

# Práctica 2:

## Interfaz gráfica para el simulador físico

**Fecha de entrega:** 17 de Abril 2023, 09:00h

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

### Control de Copias

Durante el curso se realizará control de copias de todas las prácticas, comparando las entregas de todos los grupos de TP2. 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 de la Facultad, que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres medidas siguientes:

1. Calificación de cero en la convocatoria de TP2 a la que corresponde la práctica o examen.
2. Calificación de cero en todas las convocatorias de TP2 del curso actual.
3. Apertura de expediente académico ante la *Inspección de Servicios de la Universidad*.

### 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 1 antes de hacer cambios en ella para la práctica 2.
3. Crea un nuevo paquete `simulator.view` para colocar en él todas las clases de la vista.
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.

### Descripción General de la Interfaz Gráfica para el Simulador Físico

En esta práctica vas a desarrollar una interfaz gráfica de usuario (GUI) para el simulador físico siguiendo el patrón de diseño modelo-vista-controlador (MVC). 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 grupos; (3) una tabla que muestra el estado de todos los cuerpos; y (4) una barra de estado en la que aparece más información, que detallaremos después. Además, incluye un diálogo que permite cambiar las leyes de fuerza de un grupo, y una ventana (que se abre de forma separada) para dibujar el estado de la simulación (similar al visor HTML que usaste en la primera práctica).

## Cambios en el Modelo y el Controlador

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

### Nueva funcionalidad

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

1. `public void reset()`: vacía el mapa de grupos y la lista de identificadores de los grupos llamando a sus métodos `clear` y pone el tiempo actual a `0.0`.
2. `public void setDeltaTime(double dt)`: cambia el valor del tiempo por paso actual (delta-time de aquí en adelante) a `dt`. Debe lanzar una excepción del tipo `IllegalArgumentException` si `dt` no es positivo.

Además, si no lo hiciste en la práctica 1, añade un método `toString()` a todas las clases que implementen la interfaz `ForceLaw` que devuelva una pequeña descripción de la correspondiente fuerza, por ejemplo:

- "Newton's Universal Gravitation with `G="+_G`
- "Moving towards `"+_c+" with constant acceleration "+_g`
- "No Force"

donde `_G`, `_c`, y `_g` son los atributos que almacenan los parámetros de las leyes de fuerza en las correspondientes clases. Modifica los builders `NoForceBuilder`, `NewtonUniversalGravitationBuilder`, y `MovingTowardsFixedPointBuilder` para que devuelvan un JSON con la siguiente estructura al llamar al método `getInfo()` (esta información se mostrará en la GUI):

```
{
  "type": "nf",
  "desc": "No force",
  "data": {}
}

{
  "type": "nlug",
  "desc": "Newton's law of universal gravitation",
  "data": {
    "G": "the gravitational constant (a number)"
  }
}

{
  "type": "mtfp",
  "desc": "Moving towards a fixed point",
  "data": {
    "c": "the point towards which bodies move (e.g., [100.0,50.0])",
    "g": "the length of the acceleration vector (a number)"
  }
}
```

Puedes hacer lo mismo con los builders de los cuerpos (aunque no sea necesario para esta práctica)

### La interfaz observer

Los observers implementan la siguiente interfaz, que incluye varios tipos de notificaciones (colócala en el paquete `simulator.model`):

```

public interface SimulatorObserver {
    public void onAdvance(Map<String, BodiesGroup> groups, double time);
    public void onReset(Map<String, BodiesGroup> groups, double time, double dt);
    public void onRegister(Map<String, BodiesGroup> groups, double time, double dt);
    public void onGroupAdded(Map<String, BodiesGroup> groups, BodiesGroup g);
    public void onBodyAdded(Map<String, BodiesGroup> groups, Body b);
    public void onDeltaTimeChanged(double dt);
    public void onForceLawsChanged(BodiesGroup g);
}

```

Los nombres de los métodos dan información sobre el significado de los eventos que notifican. En cuanto a los parámetros: `groups` es el mapa de grupos; `g` es un grupo de cuerpos; `b` es un cuerpo, `time` es el tiempo actual del simulación; y `dt` es el delta-time actual.

## Nuevos observadores

Modifica la clase `PhysicsSimulator` para que implemente `Observable<SimulatorObserver>` donde la interfaz `Observable<T>` está definida como:

```

public interface Observable<T> {
    void addObserver(T o);
    void removeObserver(T o);
}

```

Añade a la clase `PhysicsSimulator` una lista de observadores, que inicialmente es vacía, y añade los siguientes métodos para registrar/eliminar observadores:

1. `public void addObserver(SimulatorObserver o)`: añade el observador `o` a la lista de observadores, si es no está ya en ella.
2. `public void removeObserver(SimulatorObserver o)`: elimina el observador `o` de la lista de observadores.

## Mapa de grupos no modificables

La clase `PhysicsSimulator` debe pasar el mapa de grupos a los observadores como parámetro en alguno de los métodos de notificación. Sin embargo, con el objetivo de garantizar que no se puedan modificar desde fuera de la clase, podemos utilizar mapas no modificables. Supón que el atributo `_groups` contiene un mapa de grupos, podemos definir un nuevo mapa

```
private Map<String, BodiesGroup> _groupsR0;
```

e inicializarlo en el constructor como sigue (después de inicializar `_groups`):

```
_groupsR0 = Collections.unmodifiableMap(_groups);
```

Ahora, cada vez que queramos pasarle el mapa a un observador podemos usar `_groupsR0`, que contiene la misma información que `_groups` pero no puede ser modificado.

## Envío de notificaciones

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

1. Al final del método `addObserver` envía una notificación `onRegister` **solo al observador que se acaba de registrar**, para pasarle el estado actual del simulador.
2. Al final del método `reset`, envía una notificación `onReset` a **todos los observadores**.
3. Al final del método `addGroup` envía una notificación `onGroupAdded` a **todos los observadores**.

4. Al final del método `addBody` envía una notificación `onBodyAdded` a **todos los observadores**.
5. Al final del método `advance` envía una notificación `onAdvance` a **todos los observadores**.
6. Al final del método `setDeltaTime` envía una notificación `onDeltaTimeChanged` a **todos los observadores**.
7. Al final del método `setForceLaws` envía una notificación `onForceLawsChanged` a **todos los observadores**.

## Cambios en la clase `BodiesGroup`

Necesitamos modificar la clase `BodiesGroup` para reflejar información necesaria para la GUI. En primer lugar tenemos que añadir un método `getForceLawsInfo()` que devuelve la descripción de la correspondiente fuerza llamando a `_forceLaws.toString()` – puedes devolver también el objeto que representa la ley de fuerza, ya que es inmutable. En segundo lugar debemos poder recorrer el grupo de cuerpos `BodiesGroup g` como sigue:

```
for (Body b : g) {  
    // do something with b  
}
```

La solución más simple es hacer que `BodiesGroup` implemente `Iterable<Body>` de tal manera que su método `iterator()` devuelva `_bodies.iterator()` donde `_bodies` es la lista de cuerpos. Sin embargo, esto puede ser peligroso ya que los iteradores de las implementaciones de la interfaz `List` de Java (`ArrayList<T>`, `LinkedList<T>`, etc.) permiten eliminar elementos (la interfaz `Iterator<E>` tiene un método `remove()`). Para resolver este problema podemos definir una versión no modificable de `_bodies`:

```
List<Body> _bodiesRO;
```

e inicializarla en el constructor a lo siguiente (tras inicializar `_bodies`):

```
_bodiesRO = Collections.unmodifiableList(_bodies)
```

y, entonces, devolver `_bodiesRO.iterator()`. Una solución alternativa es devolver un *objeto decorado* para `_bodies.iterator()` cuyo método `remove()` simplemente lanza una excepción (de hecho esto es lo que hace la implementación de `Collections.unmodifiableList`):

```
return new Iterator<Body>() {  
    Iterator<Body> it = _bodies.iterator();  
  
    @Override  
    public Body next() { return it.next(); }  
  
    @Override  
    public boolean hasNext() { return it.hasNext(); }  
  
    @Override  
    public void remove() { throw new UnsupportedOperationException("..."); }  
};
```

## Cambios en el `Controller`

La clase `Controller` tiene que ser extendida con funcionalidad adicional (para evitar pasar el simulador a la GUI) como sigue:

1. `public void reset()`: llama al método `reset` del simulador.
2. `public void setDeltaTime(double dt)`: ejecuta `setDeltaTime` del simulador.

3. `public void addObserver(SimulatorObserver o)`: llama al método `addObserver` del simulador.
4. `public void removeObserver(SimulatorObserver o)`: llama al método `removeObserver` del simulador.
5. `public List<JSONObject> getForceLawsInfo()`: devuelve la lista generada por el método `getInfo()` de la factoría de leyes de fuerza. Será utilizado por la GUI para mostrar las leyes de fuerza disponibles. Modifica el método `getInfo()` de `BuilderBasedFactory` para devolver una lista de solo lectura utilizando `Collections.unmodifiableList` si no lo hace ya.
6. `public void setForcesLaws(String gId, JSONObject info)`: usa la factoría de leyes de fuerza actual para crear una nueva instancia de la ley de fuerza a partir de `info`. Tras ello, llama al simulador para modificar la ley de fuerza del grupo identificado por `gId`.
7. `public void run(int n)`: ejecuta el simulador `n` pasos, *sin mostrar nada en un `OutputStream`*. Usa `for(int i=0; i<n; i++) _sim.advance();`

## La interfaz gráfica de usuario

En esta sección describiremos las distintas clases de nuestra GUI.

### MainWindow

La ventana principal está representada por la siguiente clase. Lee el código y completa las partes que no están implementadas.

```
public class MainWindow extends JFrame {

    private Controller _ctrl;

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

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

        // TODO crear ControlPanel y añadirlo en PAGE_START de mainPanel
        // TODO crear StatusBar y añadirlo en PAGE_END de mainPanel

        // Definición del panel de tablas (usa un BoxLayout vertical)
        JPanel contentPanel = new JPanel();
        contentPanel.setLayout(new BoxLayout(contentPanel, BoxLayout.Y_AXIS));
        mainPanel.add(contentPanel, BorderLayout.CENTER);

        // TODO crear la tabla de grupos y añadirla a contentPanel.
        //      Usa setPreferredSize(new Dimension(500, 250)) para fijar su tamaño

        // TODO crear la tabla de cuerpos y añadirla a contentPanel.
        //      Usa setPreferredSize(new Dimension(500, 250)) para fijar su tamaño

        // TODO llama a Utils.quit(MainWindow.this) en el método windowClosing
        addWindowListener( ... );
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        pack();
    }
}
```

```

        setVisible(true);
    }
}

```

## ControlPanel

El panel de control es el responsable de la interacción entre el usuario y el simulador. Se corresponde con la barra de herramientas que aparece en la parte superior de la ventana de la Figura 1. Incluye los siguientes componentes: botones para interactuar con el simulador, un JSpinner para seleccionar los pasos de simulación deseados, y un JTextField para actualizar el delta-time (o mostrar el actual):

```

class ControlPanel extends JPanel implements SimulatorObserver {

    private Controller _ctrl;
    private JToolBar _toolBar;
    private JFileChooser _fc;
    private boolean _stopped = true; // utilizado en los botones de run/stop
    private JButton _quitButton;
    // TODO añade más atributos aquí ...

    ControlPanel(Controller ctrl) {
        _ctrl = ctrl;
        initGUI();
        // TODO registrar this como observador
    }

    private void initGUI() {
        setLayout(new BorderLayout());
        _toolBar = new JToolBar();
        add(_toolBar, BorderLayout.PAGE_START);

        // TODO crear los diferentes botones/atributos y añadirlos a _toolBar.
        //      Todos ellos han de tener su correspondiente tooltip. Puedes utilizar
        //      _toolBar.addSeparator() para añadir la línea de separación vertical
        //      entre las componentes que lo necesiten






        // Quit Button
        _toolBar.add(Box.createGlue()); // this aligns the button to the right
        _toolBar.addSeparator();
        _quitButton = new JButton();
        _quitButton.setToolTipText("Quit");
        _quitButton.setIcon(new ImageIcon("resources/icons/exit.png"));
        _quitButton.addActionListener((e) -> Utils.quit(this));
        _toolBar.add(_quitButton);

        // TODO crear el selector de ficheros
        _fc = ...

    }
    // TODO el resto de métodos van aquí...
}



```

La funcionalidad de los distintos botones es la siguiente:

- Cuando se pulsa el botón : (1) utiliza `_fc.showOpenDialog(Utils.getWindow(this))` para abrir el selector de ficheros para que el usuario pueda seleccionar el archivo de entrada; (2) si el usuario ha seleccionado un fichero, resetea el simulador utilizando `_ctrl.reset()` y carga el fichero seleccionado en el simulador.
- Cuando se pulsa el botón  crea una instancia de `ForceLawsDialog`, almacénala en un atributo, y llama a su método `open`. Esto permitirá al usuario cambiar la ley de fuerza de un grupo específico. A continuación se encuentra la descripción de `ForceLawsDialog`. Ten en cuenta que esta instancia ha de crearse una única vez, es decir, antes de crearla debes comprobar que el correspondiente atributo es `null`, en otro caso usa la instancia que has creado anteriormente.
- Cuando se pulsa el botón  crea una instancia de `ViewerWindow` (descripción a continuación). Esto permite al usuario ver una representación visual de la simulación. Ten en cuenta que el usuario puede tener varios visores abiertos al mismo tiempo.
- Cuando se pulsa el botón : (1) deshabilita todos los botones excepto el botón de stop , y cambia el valor del atributo `_stopped` a `false`; (2) actualiza el valor actual de delta-time del simulador con el especificado en el correspondiente `JTextField`; y (3) llama al método `run_sim` con el valor actual de pasos, especificado en el correspondiente `JSpinner`:

```
private void run_sim(int n) {
    if (n > 0 && !_stopped) {
        try {
            _ctrl.run(1);
        } catch (Exception e) {
            // TODO llamar a Utils.showErrorMsg con el mensaje de error que
            // corresponda
            // TODO activar todos los botones
            _stopped = true;
            return;
        }
        SwingUtilities.invokeLater(() -> run_sim(n - 1));
    } else {
        // TODO activar todos los botones
        _stopped = true;
    }
}
```

Debes completar el método `run_sim` como se indica en los comentarios. Fíjate que el método `run_sim` tal y como está definido garantiza que el interfaz no se quedará bloqueado. Para entender este comportamiento modifica el método `run_sim` para incluir una única instrucción `_ctrl.run(n)` — ahora, al comenzar la simulación, no verás los pasos intermedios, únicamente el estado final, además de que la interfaz estará completamente bloqueada.

- Cuando se pulsa el botón , actualiza el valor del atributo `_stopped` a `true`. Esto “detendrá” el método `run_sim` si hay llamadas en la cola de eventos de swing (observa la condición del método `run_sim`).
- La funcionalidad del botón  se proporciona como parte del código.

Debes capturar todas las posibles excepciones lanzadas por el controlador/simulador y mostrar el correspondiente mensaje utilizando `Utils.showErrorMsg`. En principio, los métodos de la clase `SimulatorObserver` están vacíos, excepto `onDeltaTimeChanged` que debe modificar el correspondiente `JTextField` con el valor actual de delta-time.

## StatusBar



La barra de estado es la responsable de mostrar información general sobre el simulador. Se corresponde con el área de la parte inferior de la ventana en la Figura 1. Lee el código y completa las partes que faltan. Es obligatorio añadir el tiempo de simulación y el número de grupos, pero puedes añadir más información si lo deseas.

```
class StatusBar extends JPanel implements SimulatorObserver {

    // TODO Añadir los atributos necesarios, si hace falta ...

    StatusBar(Controller ctrl) {
        initGUI();
        // TODO registrar this como observador
    }

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

        // TODO Crear una etiqueta de tiempo y añadirla al panel
        // TODO Crear la etiqueta de número de grupos y añadirla al panel
        // TODO Utilizar el siguiente código para añadir un separador vertical
        //
        //     JSeparator s = new JSeparator(JSeparator.VERTICAL);
        //     s.setPreferredSize(new Dimension(10, 20));
        //     this.add(s);
    }

    // TODO el resto de métodos van aquí...
}
```

## Info Table, GroupsTableModel and BodiesTableModel

Las tablas son las responsables de mostrar la información de los grupos/cuerpos. Tendremos una clase que incluya un JTable, que recibirá como parámetro el correspondiente modelo de la tabla, y dos clases para los modelos de los grupos y los cuerpos. El siguiente código implementa la creación de las tablas en MainWindow

```
new InfoTable("Groups", new GroupsTableModel(_ctrl));
new InfoTable("Bodies", new BodiesTableModel(_ctrl));
```

donde InfoTable, GroupsTableModel y BodiesTableModel están implementadas como sigue:

```
public class InfoTable extends JPanel {

    String _title;
    TableModel _tableModel;

    InfoTable(String title, TableModel tableModel) {
        _title = title;
        _tableModel = tableModel;
        initGUI();
    }

    private void initGUI() {
        // TODO cambiar el layout del panel a BorderLayout()
    }
}
```



```

        // TODO añadir un borde con título al JPanel, con el texto _title
        // TODO añadir un JTable (con barra de desplazamiento vertical) que use
        //      _tableModel
    }
}

class GroupsTableModel extends AbstractTableModel implements SimulatorObserver {

    String[] _header = { "Id", "Force Laws", "Bodies" };
    List<BodiesGroup> _groups;

    GroupsTableModel(Controller ctrl) {
        _groups = new ArrayList<>();
        // TODO registrar this como observador;
    }
    // TODO el resto de métodos van aquí ...
}

class BodiesTableModel extends AbstractTableModel implements SimulatorObserver {

    String[] _header = { "Id", "gId", "Mass", "Velocity", "Position", "Force" };
    List<Body> _bodies;

    BodiesTableModel(Controller ctrl) {
        _bodies = new ArrayList<>();
        // TODO registrar this como observer
    }
    // TODO el resto de métodos van aquí...
}

```

En `GroupsTableModel`, los métodos de `AbstractTableModel` deben usar el atributo `_groups` para proporcionar la información del contenido de la tabla. Además los métodos de `SimulatorObserver` deben mantener la lista `_groups` actualizada, es decir, debes actualizar `_groups` con los nuevos grupos en los métodos `onReset`, `onRegister`, `onBodyAdded`, y `onGroupAdded`, y llamar al método `fireTableStructureChanged()` para redibujar la tabla. De forma similar, en `BodiesTableModel`, los métodos de `AbstractTableModel` deben usar el atributo `_bodies` para proporcionar la información del contenido de la tabla, y los métodos de `SimulatorObserver` deben mantener la lista `_bodies` actualizada, es decir, actualiza la lista `_bodies` con los nuevos cuerpos en los métodos `onReset`, `onRegister`, `onBodyAdded`, y `onGroupAdded`, y llama al método `fireTableStructureChanged()` para redibujar la tabla, y al método `fireTableDataChanged()` en el método `onAdvance` para actualizar la información de la misma.

## ForceLawsDialog

La clase `ForceLawsDialog` es la responsable de implementar la ventana de diálogo que permite modificar la ley de fuerza de los distintos grupos (ver Figura 2):

```

class ForceLawsDialog extends JDialog implements SimulatorObserver {

    private DefaultComboBoxModel<String> _lawsModel;
    private DefaultComboBoxModel<String> _groupsModel;
    private DefaultTableModel _dataTableModel;
    private Controller _ctrl;
    private List<JSONObject> _forceLawsInfo;
}

```

```

private String[] _headers = { "Key", "Value", "Description" };

// TODO en caso de ser necesario, añadir los atributos aquí...

ForceLawsDialog(Frame parent, Controller ctrl) {
    super(parent, true);
    _ctrl = ctrl;
    initGUI();
    // TODO registrar this como observer;
}

private void initGUI() {

    setTitle("Force Laws Selection");
    JPanel mainPanel = new JPanel();
    mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
    setContentPane(mainPanel);

    // _forceLawsInfo se usará para establecer la información en la tabla
    _forceLawsInfo = ctrl.getForceLawsInfo();

    // TODO crear un JTable que use _dataTableModel, y añadirla al panel
    _dataTableModel = new DefaultTableModel() {
        @Override
        public boolean isCellEditable(int row, int column) {
            // TODO hacer editable solo la columna 1
        }
    };
    _dataTableModel.setColumnIdentifiers(_headers);

    _lawsModel = new DefaultComboBoxModel<>();
    // TODO añadir la descripción de todas las leyes de fuerza a _lawsModel
    // TODO crear un combobox que use _lawsModel y añadirlo al panel

    _groupsModel = new DefaultComboBoxModel<>();
    // TODO crear un combobox que use _groupsModel y añadirlo al panel

    // TODO crear los botones OK y Cancel y añadirlos al panel

    setPreferredSize(new Dimension(700, 400));
    pack();
    setResizable(false);
    setVisible(false);
}

public void open() {
    if (_groupsModel.getSize() == 0)
        return _status;
}

```

```

        // TODO Establecer la posición de la ventana de diálogo de tal manera que se
        // abra en el centro de la ventana principal

        pack();
        setVisible(true);
        return _status;
    }
    // TODO el resto de métodos van aquí...
}

```

El diálogo se crea/abre en `ControlPanel` cuando se pulsa sobre el correspondiente botón. Recuerda que se debe crear una única instancia de la ventana de diálogo (la primera vez que se pulse el botón), y después basta con llamar al método `open`. De esta forma el diálogo mantendrá su último estado. La funcionalidad a implementar es la siguiente:

1. En los métodos de `SimulatorObserver` debes mantener la lista de grupos actualizada en el combobox de grupos. Para ello tendrás que modificar `_groupsModel` como corresponda. Lo tienes que implementar en los métodos `onReset`, `onRegister`, y `onGroupAdded`. Como `_groupsModel` se mantiene actualizado, cada vez que el diálogo se abra, el combobox de grupos mostrará los grupos existentes en ese momento como opciones.
2. Cuando el usuario selecciona la *i*-ésima ley de fuerza (del correspondiente combobox), debes actualizar `_dataTableModel` para tener las claves y las descripciones en la primera y tercera columna respectivamente, lo que modificará el contenido de la correspondiente `JTable`. Para implementar este comportamiento (a) obtén el *i*-ésimo elemento de `_forceLawsInfo`, llámalo `info`; (b) obtén el valor asociado a la clave "data" de `info`, llámalo `data`; y (3) itera sobre `data.keySet()` y añade cada elemento a la primera columna y su valor (que es la descripción) en la tercera columna. Además debes almacenar el índice de la ley de fuerza seleccionada en un atributo ya que necesitarás acceder a ella más tarde, llámala `_selectedLawsIndex`.
3. Si el usuario pulsa el botón `Cancel`, simplemente pon el `_status` a 0 y haz el diálogo invisible.
4. Si el usuario hace click en `OK` (a) convierte la información en la tabla en un `JSONObject` que incluye la clave y el valor para cada fila en la tabla; (b) crea otro `JSONObject` que tiene una clave llamada "data" cuyo valor es el `JSONObject` del punto anterior y una clave "type" que es igual a la clave `type` de la ley seleccionada, es decir,

```

_forceLawsInfo.get(_selectedLawsIndex).getString("type");

```

y (c) llama al controlador para fijar esta ley en el grupo seleccionado en el combobox correspondiente. Si todos los pasos anteriores tienen éxito pon `_status` a 1 y haz el diálogo invisible, en otro caso captura cualquier excepción y muestra el mensaje de error correspondiente usando `Utils.showErrorMsg`.

## Viewer, SimulationViewer, and ViewerWindow

Este componente dibuja el estado de la simulación gráficamente en cada paso (ver Figura 3). Es implementado por dos clases: una clase llamada `ViewerWindow` que representa la ventana, y una clase llamada `Viewer` que hace la visualización (extiende una clase abstracta llamada `SimulationViewer` de tal forma que nos abstraemos de la implementación actual; notar que `SimulationViewer` extiende `JComponent` así que podemos tratar una instancia como un componente Swing). Lo siguiente es un esqueleto de `ViewerWindow`:

```

class ViewerWindow extends JFrame implements SimulatorObserver {

    private Controller _ctrl;
    private SimulationViewer _viewer;
    private JFrame _parent;
}

```

```

ViewerWindow(JFrame parent, Controller ctrl) {
    super("Simulation Viewer");
    _ctrl = ctrl;
    _parent = parent;
    intiGUI();
    // TODO registrar this como observador
}

private void intiGUI() {
    JPanel mainPanel = new JPanel(new BorderLayout());
    // TODO poner contentPane como mainPanel con scrollbars (JScrollPane)

    // TODO crear el viewer y añadirlo a mainPanel (en el centro)

    // TODO en el método windowClosing, eliminar 'this' de los observadores
    addWindowListener(new WindowListener() { ... });

    pack();
    if (_parent != null)
        setLocation(
            _parent.getLocation().x + _parent.getWidth()/2 - getWidth()/2,
            _parent.getLocation().y + _parent.getHeight()/2 - getHeight()/2);
    setVisible(true);
}
// TODO otros métodos van aquí....
}

```

Deberías completarlo de forma que los métodos de SimulatorObserver mantienen la lista de grupos/bodies actualizados en `_viewer`, i.e., llamando a los métodos `_viewer.addBody(b)` y `_viewer.addGroup(g)` desde los métodos `onReset`, `onRegister`, `onBodyAdded`, y `onGroupAdded`. Además, en el método `onAdvance` deberías llamar a `_viewer.update()` para redibujar el nuevo estado. La clase `Viewer.java` es dada sin parte de su funcionalidad, lee todos los comentarios `TODO` dentro del código y complétalos – más información será explicada en las clases/laboratorios.

## Modificar 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 1) y en modo GUI. Esta opción es opcional con un valor predeterminado que inicia el modo GUI:

```

usage: simulator.launcher.Main [-dt <arg>] [-fl <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.
-fl,--force-laws <arg>  Force laws to be used in the simulator. Possible
                        values: 'nlug' (Newton's law of universal
                        gravitation), 'mtfp' (Moving towards a fixed
                        point), 'nf' (No force). You can provide the
                        'data' json attaching :{...} to the tag, but

```

	without spaces.. 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: 'gui'.
-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 1), de modo que la interfaz gráfica tendrá un contenido inicial en este caso. Las opciones -o y -s se ignoran en el modo GUI. Para crear la ventana en modo GUI debes usar:

```
SwingUtilities.invokeLater(() -> new MainWindow(ctrl));
```

## Figuras

Esta sección incluye 3 capturas de pantalla de las diferentes ventanas.

Figura 1

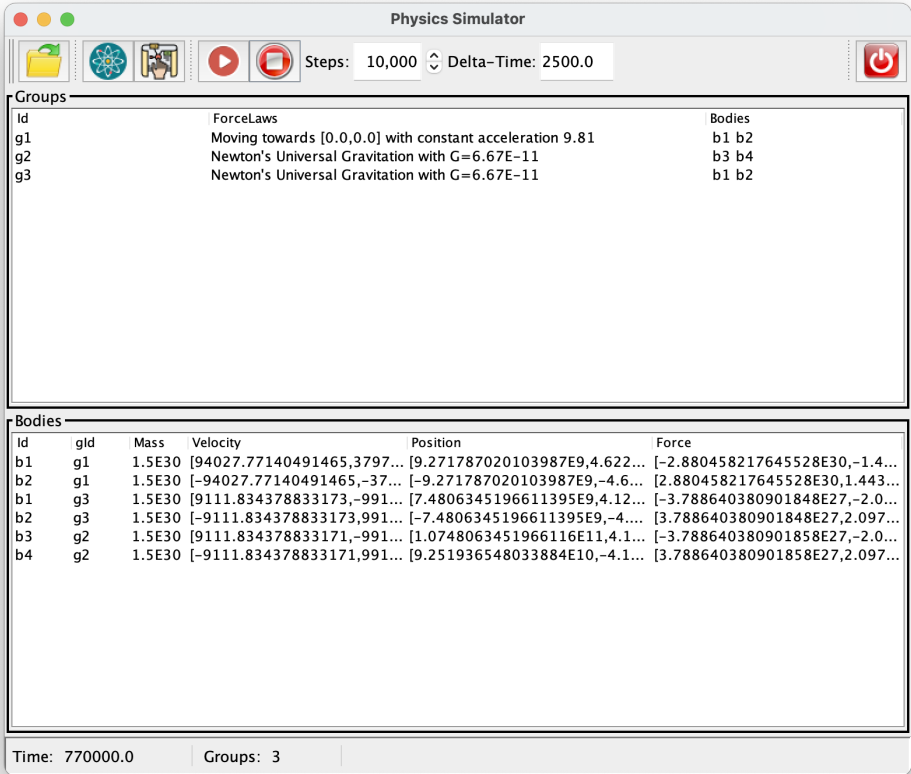


Figura 2

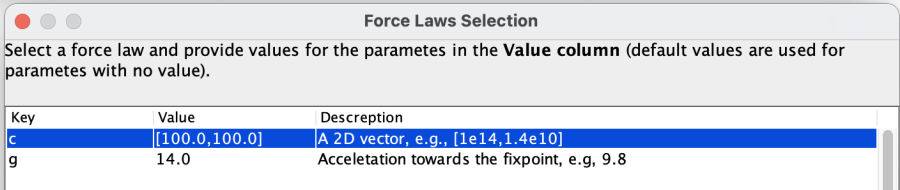


Figura 3

