

Tecnología de la Programación

Presentación de la Práctica 5

(Basado en la práctica de Samir Genaim)

Ana M. González de Miguel (ISIA, UCM)

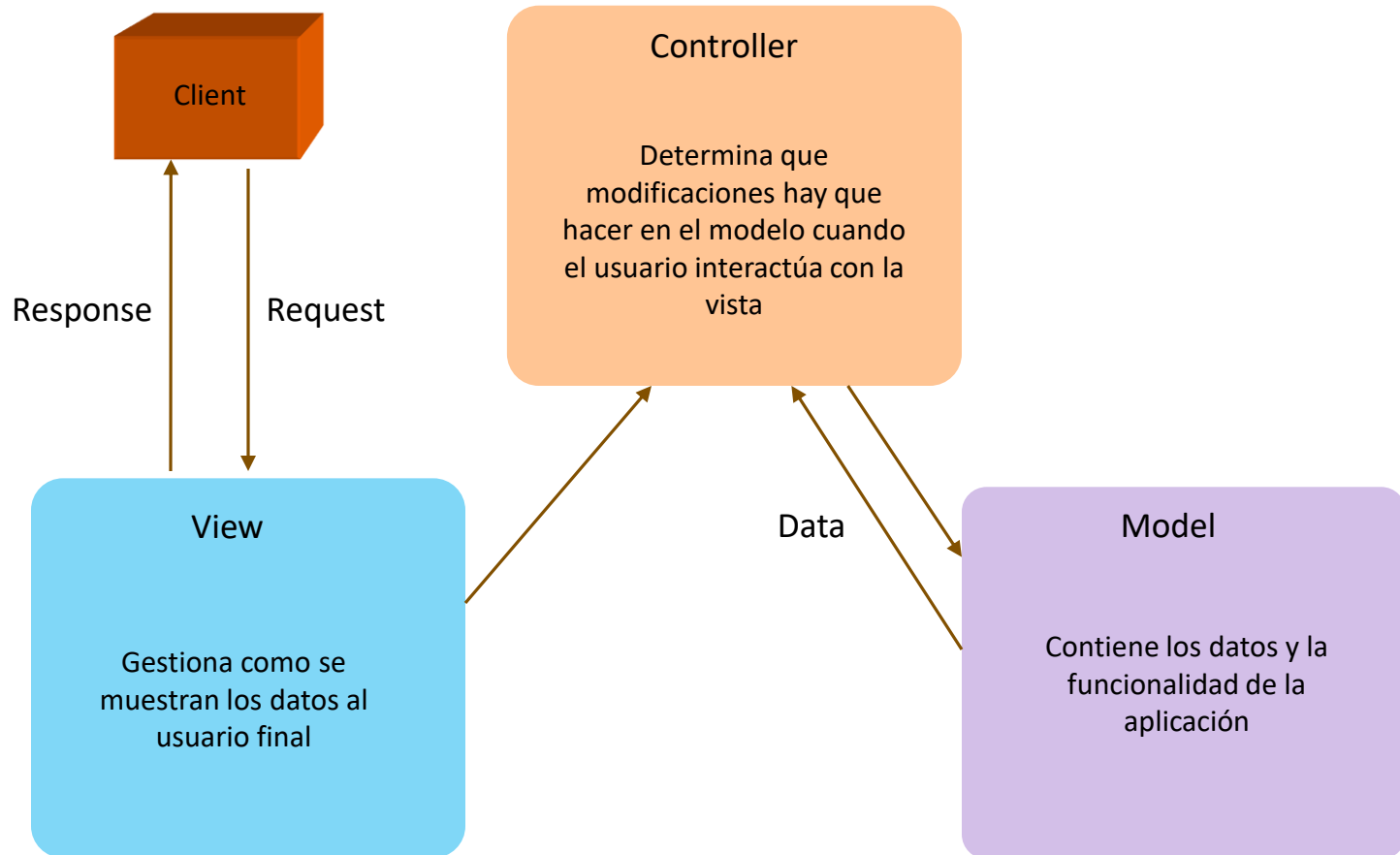
Índice

1. Introducción.
2. Descripción General de la Práctica.
3. Cambios en el Modelo.
4. Preparación de Controller.
5. La Interfaz Gráfica de Usuario.
6. Cambios en la clase Main.

1. Introducción

- ✓ Los objetivos cubiertos son: **diseño orientado a objetos, MVC e interfaces gráficas de usuario con Swing.**
- ✓ Fecha de entrega: **23 de Abril** a las 9:00.
- ✓ Las siguientes instrucciones son obligatorias:
 - Lee el enunciado completo de la práctica.
 - Haz una copia de la práctica 4 antes de empezar.
 - Crea un nuevo paquete **simulator.view** para colocar en él todas las clases necesarias de las vistas.
 - Utiliza exactamente la misma estructura de paquetes y nombres de clases usados en el enunciado.
 - No es posible utilizar ninguna herramienta para la generación automática de interfaces gráficas de usuario.
 - Para la entrega sube un fichero **zip** del proyecto con todos los subdirectorios excepto **bin**. Otros formatos no están permitidos.

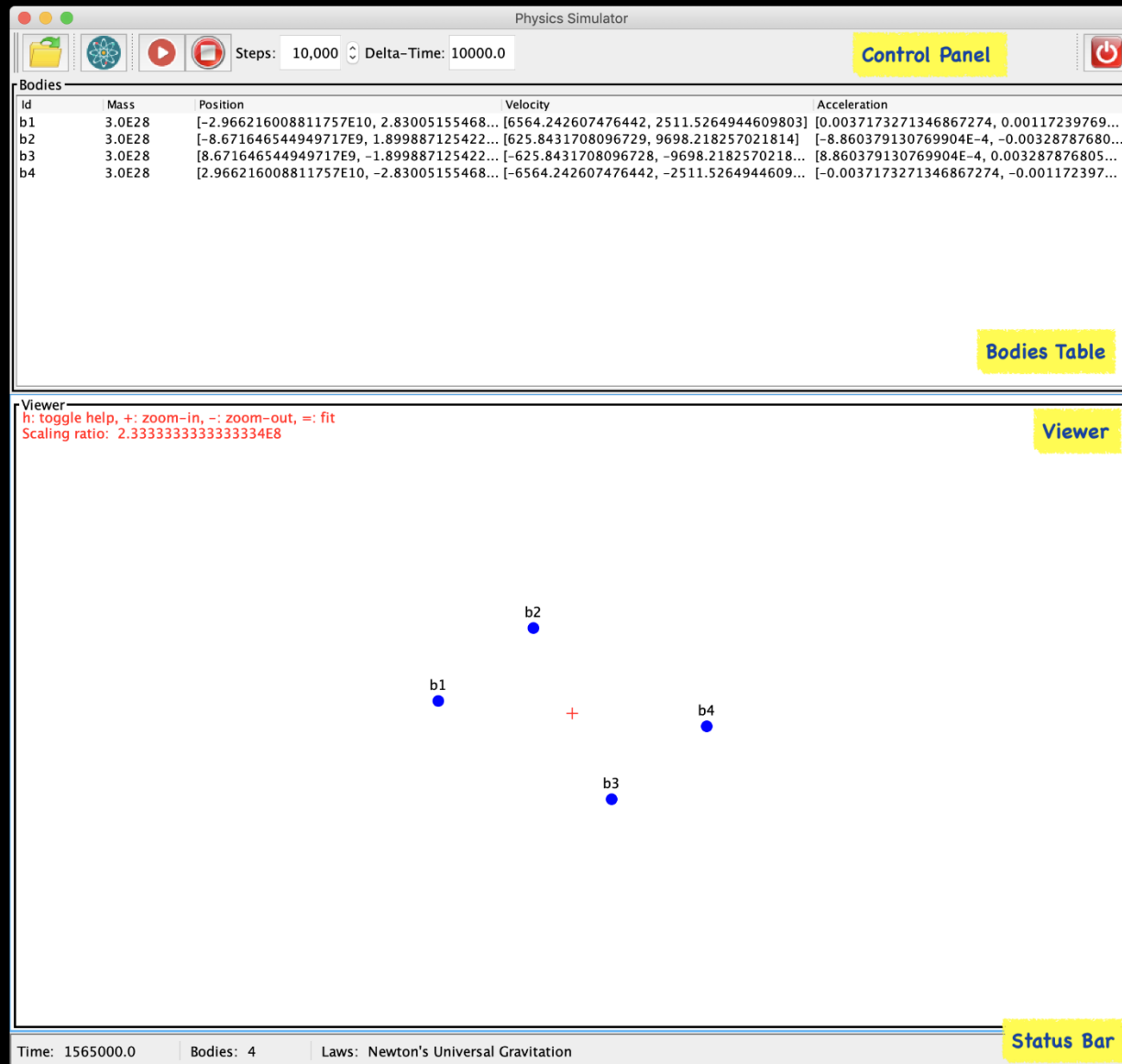
Arquitectura MVC



2. Descripción General de la Práctica

- ✓ En esta práctica vas a desarrollar una GUI para el simulador físico utilizando el patrón de diseño MVC.
- ✓ En la siguiente transparencia puedes ver la GUI que hay que construir.
- ✓ Esta interfaz tiene una **ventana principal** con cuatro componentes:
 - Un **panel de control** para interactuar con el simulador.
 - Una **tabla** que muestra el estado de todos los cuerpos.
 - Un **visor** (*viewer*) en el que aparecen dibujados los cuerpos en cada paso de simulación.
 - Una **barra de estado** en la que aparece más información.

Ventana Principal y Componentes de la GUI



3. Cambios en el Modelo

✓ Introduce los siguientes métodos en la clase *PhysicsSimulator*:

- `public void reset()` // vacía la lista de cuerpos y pone el tiempo a 0,0.
- `public void 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*, *Falling-ToCenterGravity* y *NoGravity* que devuelva una breve descripción (cadena) de las leyes de gravedad correspondientes.

- ✓ A continuación explicamos como modificar el modelo para usar el patrón MVC permitiendo que los **observadores** reciban del modelo notificaciones de determinados eventos.
- ✓ Representamos los observadores a través de la siguiente interfaz que incluye varios tipos de notificaciones y 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; *dt* es el delta-time actual (tiempo real de cada paso de simulación); *gLawsDesc* es un *string* que describe las leyes de gravedad actuales (que se obtiene invocando al método *toString()* de la ley de gravedad actual).
- ✓ 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.
```

- ✓ 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.

## 4. Preparación de Controller

- ✓ Cambia la constructora de *Controller* 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.`

## 5. La Interfaz Gráfica de Usuario

- ✓ 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.

- ✓ 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;
 // Añade atributos para JToolBar, botones, etc.


 ControlPanel(Controller ctrl) {
 _ctrl = ctrl;
 _stopped = true;
 initGUI();
 _ctrl.addObserver(this);
 }
 private void initGUI() {
 // Construye la tool bar con todos sus botones, etc.
 }
 // Añade métodos privados/protegidos
```


```
private void run_sim(int n) {
 if (n > 0 && !_stopped) {
 try {
 _ctrl.run(1);
 } catch (Exception e) {
 // Muestra el error con una ventana JOptionPane
 // Pon enable todos los botones
 _stopped = true;
 return;
 }
 SwingUtilities.invokeLater(new Runnable() {
 @Override
 public void run() {
 run_sim(n-1);
 } });
 } else {
 _stopped = true;
 // Pon enable todos los botone }
 }
 // Añade los métodos de SimulatorObserver
}
```

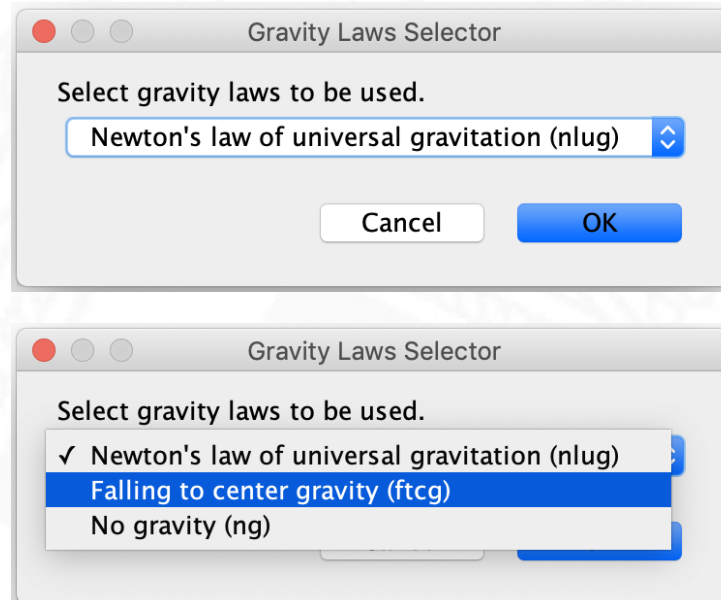


- ✓ Es necesario completar el método *initGUI()* para crear todas las componentes del panel (botones, selector de número de pasos, etc.).
- ✓ Puedes encontrar los iconos en el directorio **resources/icons**. Por ejemplo,
- ✓ Como selector de pasos usa un *JSpinner* y para el valor Delta-Time usa un *JTextField*.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





```
_loadButton.setToolTipText("Load bodies file into the editor");
```

- ✓ 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; 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 , y cambia el valor del campo `_stopped` a `false`; (2) pon el delta-time actual del simulador al especificado por el correspondiente campo de texto; y (3) llama al método `run_sim` con el valor actual del número de pasos según el *JSpinner*. Es necesario completar el método `run_sim` para que active de nuevo todos los botones al concluir la ejecución de la simulación. Observa que el método `run_sim` anterior garantiza que la interfaz no se bloquea (prueba a modificar el cuerpo de `run_sim` 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_sim` si hay llamadas pendientes en la cola (véase la condición en el bucle del método `run_sim`).
- 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*). Haz esta modificación con *invokeLater*.

- ✓ La **tabla de cuerpos** 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 de tabla y se actualice la tabla;
  - (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 {

 // Añade los nombres de columnas
 private List<Body> _bodies;
 BodiesTableModel(Controller ctrl) {
 _bodies = new ArrayList<>();
 ctrl.addObserver(this);
 }
 @Override
 public int getRowCount() {
 // Completa
 }
 @Override
 public int getColumnCount() {
 // Completa
 }
 @Override
 public String getColumnName(int column) {
 // Completa
 }
}
```

```
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
 // Completa
}
// Añade métodos de SimulatorObserver
```

- ✓ 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). Haz estas actualizaciones con *invokeLater*.
- ✓ La tabla en sí viene dada por la clase *BodiesTable*:

```
public class BodiesTable extends JPanel {
 BodiesTable(Controller ctrl) {
 setLayout(new BorderLayout());
 setBorder(BorderFactory.createTitledBorder(
 BorderFactory.createLineBorder(Color.black, 2), "Bodies",
 TitledBorder.LEFT, TitledBorder.TOP));
 // Completa
 }
}
```

✓ Es necesario que completes el código anterior:

- (1) creando una instancia de *BodiesTableModel* que se le pase a la *JTable*.
- (2) añadiendo la *JTable* al panel (es decir, a *this*) con un *JScrollPane*.



- ✓ El componente **viewer** muestra de forma gráfica el estado de todos los cuerpos en cada paso de la simulación.
- ✓ Le 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 static final int _WIDTH = 1000;
 private static final int _HEIGHT = 1000;
 // Añade constantes para los colores
 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() {
 // Suma border con title
 _bodies = new ArrayList<>();
 _scale = 1.0; _showHelp = true;
 }
}
```



```
addKeyListener(new KeyListener() {
 // Completa con métodos de la interfaz
 @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() {
 // Completa con métodos de la interfaz
 @Override
 public void mouseEntered(MouseEvent e) {
 requestFocus();
 }
});
}

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);
 // Usa 'gr' para dibujar, no 'g'
 _centerX = getWidth() / 2;
 _centerY = getHeight() / 2;
```

```
// Dibuja una cruz en el centro
// Dibuja los bodies
// Dibuja help si _showHelp es true
}

// Añade otros metodos
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;
}

// Añade metodos de SimulatorObserver
}
```

## ✓ A continuación explicamos el código:

- 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()*.



- ✓ 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; // para current time
 private JLabel _currLaws; // para gravity laws
 private JLabel _numOfBodies; // para number of bodies

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

```
private void initGUI() {
 this.setLayout(new FlowLayout(FlowLayout.LEFT));
 this.setBorder(BorderFactory.createBevelBorder(1));
 // Completa para construir la barra de estado
}
// Añade private/protected methods
// Añade SimulatorObserver methods
}
```

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



- ✓ La **ventana principal** viene dada por una clase llamada *MainWindow* que extiende a *JFrame*:

```
public class MainWindow extends JFrame {
 // Añade atributos para todos los componentes (clases)
 Controller _ctrl;
 public MainWindow(Controller ctrl) {
 super("Physics Simulator");
 _ctrl = ctrl;
 initGUI();
 }
 private void initGUI() {
 JPanel mainPanel = new JPanel(new BorderLayout());
 setContentPane(mainPanel);
 // Completa el método para construir la GUI
 }
 // Añade 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.

## 6. 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);
 }
});
```