amarillos son anotaciones, es decir, no son parte de la GUI.

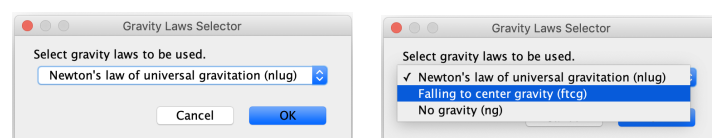


Figura 2: Cuadro de diálogo para cambiar las leyes de la gravedad.

```

    // TODO build the tool bar by adding buttons, etc.
}







// other private/protected methods
// ...

private void run(int n) {
    if ( n>0 && !_stopped) {
        _ctrl.run(1);
        SwingUtilities.invokeLater( new Runnable() {
            @Override
            public void run() {
                run(n-1);
            }
        });
    } else {
        // TODO complete this code to enable all buttons
    }
}

// SimulatorObserver methods
// ...
}

```

Es necesario completar el método `initGUI()` para crear todas las componentes del panel (botones, selector de número de pasos, etc.). Puedes encontrar los iconos en el directorio `resources/icons`. Como selector de *pasos* usa un `JSpinner` y para el área *Delta-Time* usa un `JTextField`. Todos los botones han de tener *tooltips* para describir el efecto de pulsarlos. Los botones han de tener la siguiente funcionalidad:

- Al pulsar : (1) pídele al usuario un fichero mediante un `JFileChooser`; (2) limpia el simulador usando `_ctrl.reset()`; y (3) carga el fichero seleccionado en el simulador usando `_ctrl.loadBodies(...)`.
- Al pulsar : (1) abre una caja de diálogo en la que se le pida al usuario una de las leyes de gravedad disponibles – véase la figura 2; y (2) cambia las leyes de gravedad del simulador por las seleccionadas (usando `_ctrl.setGravityLaws(...)`). Puedes usar `_ctrl.getGravityLawsFactory().getInfo()` para obtener las leyes de gravedad disponibles.
- Al pulsar : (1) desactiva todos los botones, excepto ; (2) pon el delta-time actual del simulador al especificado por el correspondiente campo de texto; y (3) llama al método `run` con el valor actual del número de pasos según el `JSpinner`. Es necesario completar el método `run` para que active de nuevo todos los botones al concluir la ejecución de la simulación. Observa que el método `run` anterior garantiza que la interfaz no se bloquea (prueba a modificar el cuerpo de `run` por la instrucción `_ctrl.run(n)`; verás que no se muestran los pasos intermedios, solo el resultado final, y que entre tanto la interfaz permanece bloqueada).
- Al pulsar  cambia el valor del campo `_stopped` a `true`, lo que detendrá la ejecución del método `run` si hay llamadas pendientes en la cola (véase la condición en el bucle del método `run`).
- Al pulsar  pídele al usuario confirmación y después cierra la aplicación usando `System.exit(0)`.

Además, captura todas las excepciones lanzadas por el controlador/simulador y muestra el correspondiente mensaje de error usando un cuadro de diálogo (por ejemplo, usando `JOptionPane.showMessageDialog()`). En los métodos de `SimulatorObserver` modifica el valor del *delta-time* en la correspondiente `JTextField` siempre que sea necesario (es decir, en `onRegister`, `onReset` y `onDeltaTimeChanged`).

6.2. Tabla de cuerpos

Esta componente muestra el estado de todos los cuerpos usando una `JTable` (un cuerpo en cada fila). Para implementar esta tabla vamos a usar dos clases: (1) una clase para el modelo de tabla, que es también un observador, de forma que cuando cambie el estado del simulador éste notifique al modelo y se actualice la tabla; y (2) una clase que cree una `JTable` y le asigne el modelo de tabla anterior.

La clase para el modelo de tabla será la clase llamada `BodiesTableModel`:

```
public class BodiesTableModel extends AbstractTableModel implements SimulatorObserver {

    // ...
    private List<Body> _bodies;

    BodiesTableModel(Controllor ctrl) {
        _bodies = new ArrayList<>();
        ctrl.addObserver(this);
    }

    @Override
    public int getRowCount() {
        // TODO complete
    }

    @Override
    public int getColumnCount() {
        // TODO complete
    }

    @Override
    public String getColumnName(int column) {
        // TODO complete
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        // TODO complete
    }

    // SimulatorObserver methods
    // ...
}
```

En los métodos como observador, cuando cambia el estado (por ejemplo en `onAdvance()`, `onRegister`, `onBodyAdded` y `onReset`) es necesario en primer lugar actualizar el valor del campo `_bodies` y después llamar a `fireTableStructureChanged()` para notificar a la correspondiente `JTable` que hay cambios en la tabla (y por lo tanto es necesario redibujarla).

La tabla en sí viene dada por la clase `BodiesTable`:

```
public class BodiesTable extends JPanel {

    BodiesTable(Controllor ctrl) {
        setLayout(new BorderLayout());
        setBorder(BorderFactory.createTitledBorder(
            BorderFactory.createLineBorder(Color.black, 2),
            "Bodies",
```

```

        TitledBorder.LEFT, TitledBorder.TOP));

        // TODO complete
    }
}

```

Es necesario que completes el código anterior (1) creando una instancia de `BodiesTableModel` que se le pase a la `JTable`; y (2) añadiendo la `JTable` al panel (es decir, a `this`) con un `JScrollPane`.

6.3. Viewer

Esta componente muestra de forma gráfica el estado de todos los cuerpos en cada paso de la simulación. La implementamos en la clase `Viewer`, que hereda de `JComponent` y que sobrescribe el método `paintComponent` (también podríamos heredar de `JPanel`). Swing invoca a este método cada vez que es necesario volver a pintar la componente. Esta clase también es un observador, de modo que cuando el estado del simulador cambia le pediremos a Swing que vuelva a pintar la componente llamando al método `repaint()` (que a su vez llama automáticamente a `paintComponent`).

```

public class Viewer extends JComponent implements SimulatorObserver {

    // ...
    private int _centerX;
    private int _centerY;
    private double _scale;
    private List<Body> _bodies;
    private boolean _showHelp;

    Viewer(Controller ctrl) {
        initGUI();
        ctrl.addObserver(this);
    }

    private void initGUI() {
        // TODO add border with title

        _bodies = new ArrayList<>();
        _scale = 1.0;
        _showHelp = true;

        addKeyListener(new KeyListener() {
            // ...
            @Override
            public void keyPressed(KeyEvent e) {
                switch (e.getKeyChar()) {
                    case '-':
                        _scale = _scale * 1.1;
                        break;
                    case '+':
                        _scale = Math.max(1000.0, _scale / 1.1);
                        break;
                    case '=':
                        autoScale();
                        break;
                    case 'h':
                        _showHelp = !_showHelp;
                        break;
                    default:
                }
                repaint();
            }
        });

        addMouseListener(new MouseListener() {

```



```

        // ...
        @Override
        public void mouseEntered(MouseEvent e) {
            requestFocus();
        }
    });
}

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D gr = (Graphics2D) g;
    gr.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
    gr.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
        RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

    // use 'gr' to draw not 'g'

    // calculate the center
    _centerX = getWidth() / 2;
    _centerY = getHeight() / 2;

    // TODO draw a cross at center
    // TODO draw bodies
    // TODO draw help if _showHelp is true
}

// other private/protected methods
// ...

private void autoScale() {
    double max = 1.0;

    for (Body b : _bodies) {
        Vector p = b.getPosition();
        for (int i = 0; i < p.dim(); i++)
            max = Math.max(max, Math.abs(b.getPosition().coordinate(i)));
    }

    double size = Math.max(1.0, Math.min((double) getWidth(), (double) getHeight()));
    _scale = max > size ? 4.0 * max / size : 1.0
}

// SimulatorObserver methods
// ...
}

```

A continuación explicamos las distintas partes del código anterior:

- Los campos `_centerX` y `_centerY` representan la posición del centro de la componente, es decir, la anchura y la altura divididas por 2, respectivamente (véase el método `paintComponent`).
- El campo `_bodies` representa la lista actual de cuerpos. Es necesario actualizar esta lista cada vez que cambia el estado del simulador.
- El campo `_scale` se usa para escalar el universo, es decir, para dibujar todos los cuerpos dentro del área de la componente (ya que el universo suele usar coordenadas bastante mayores). El usuario puede modificar su valor pulsando `+`, lo que lo incrementa, pulsando `-`, lo que lo decrementa, o escribiendo `=`, en cuyo caso su valor se calcula automáticamente en el método `autoScale`.
- El campo `_showHelp` indica si se muestra el texto de ayuda (en la esquina superior izquierda). Su valor cambia al pulsar `h`.

- La llamada `addKeyListener` registra al listener que captura eventos del teclado cuando la componente tiene el foco. Análogamente, la llamada a `addMouseListener` registra al listener que captura eventos del ratón (para solicitar el foco cuando el ratón entra en esta componente).

Tienes que completar el método `paintComponent` dibujando (1) una cruz en el centro; (2) el mensaje de ayuda si `_showHelp` es `true`; y (3) los cuerpos. Para dibujar un cuerpo pinta un círculo de radio 5 con centro en

```
( _centerX + (int) (x/_scale), _centerY - (int) (y/_scale) )
```

donde `x` e `y` son las coordenadas 0 y 1 del cuerpo (si tiene más de dos dimensiones usa solo las dos primeras). Además, escribe el nombre del cuerpo junto al círculo. Para dibujar usa la variable `gr` y sus métodos, como `gr.setColor`, `gr.fillOval`, `gr.drawString`, o `gr.drawLine`.

En los métodos de `SimulatorObserver`, cuando cambie el estado (es decir, en los métodos `onRegister`, `onBodyAdded` y `onReset`) actualiza el valor de `_bodies` e invoca a `autoScale()` y a `repaint()`. En el método `onAdvance()` llama sólo a `repaint()`.

6.4. Barra de estado

La barra de estado muestra información adicional sobre el estado del simulador: el tiempo actual, el número total de cuerpos y las leyes de gravedad. La representamos mediante la clase `StatusBar`:

```
public class StatusBar extends JPanel implements SimulatorObserver {

    // ...
    private JLabel _currTime;      // for current time
    private JLabel _currLaws;      // for gravity laws
    private JLabel _numOfBodies;   // for number of bodies

    StatusBar(Controller ctrl) {
        initGUI();
        ctrl.addObserver(this);
    }

    private void initGUI() {
        this.setLayout( new FlowLayout( FlowLayout.LEFT ) );
        this.setBorder( BorderFactory.createBevelBorder( 1 ) );

        // TODO complete the code to build the tool bar
    }

    // other private/protected methods
    // ...

    // SimulatorObserver methods
    // ...
}
```

Observa que los campos `_currTime`, `_numberOfBodies` y `_currLaws` son etiquetas en las que se almacena la correspondiente información. En los métodos como observadora es necesario modificar la correspondiente `JLabel` si la información cambia.

6.5. Ventana principal

La ventana principal viene dada por una clase llamada `MainWindow` que extiende a `JFrame`:

```

public class MainWindow extends JFrame {

    // ...
    Controller _ctrl;
    private ControlPanel _ctrlPanel;
    private BodiesTable _bodiesInfo;
    private Viewer _universeViewer;
    private JPanel _contentPanel;
    private StatusBar _statusBar;

    public MainWindow(Controller ctrl) {
        super("Physics Simulator");
        _ctrl = ctrl;
        initGUI();
    }

    private void initGUI() {
        JPanel mainPanel = new JPanel(new BorderLayout());
        setContentPane(mainPanel);

        // TODO complete this method to build the GUI
    }

    // other private/protected methods
    // ...
}

```

Es necesario completar el método `initGUI()` para crear los correspondientes objetos y construir la GUI: (1) coloca el panel de control en el `PAGE_START` del panel `mainPanel`; (2) coloca la barra de estado en el `PAGE_END` del `mainPanel`; (3) crea un nuevo panel que use `BoxLayout` (y `BoxLayout.Y_AXIS`) y colócalo en el `CENTER` de `mainPanel`. Añade la tabla de cuerpos y el viewer en este panel.

Para controlar el tamaño inicial de cada componente puedes usar el método `setPreferredSize`. También necesitarás hacer visible la ventana, etc.

7. Cambios en la clase Main

En la clase `Main` es necesario añadir una nueva opción `-m` que permita al usuario usar el simulador en modo `BATCH` (como en la Práctica 4) y en modo `GUI`:

```

usage: simulator.launcher.Main [-dt <arg>] [-gl <arg>] [-h] [-i <arg>] [-m
    <arg>] [-o <arg>] [-s <arg>]
    -dt,--delta-time <arg>    A double representing actual time, in seconds,
                                per simulation step. Default value: 2500.0.
    -gl,--gravity-laws <arg>  Gravity laws to be used in the simulator.
                                Possible values: 'nlug' (Newton's law of
                                universal gravitation), 'ftcg' (Falling to
                                center gravity), 'ng' (No gravity). Default
                                value: 'nlug'.
    -h,--help                  Print this message.
    -i,--input <arg>          Bodies JSON input file.
    -m,--mode <arg>           Execution Mode. Possible values: 'batch'
                                (Batch mode), 'gui' (Graphical User Interface
                                mode). Default value: 'batch'.
    -o,--output <arg>         Output file, where output is written. Default
                                value: the standard output.
    -s,--steps <arg>          An integer representing the number of
                                simulation steps. Default value: 150.

```

Dependiendo del valor dado para la opción `-m`, el método `start` invoca al método `startBatchMode` o al nuevo método `startGUIMode`. Ten en cuenta que a diferencia del modo `BATCH`, en el modo `GUI` el parámetro `-i` es opcional. Si se incluye el parámetro es necesario cargar el archivo correspondiente en el simulador (igual que en la práctica 4), de

modo que la interfaz gráfica tendrá un contenido inicial en este caso. Las opciones `-o` y `-s` se ignoran en el modo **GUI**. Recuerda que para crear la ventana en modo GUI tienes que usar:

```
SwingUtilities.invokeLater(new Runnable() {  
    @Override  
    public void run() {  
        new MainWindow(ctrl);  
    }  
});
```