Assignment 2: Graphical User Interface for the Physics Simulator

Submission date: 14th of May 2021, 23:00

Objective: Object Oriented Design, Model-View-Controller, Building Graphical User Interfaces with Swing.

1. Copy detection

For each of the TP assignments, all the submissions from all the different TP groups will be checked using anti-plagiarism software, by comparing all of them pairwise and by searching to see if any of the code of any of them is copied from other sources on the Internet[1]. Any plagiarism detected will be reported to the *Comité de Actuación ante Copias* which, after interviewing the student or students in question, will decide whether further action is appropriate, and if so, will propose one of the following sanctions:

- A grade of zero for the TP-course exam session (*convocatoria*) to which the assignment belongs.
- A grade of zero for all the TP-course exam sessions (*convocatorias*) for that year.
- Opening of disciplinary proceedings (*apertura de un expediente académico*) with the relevant university authority (*Inspección de Servicios*).

2. General instructions

The following instruction are strict, you **MUST** follow them.

1. Read the whole assignment before you start coding.

[1]If you decide to store your code in a remote repository, e.g. in a free version-control system with a view to facilitating collaboration with your lab partner, make sure your code is not in reach of search engines. If you are asked to provide your code by anyone other than your course lecturer, e.g. an employer of a private academy, you must refuse.

2. Make a copy of your assignment 1 before you make any change for assignment 5.
3. Create a new package simulator.view, this is where you should place all view classes.
4. You have to strictly follow the package structure and class names that we suggest.
5. You are not allowed to use any tool for automatically generating graphical user interfaces.
6. Submit a **zip** file of the project's directory, including all subdirectories except the **bin** directory. **Other formats (e.g., `7zip`, `rar`, etc.) are not accepted**.

3. Overview of the physics simulator

In this assignment you will develop a graphical user interface (`GUI`), following the model-view-controller design pattern, for the physics simulator. The overall view of the `GUI` is depicted in Figure 1. It consists of a main window that includes 4 components: (1) a control panel for interacting with the simulator; (2) a table where the states of all bodies are displayed; (3) a viewer where the universe is drawn graphically in each simulation step; (4) a status bar where some other information is displayed.

Sections 4 and 5 describe the changes required in the model and the controller, respectively; Section 6 describes the functionality and provides implementation details of the `GUI`; and Section 7 describes the changes required in the Main class.

4. Preparing the Model

This section describes the changes in the model that should be done in order to use the `MVC` design pattern and to add some new functionality.

4.1. Additional Functionality

Add the following methods to the class PhysicsSimulator (if you do not have them already):

- public void reset(): clears the list of bodies and sets the current time to 0.0.
- public void setDeltaTime(double dt): changes the current value of the delta-time (i.e., the *real time per step*) to dt. It should throw an IllegalArgumentException exception exception if the value is not valid.
- public void setForceLaws(ForceLaws forceLaws): changes the force laws of the simulator to forceLaws. It should throw an IllegalArgumentException if the value is not valid (i.e., null).

In addition, change all force laws classes (NewtonUniversalGravitation, MovingTowardsAFixedPoint, and NoForce) to include a toString() method that returns a short description of the corresponding force laws, for example:

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

where _G, _c, and _g are fields of the corresponding classes that stores the force laws parameters (change them to refer to your fields if you have used different names).

Change NewtonUniversalGravitationBuilder, MovingTowardsFixedPointBuilder, and NoForceBuilder such that a call to getBuilderInfo return the corresponding JSON structure as follows (this information will be shown in the GUI):

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

{
      "type": "mtfp",
      "data": {
            "c": "the point towards which bodies move (a json list of 2 numbers, e.g., [100.0,50.0])",
            "g": "the length of the acceleration vector (a number)"
      },
      "desc": "Moving towards a fixed point"
}

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

4.2. Using MVC in the Model

This section describes how to modify the model in order to use the MVC design pattern, i.e., to allow observers to receive notifications on specific events.

The Observer Interface

Observers are represented by the following interface, which includes several kinds of notifications, place it in the package simulator.model:

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

Method names carry information on the meaning of the corresponding events, let us explain the meaning of the different parameters: bodies is the current list of bodies; b is a body, time is the current time of the simulator; dt is the current delta-time, i.e., the real time per step; fLawsDesc is a description of the current force laws (obtained by calling toString of the current force laws).

Adding an Observer

Change the class PhysicsSimulator to maintain a list of observers (as a field), which is initially empty, and add the following method for registering an observer:

- public void addObserver(SimulatorObserver o): add o to the list of observers, if it is not there already.

Optionally, change PhysicsSimulator to implement Observable<SimulatorObserver> where

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

Sending Notifications

Change the class PhysicsSimulator in order to send notifications as follows:

- At the end of method addObserver, send an onRegister notification **only to the new observer** in order to pass it the current state of the simulator.
- At the end of method reset, send an onReset notification to **all observers**.
- At the end of method addBody, send an onBodyAdded notification to **all observers**.
- At the end of method advance, send an onAdvance notification to **all observers**.
- At the end of method setDeltaTime, send an onDeltaTimeChanged notification to **all observers**.
- At the end of method setForceLaws, send an onForceLawsChanged notification to **all observers**.

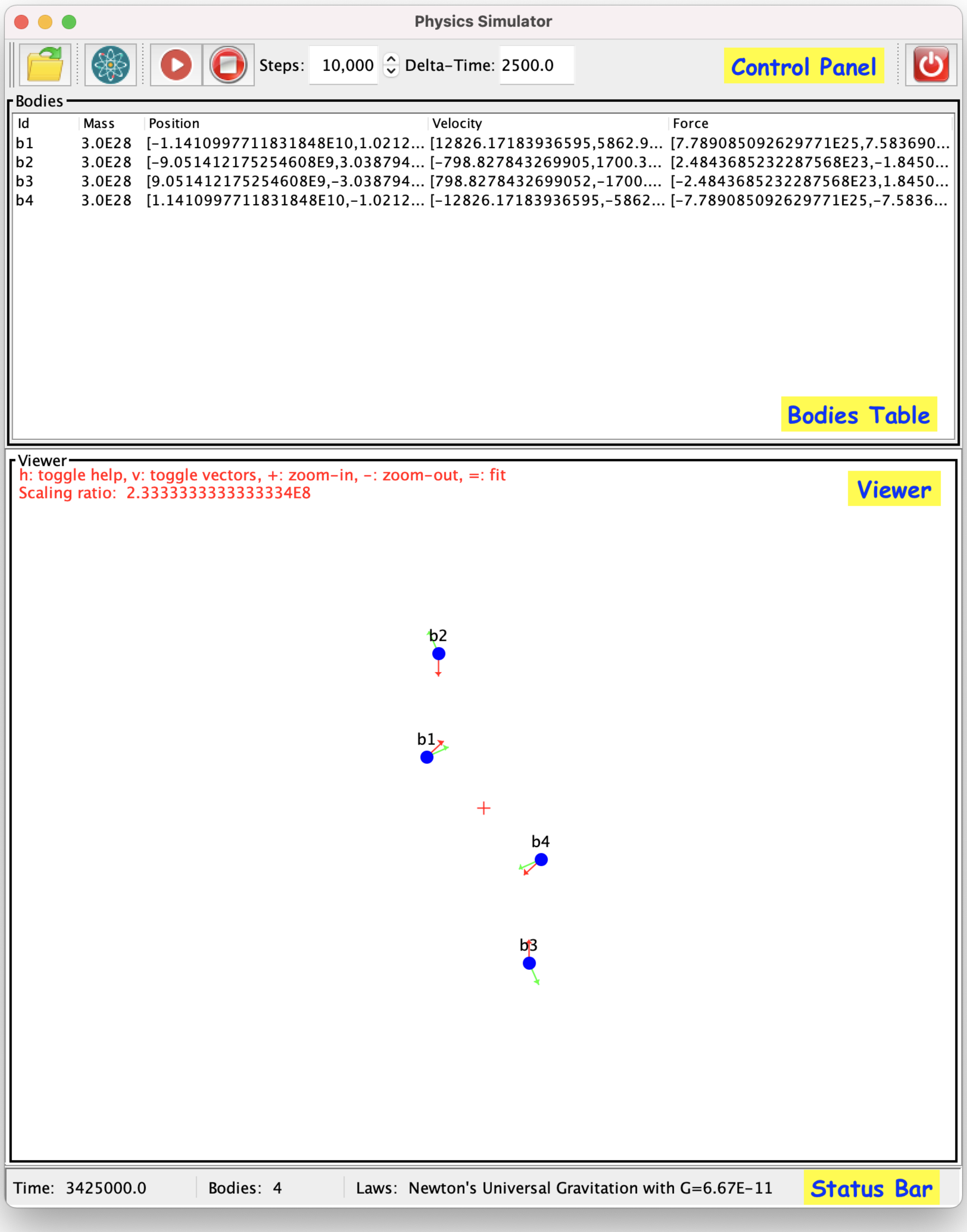
5. Preparing the Controller

The controller needs to be extended with some additional functionality as well. First change the constructor to receive a new parameter of type Factory<ForceLaws> and stores it in a corresponding field, and then add the following methods:

- public void reset(): calls reset of the simulator.
- public void setDeltaTime(double dt): calls setDeltaTime of the simulator.
- public void addObserver(SimulatorObserver o): calls addObserver of the simulator.
- public void run(int n): runs the simulator for n steps, **without printing anything to the console**. This is optional, you can also use current run method (when calling it from the GUI) providing it an OutputStream that does not print anything:

```
new OutputStream() {
      @Override
       public void write(int b) throws IOException {
       };
}
```

- public List<JSONObject> getForceLawsInfo() returns the list returned by calling getInfo() of the force laws factory. This will be used in the GUI to show the available force laws and allow changing them.
- public void setForcesLaws(JSONObject info): uses the current force laws factory to create a new force laws object as indicated in info, and then changes the simulator's force laws to the new one.



Steps: 10,000 Delta-Time: 2500.0

Bodies

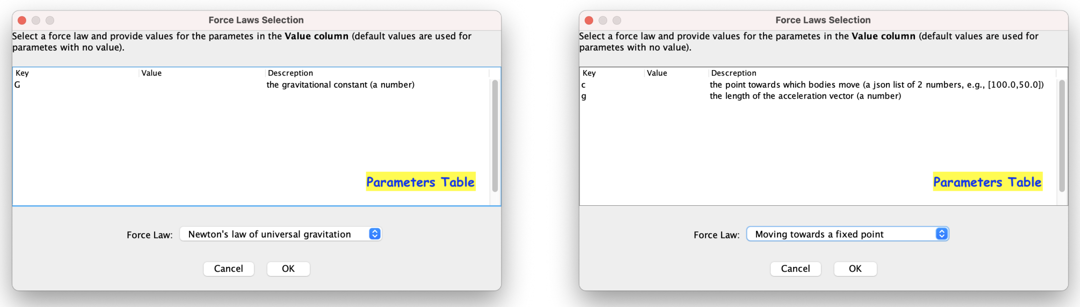
Id	Mass	Position	Velocity	Force
b1	3.0E28	[-1.1410997711831848E10,1.0212...	[12826.17183936595,5862.9...	[7.789085092629771E25,7.583690...
b2	3.0E28	[-9.051412175254608E9,3.038794...	[-798.827843269905,1700.3...	[2.4843685232287568E23,-1.8450...
b3	3.0E28	[9.051412175254608E9,-3.038794...	[798.8278432699052,-1700....	[-2.4843685232287568E23,1.8450...
b4	3.0E28	[1.1410997711831848E10,-1.0212...	[-12826.17183936595,-5862...	[-7.789085092629771E25,-7.5836...

Viewer

h: toggle help, v: toggle vectors, +: zoom-in, -: zoom-out, =: fit
Scaling ratio: 2.3333333333333334E8

Time: 3425000.0 Bodies: 4 Laws: Newton's Universal Gravitation with G=6.67E-11

Figure 1: The Graphical User Interface. Yellow boxes are annotations, not part of the `GUI`.



Force Laws Selection

Select a force law and provide values for the parametes in the **Value column** (default values are used for parametes with no value).

Key	Value	Descreption
G		the gravitational constant (a number)

Force Law: Newton's law of universal gravitation

Force Laws Selection

Select a force law and provide values for the parametes in the **Value column** (default values are used for parametes with no value).

Key	Value	Descreption
c		the point towards which bodies move (a json list of 2 numbers, e.g., [100.0,50.0])
g		the length of the acceleration vector (a number)

Force Law: Moving towards a fixed point

Figure 2: Dialog box for changing force laws. Yellow boxes are annotations, not part of the GUI.

6. The Graphical User Interface

The overall view of the GUI is depicted in Figure 1. It consists of a main window that includes 4 components: (1) a control panel for interacting with the simulator; (2) a table where the state of all bodies is displayed; (3) a viewer where the universe is drawn graphically in each simulation step; (4) a status bar where some other information is displayed.

In what follows we describe the implementation details of all parts. We first describe the components, and then how to organize them in a main window. The main window will be represented by a class that extends JFrame, and the other components by classes that extend JPanel (or JComponent). This allows treating all components as Swing components in the main window, which in turn allows changing one implementation by another without making deep modifications to the code.

6.1. Control Panel

The control panel is the one responsible on the user-simulator interaction. It is represented by a class called 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() {
      // TODO build the tool bar by adding buttons, etc.
   }

   // other private/protected methods
   // ...
   private void run_sim(int n) {
      if ( n>0 && !_stopped ) {
         try {
             _ctrl.run(1);
         } catch (Exception e) {
             // TODO show the error in a dialog box
             // TODO enable all buttons
             _stopped = true;
             return;
         }
         SwingUtilities.invokeLater( new Runnable() {
              @Override
              public void run() {
                 run_sim(n-1);
              }
         });
      } else {
         _stopped = true;
         // TODO enable all buttons
      }
   }

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

Complete method initGUI() to create and add all components to the panel (buttons, steps selector, etc.). Corresponding icons are available in the directory resources/icons. For the *steps* selector use a JSpinner and for the *Delta-Time* area use a JTextField. All buttons should have tooltips to describe the corresponding operations, and have the following functionality:

- When is clicked: (1) ask the user to select a file using a JFileChooser; (2) reset the simulator using _ctrl.reset(); and (3) load the selected file into the simulator by calling _ctrl.loadBodies(...).

- When is clicked: (1) open a dialog box and ask the user to select one of the available force laws – see Figure 2; and (2) once selected, change the force laws of the simulator to the chosen one (using _ctrl.setForceLaws(...)). In order to get information on the available force laws you can use _ctrl.getForceLawsInfo(). The combo-box should include the list of all available force laws, where the description of each is obtained from the value of the key "desc" of the corresponding JSONObject that describes the force laws. Once a force law is selected in the combo-box, the parameters table should change to include a list of corresponding parameters (the keys of the corresponding "data" section) so the use can provide corresponding values: the first column is the parameter name, the second is where the user provide values, and the this is a description that is taken from the value of the corresponding key in the "data" section. The user can edit only the "Values" column. You should display a corresponding error message (e.g., using JOptionPane.showMessageDialog) if the change of force laws did not succeed.

- When is clicked: (1) disable all buttons, except the stop button , and set the value of _stopped to false (2) set the current delta-time of the simulator to the one specified in the corresponding text field; and (3) call method run_sim with the current value of steps as specified in the JSpinner. You should complete method run_sim to enable all buttons again once the execution is over. Note that method run_sim as provided above guarantees that the interface will not block, in order to understand this behaviour change the body of method run_sim a single instruction _ctrl.run(n) — you will not see the intermediate steps, only the final result, and in the meantime the interface will be completely blocked.

- When is clicked, change the value of field _stopped to true, this will "stop" method run if there are calls in the queue (see the condition of method run).

- When is clicked, ask for the user's confirmation and then exit using System.exit(0).

In addition, catch all exceptions thrown by the controller/simulator and show a corresponding message using a dialog box (e.g., using JOptionPane.showMessageDialog). In the observer methods modify the value of delta-time in the corresponding JTextField when needed (i.e., in onRegister, onReset, and onDeltaTimeChanged).

6.2. Bodies Table

This component displays the state of bodies using a JTable – each row corresponds to a body. To implement this use two classes: (1) a class for a table model, that is also an observer, such that when the state of the simulator changes it will refresh the table; and (2) a class that creates a JTable and assigns it a corresponding table model.

The table model class is represented by a class called BodiesTableModel:

```
public class BodiesTableModel extends AbstractTableModel
                    implements SimulatorObserver {

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

   BodiesTableModel(Controller 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
   // ...
}
```

In the observer methods, when the state changes (e.g., in onAdvance, onRegister, onBodyAdded, and onReset), update the value of the field _bodies and then call fireTableStructureChanged() in order to notify the corresponding JTable that something has changed (it will then redraw the table).

The table class is represented by a class called 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));

         // TODO complete
   }
}
```

You need to complete the code to (1) create an instance of BodiesTableModel and pass it to a corresponding JTable; and (2) add the JTable to the panel (this) with a JScrollPane.

6.3. Viewer

This component draws the state of all bodies graphically in each simulation step. It is implemented by a class called Viewer that extends JComponent and overwrites method paintComponent (we could extend JPanel instead). This method is called by Swing whenever it needs to redraw the component. This class is also an observer, so when the state of the simulator changes we will ask Swing to redraw the component by calling method repaint() — this will automatically call paintComponent.

```
public class Viewer extends JComponent implements SimulatorObserver {

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

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

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

   _bodies = new ArrayList<>();
   _scale = 1.0;
   _showHelp = true;
   _showVectors = true;
   addKeyListener(new KeyListener() {
      // ...
      @Override
      public void keyPressed(KeyEvent e) {
         switch (e.getKeyChar()) {
            case '-':
               _scale = _scale * 1.1;
               repaint();
               break;
            case '+':
               _scale = Math.max(1000.0, _scale / 1.1);
               repaint();
               break;
            case '=':
               autoScale();
               repaint();
               break;
            case 'h':
               _showHelp = !_showHelp;
               repaint();
               break;
            case 'v':
               _showVectors = !_showVectors;
               repaint();
               break;
            default:
         }
      }
   });

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

@Override
protected void paintComponent(Graphics g) {
   super.paintComponent(g);

   // use 'gr' to draw not 'g' --- it gives nicer results
   Graphics2D gr = (Graphics2D) g;
   gr.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);
   gr.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                      RenderingHints.VALUE_TEXT_ANTIALIAS_ON);



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

   // TODO draw a cross at center
   // TODO draw bodies (with vectors if _showVectors is true)
   // TODO draw help if _showHelp is true
}

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

private void autoScale() {
   double max = 1.0;

   for (Body b : _bodies) {
      Vector2D p = b.getPosition();
      max = Math.max(max, Math.abs(p.getX()));
      max = Math.max(max, Math.abs(p.getY()));
   }

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

// This method draws a line from (x1,y1) to (x2,y2) with an arrow.
// The arrow is of height h and width w.
// The last two arguments are the colors of the arrow and the line
private void drawLineWithArrow(//
   Graphics g, //
   int x1, int y1, //
   int x2, int y2, //
   int w, int h, //
   Color lineColor, Color arrowColor) {

   int dx = x2 - x1, dy = y2 - y1;
   double D = Math.sqrt(dx * dx + dy * dy);
   double xm = D - w, xn = xm, ym = h, yn = -h, x;
   double sin = dy / D, cos = dx / D;

   x = xm * cos - ym * sin + x1;
   ym = xm * sin + ym * cos + y1;
   xm = x;

   x = xn * cos - yn * sin + x1;
   yn = xn * sin + yn * cos + y1;
   xn = x;

   int[] xpoints = { x2, (int) xm, (int) xn };
   int[] ypoints = { y2, (int) ym, (int) yn };

   g.setColor(lineColor);
```

```
        g.drawLine(x1, y1, x2, y2);
        g.setColor(arrowColor);
        g.fillPolygon(xpoints, ypoints, 3);
    }

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

Let us explain the different parts of this code:

- Fields _centerX and _centerY represent the origin position in this component, i.e., width and the height of the component by 2 – see method paintComponent.
- Field _bodies is the current list of bodies for drawing them, this list should be updated when the state changes.
- Field _scale is used to scale the universe so we can draw it within the area of the component (since the universe usually uses much bigger coordinates). Its value can be controlled by the user. You can see (in the switch statements) that when key '+' is pressed it is increased, when key '-' is pressed it is decreased, and when key '=' is pressed its value is calculated automatically by calling method autoScale.
- Field _showHelp indicates if the help text (in the left-top corner) should be shown, its value is changed when key 'h' is pressed.
- Field _showVectors indicates if the velocity/force vectors of each body are shown, its value is changed when key 'v' is pressed.
- The call to addKeyListener in the constructor registers a listener to capture keyboard events when this component has the focus, and the call to addMouseListener registers a listener to capture mouse events (it requests the focus when the mouse enters the component).

You need to complete method paintComponent in order to (1) draw a cross at the center; (2) draw the help message if _showHelp is true; and (3) draw the bodies where the velocity and force vectors are drawn if _showVectors is true (use two different colors, e.g., red and green). For drawing a body, draw a circle with radius 5 with its center at

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

where x and y are coordinates 0 and 1 of the body (in case we use more than two dimensions, we just draw the first two). In addition, draw the name of the body near the circle. Note the use of _scale in order to scale the universe to fit in the component. For drawing use the canvas variable gr, e.g., methods gr.setColor, gr.fillOval, gr.drawString, and gr.drawLine.

In the observer methods, when the list of bodies changes (i.e., in onRegister, onBodyAdded, and onReset) update the value of _bodies and call autoScale() and repaint(). In method onAdvance just call repaint(), do not call autoScale().

6.4. Status Bar

The status bar displays some additional information related to the simulator's state: the current time, the total number of bodies, and the current force laws. It is represented by a class called StatusBar:

```
public class StatusBar extends JPanel implements SimulatorObserver {

   // ...
   private JLabel _currTime;      // for current time
   private JLabel _currLaws;      // for force 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
   // ...
}
```

Note that fields _currTime, _numberOfBodies and _currLaws are labels where the corresponding information is stored. In the observer methods, if any of these information is changed you should modify the corresponding JLabel.

6.5. Main Window

The main window is a class called MainWindow that extends JFrame:

```
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 complete this method to build the GUI
     // ...
   }

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

Complete method initGUI() to create the corresponding objects and build the GUI: (1) place the control panel at the PAGE_START of mainPanel panel; (2) place the status bar at the PAGE_END of mainPanel; (3) create a new panel with BoxLayout (with BoxLayout.Y_AXIS) and place it at the CENTER of mainPanel, then add the bodies table and the viewer to this new panel.

In order to control the initial size of the different components you can use method setPreferredSize. You will also need to make the window visible, etc.

7. Modifying the Main Class

In the main class you will need to add a new option -m to allow the user to use the simulator in BATCH mode (as in assignment 1) or GUI mode:

```
usage: simulator.launcher.Main [-cmp <arg>] [-dt <arg>] [-eo <arg>] [-fl
       <arg>] [-h] [-i <arg>] [-m <arg>] [-o <arg>] [-s <arg>]
 -cmp,--comparator <arg>       State comparator to be used when comparing
                               states. Possible values: 'epseq'
                               (Espsilon-equal states comparator),
                               'masseq' (Mass equal states comparator).
                               You can provide the 'data' json attaching
                               :{...} to the tag, but without spaces..
                               Default value: 'epseq'.
 -dt,--delta-time <arg>        A double representing actual time, in
                               seconds, per simulation step. Default
                               value: 2500.0.
 -eo,--expected-output <arg>   The expected output file. If not provided
                               no comparison is applied
 -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), 'ng' (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: '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.
```

In method start, depending on the value provided for option -m, call method startBatchMode or a new method startGUIMode which includes the code for the new GUI mode. Unlike BATCH mode, in GUI mode parameter -i is optional, and when provided the corresponding file should be loaded into the simulator as in assignment 1 — the graphical user interface will start with some content in such case. Options -o and -s should be ignored in GUI mode. Recall that in order to create the window you should use:

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