Assignment 2: ecosystem simulator GUI

Objectives: Object-oriented design, the Model-View-Controller pattern, graphical interfaces with Swing.

Submission deadline: April 22 2024, 08:30

Plagiarism

For each of the TP2 assignments, all the submissions from all the different TP2 groups will be checked using anti-plagiarism software, firstly, by comparing all of them pairwise and, secondly, by searching to see if any of the code of any of them is copied from other sources on the Internet (without explicit permission from the lecturer)[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 both TP-course exam sessions (*convocatorias*) for that academic year,
- The opening of disciplinary proceedings (*apertura de un expediente académico*) with the relevant university authority (*Inspección de Servicios*)

General instructions

The following instructions must be strictly followed:

1. Read all the assignment problem statement before starting.
2. Make a backup copy of your solution to the first assignment before making any modifications to it for the second assignment.
3. Create a new package `simulator.view` to hold all the view classes.
4. You must use exactly the same package structure and class names as in the problem statement.
5. The use of tools for automatic GUI generation is not permitted.
6. Download `extra.zip` and unzip it in the `src` folder (it includes examples of `JTable` and `JDialog`).
7. Download `ViewUtils.java`, `AbstractMapViwer.java` and `MapViewer.java` and copy them to the package `simulator.view`.
8. Download `icons.zip` and unzip it in the directory `resources` so that the icons will be in the directory `resources/icons`. It is not permitted to place the icons in another directory.
9. Do not display errors via `System.out` and do not use `printStackTrace()` for exceptions; all errors must be displayed using `ViewUtils.showErrorMsg`.
10. When submitting your solution to the assignment, you must upload only the **`src`** folder which must be compressed with zip in a file which must be called **`src.zip`**. Other types of compression (**`7zip`**, **`rar`**, etc.) are not allowed, nor is any other file name. If you use additional icons, your submission may include the directory `resources/icons` as long as the overall size of the zip is under **`100k`**.

General description of the simulator GUI

The second assignment is to develop a GUI for the ecosystem simulator developed in the previous assignment, according to the model-view-controller (`MVC`) pattern. The section of this document entitled Figures contains images of the `GUI` to be developed. It comprises a main window with four components: (1) a control panel for interacting with the simulator (2) a table showing information about the animal species and their state; (3) a table showing the state of each of the regions; and (4) a status bar containing more information which we explain below. It also includes a dialog for changing the properties of the regions, and a window (that opens separately) containing a graphical representation of the state of the simulation (similar to that of the viewer used in the previous assignment).

1 If you decide to store your code in a remote repository, e.g. github, with a view to facilitating collaboration with your lab partner, make sure your code is not in reach of search engines. Also, do not accept any code from, or provide any code to, anyone other than your lab partner or your course lecturer (e.g. do not accept any code from, or provide any code to, an employee of a private academy).

Changes to the Model and to the Controller

This section describes the changes to be made to the model and the controller in order to adapt them to the MVC pattern and to add some extra functionality.

The `reset` method of the Simulator class

Add the following method to the `Simulator` class (if you do not already have it):

1. `public void reset(int cols, int rows, int width, int height)`: empties the list of animals (or creates a new list), creates a new `RegionManager` of the right size and sets the time to `0.0` .

The `toString` method of the region classes

If you did not do it in the previous assignment, to all the classes that extend `Region`, add a `toString()` method that returns a brief description of the region, for example:

- `"Default region"`
- `"Dynamic region"`

This information is to be used by the GUI, in particular, in the state-of-the-regions table.

The `fill_in` method of the builders

Complete the `fill_in_data` method in all the builders so that it provides the appropriate information (at least for the region builders, we are not going to use the rest, for the moment). For example, `get_info()` of the region builders must return the following `JSON`:

```
{
   "type": "default",
   "desc": "Infinite food supply",
   "data": {}
}
```

```
{
   "type": "dynamic",
   "desc": "Dynamic food supply",
   "data": {
             "factor": "food increase factor (optional, default 2.0)",
             "food": "initial amount of food (optional, default 100.0)"
           }
}
```

You may wish to do the same for the animal bulders (even though it is not strictly necessary for this assignment).

An iterator over regions in the `RegionManager` class

Our objective is to give access to the regions from outside the model, without it being possible to alter their state via this access.

We start by modifying the interface `RegionInfo` to give access to `List<AnimalInfo>` instead of to `List<Animal>`:

```
public interface RegionInfo extends JSONable {
  public List<AnimalInfo> getAnimalsInfo();
```

```
}
```

In the `Region` class the corresponding method will be:

```
public List<AnimalInfo> getAnimalsInfo() {
  // can use Collections.unmodifiableList(_animals);
  // since Java 9, we can also use List.of() instead of unmodifiableList
  return new ArrayList<>(_animals);
}
```

Both of the above **`return`** implementation options are valid, /though the first (`unmodifiableList`/`List.of`) is thread-safe while the second is not (this is important for the third assignment). Important exercise (not for submission, merely for understanding): explain why "`return _animals`" does not compile while the two options given do compile.

We now modify the `RegionManager` class by equipping it with an iterator to traverse the regions (adding a `get_region(int row, int col)` method in order to consult the region in the position `(row,col)` from outside the class is forbidden, you must do it with an iterator in order to practice using iterators). We modify the `MapInfo` interface to include a record of the information about the regions and to implement the iterable interface:

```
public interface MapInfo extends JSONable, Iterable<MapInfo.RegionData> {
  public record RegionData(int row, int col, RegionInfo r) {
  }
 // the rest of the interface is as before
}
```

The `RegionData` record simply includes a region object and its position but the region object has type `RegionInfo` rather than `Region` to ensure that the state cannot be altered from outside the simulation.

We can now implement the corresponding iterator in the `RegionManager` class that traverses the region matrix (row by row, from left to right) and for each region returns a corresponding instance of `RegionData.`

The `EcoSysObserver` interface

The observers implement the following interface that includes various types of notifications (place it in the `simulator.model` package):

```
public interface EcoSysObserver {
  void onRegister(double time, MapInfo map, List<AnimalInfo> animals);
  void onReset(double time, MapInfo map, List<AnimalInfo> animals);
  void onAnimalAdded(double time, MapInfo map, List<AnimalInfo> animals,
                     AnimalInfo a);
  void onRegionSet(int row, int col, MapInfo map, RegionInfo r);
  void onAvanced(double time, MapInfo map, List<AnimalInfo> animals, double dt);
}
```

The names of the methods give information about the meaning of the events that they notify. Regarding the parameters: `map` is the region manager; `animals` is the list of animals; `a` is an animal, `r` is a region, `time` is the current simulation time and `dt` is the time-delta used in the simulation step. Note that we use the types `MapInfo`, `AnimalInfo` and `RegionInfo` for these objects instead of `Animal`, `RegionManager` and `Region` to ensure that the state of their state cannot be altered from outside the simulation.

Modify the `Simulator` class so that it implements `Observable<EcoSysObserver>` dwhere the interface `Observable<T>` is defined as follows:

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

Add an initially-empty list of observers to the `Simulator` class and add the following methods to register / remove observers:

1. `public void addObserver(EcoSysObserver o)`: add the observer `o` to the list of observers, if it is not already present.
2. `public void removeObserver(EcoSysObserver o)`: remove the observer `o` from the list of observers.

Sending notifications

Modify the `Simulator` class to send notifications as described below:

1. At the end of the body of the `addObserver` method, send a notification `onRegister` **(only) to the observer that has just registered**, in order to pass the current state of the simulator.
2. At the end of the body of the `reset` method, send a notification `onReset` **to all the observers.**
3. Al the end of the `add_animal` method send a notification `onAnimalAdded` **to all the observers.**
4. At the end of the `set_region` method, send a notification `onRegionSet` **to all the observers.**
5. At the end of the `advance` method, send a notification `onAdvance` **to all the observers.**

Note that the list of animals must be passed to the observers with type List<AnimalInfo>; this can be done using `"new ArrayList<>(_animals)"` or `"Collections.unmodifiableList(_animals)"`, the difference between the two forms having been explained on P.4. For example, the advance notification can be done using the following method:

```
private void notify_on_advanced(double dt) {
List<AnimalInfo> animals = new ArrayList<>(_animals);
// for each observer, or invoke o.onAvanced(_time, _region_mngr, animals, dt)
}
```

Changes to the `Controller` class

The `Controller` class must be extended with the following additional functionality (in order to avoid the GUI needing a reference to the simulator):

1. `public void reset(int cols, int rows, int width, int height)`: calls the `reset` method of the simulator.
2. `public void set_regions(JSONObject rs)`: supposes that `rs` is a `JSON` structure that has a key `"regions"` (as in the first assignment) and modifies the corresponding regions using the `set_regions` method of the simulator. The code of the `load_data` method will have to be refactored in order to avoid duplication of code (since this method already did something similar).
3. `public void advance(double dt)`: llama a `advace` del simulador.
4. `public void addObserver(EcoSysObserver o)`: calls the `addObserver` method of the simulator.
5. `public void removeObserver(EcoSysObserver o)`: calls the `removeObserver` method of the simulator.

Attributes of the `Main` class storing the factories and the delta-time

In the `Main` class, the attributes storing a reference to the factories y the delta-time must be made public since they are to be used from the GUI (Simon: you could instead use *getters*).

The GUI

In this section we describe the different classes of our `GUI`.

The main window

The main window is represented by the following class where you must complete the parts that are not implemented. `GridBagLayout` or `GridLayout` can be used in place of `BoxLayout`.

```
  public class MainWindow extends JFrame {

    private Controller _ctrl;

    public MainWindow(Controller ctrl) {
      super("[ECOSYSTEM SIMULATOR]");
      _ctrl = ctrl;
      initGUI();
    }

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

   // TODO create ControlPanel and add it in the PAGE_START section of mainPanel

   // TODO create StatusBar and add it in the PAGE_END section of mainPanel

   // Definition of the tables panel (use a vertical BoxLayout)
      JPanel contentPanel = new JPanel();
      contentPanel.setLayout(new BoxLayout(contentPanel, BoxLayout.Y_AXIS));
      mainPanel.add(contentPanel, BorderLayout.CENTER);

   // TODO create the species table and add it to the contentPanel.
   //      Use setPreferredSize(new Dimension(500, 250)) to fix its size

   // TODO create the regions table.
   //      Use setPreferredSize(new Dimension(500, 250)) to fix its size

   // TODO call ViewUtils.quit(MainWindow.this) in the windowClosing method

      addWindowListener( … );

      setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
      pack();
      setVisible(true);
    }
  }
```

The control panel

The control panel is responsible for the interaction between the user and the simulator. Visually, it corresponds to the toolbar appearing across the top of the window (see the Figures section). It includes the following components: buttons to interact with the simulator, a `JSpinner` to select the desired number of simulation steps and a `JTextField` to update the delta-time. The initial value that must appear in the delta-time is the value of the corresponding attribute in the `Main` class.

```
class ControlPanel extends JPanel {

  private Controller _ctrl;
  private ChangeRegionsDialog _changeRegionsDialog;

  private JToolBar _toolBar;
  private JFileChooser _fc;
  private boolean _stopped = true; // used in the run/stop buttons
  private JButton _quitButton;

   // TODO add more attributes here …

  ControlPanel(Controller ctrl) {
      _ctrl = ctrl;
      initGUI();
  }

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

  // TODO create the different buttons/attributes and add them to the toolbar.
  //      Each of them should have a corresponding tooltip. You may use
  //      _toolBar.addSeparator() to add the vertical-line separator between
  //      those components that need it.

  // 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) -> ViewUtils.quit(this));
      _toolBar.add(_quitButton);

        // TODO Initialise _fc with a JfileChooser instance. In order for it
        // to open in the examples directory, you can use the following code:
        //
        //   _fc.setCurrentDirectory(new File(System.getProperty("user.dir")
        //                                    + "/resources/examples"));

        // TODO Initialise _changeRegionsDialog with an instance of the
        // change-regions dialog.

  }
   // TODO The rest of the methods go here...
}
```

The functionality of the different buttons is the following:

- When the  button is pressed: (1) use `_fc.showOpenDialog(ViewUtils.getWindow(this))` to open the file chooser for the user to select an input file; (2) when the user has selected a file, load it and parse the file contents into a `JSONObject`, reset the simulator using `_ctrl.reset(...)` with the corresponding parameters, and load the `JSONObject` created using `_ctrl.load_data(...)`.
- When the  button is pressed, create an instance of the `MapWindow` class (described below). This presents the user with a graphical representation of the simulation. Take into account that the user may

have multiple viewers open at the same time.

- If the  button is pressed, call `_changeRegionsDialog.open(ViewUtils.getWindow(this))` to open the regions dialog (recall that only one instance is created in the constructor).
- When the  button is pressed: (1) disable all the buttons except the stop button () and change the value of the attribute `_stopped` to `false`; (2) get the delta-time value from the corresponding `JTextField`; and (3) call the `run_sim` method with the value of steps specified in the corresponding `JSpinner`:

```
private void run_sim(int n, double dt) {
   if (n > 0 && !_stopped) {
      try {
            _ctrl.advance(dt);
            SwingUtilities.invokeLater(() -> run_sim(n - 1, dt));
      } catch (Exception e) {
            // TODO pass the corresponding error message to
            //      ViewUtils.showErrorMsg
            // TODO enable all the buttons
            _stopped = true;
      }
   } else {
            // TODO enable all the buttons
      _stopped = true;
   }
}
```

  Complete the `run_sim` method as indicated in the comments. Notice how the code of the run-sim method given here guarantees that the user interface will not deadlock. To understand this behaviour, try removing the *invokeLater* instruction. You will observe that, on starting the simulation, it jumps straight to the final state with no intermediate steps and the user interface will be completely blocked.
- When the  button is pressed, update the value of the attribute `_stopped` to `true`. This will stop the `run_sim` method if there are calls in the swing event queue (see the loop condition in the `run_sim` method).
- The functionality of the  button is as in the code provided.

You must catch all the possible exceptions thrown by the controller and simulator and display a corresponding message using `ViewUtils.showErrorMsg`. Do not write error messages to the standard output or standard error (`System.out` / `System.err`, nor by calling the `printStackTrace()` method of an exception.

The status bar

The status bar shows general information about the simulation. It is displayed along the bottom of the main window (see the Figures section). Read and then complete the skeleton of the code provided here. It is obligatory to include the simulation time, the number of animals and the dimensions used in the simulation (width, height, rows and cols). You may add further information if you wish to do so. Update the different values of the status bar from methods of the `EcoSysObserver` class whenever necessary.

```
class StatusBar extends JPanel implements EcoSysObserver {

    // TODO Add any necessary attributes.

    StatusBar(Controller ctrl) {
    initGUI();
    // TODO register 'this' as an observer
}
```

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

        // TODO Create several Jlabels for the time, the number of animals and
        // the dimensions, and add them to the panel. You may use the following
        // code to add a new vertical separator.
        //
        //     JSeparator s = new Jseparator(JSeparator.VERTICAL);
        //     s.setPreferredSize(new Dimension(10, 20));
        //     this.add(s);
    }

        // TODO all the other methods go here.
}
```

The information tables

The tables show information about the animals and the regions. We have an `InfoTable` class, which includes a `Jtable`, that receives the corresponding table model as a parameter to the constructor, and classes `SpeciesTableModel` y `RegionsTableModel`, for the models of the species and the regions.

Since the tables have parts in common, we define a class to represent a table that receives the table model (and the data it contains) as input and uses it for both tables:

```
public class InfoTable extends JPanel {

    String _title;
    TableModel _tableModel;

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

    private void initGUI() {
        // TODO change the panel layout to BorderLayout()
        // TODO add a border with a title (using text _title) to the Jpanel
        // TODO add a JTable (with vertical scroll bar) that uses _tableModel
    }
}
```

Using `InfoTable`, the tables in the `MainWindow` scan be created as follows:

```
new InfoTable("Species", new SpeciesTableModel(_ctrl));
new InfoTable("Regions", new RegionsTableModel(_ctrl));
```

where `SpeciesTableModel` and `RegionsTableModel` are as described below.

The species tables

The first table model represents the species table and is encapsulated in the following class:

```
class SpeciesTableModel extends AbstractTableModel implements EcoSysObserver {
```

```
    // TODO define the necessary attributes

    SpeciesTableModel(Controller ctrl) {
        // TODO initialise the corresponding data structures
        // TODO register 'this' as an observer
    }
    // TODO the rest of the methods go here...
}
```

The table contains a row for each genetic code with information about the number of animals in each possible state (see the Figures section).

IMPORTANT: If we add more genetic codes and/or states to the simulator, the table should display this extra information correctly *without any modification of its code*, for which reason (1) it is forbidden to make any explicit reference to a genetic code such "`sheep`" y "`wolf`"; this information must be obtained from the list of animals; (2) it is forbidden to make any reference to concrete states such as `NORMAL`, `DEAD`, etc.; the possible states must be obtained by using `State.values()`.

The regions table

El second table model represents la regions table and is encapsulated in the following class:

```
class RegionsTableModel extends AbstractTableModel implements EcoSysObserver {

    // TODO define the necessary attributes

    RegionsTableModel(Controller ctrl) {
    // TODO initialise the corresponding data structures
    // TODO register 'this' as an observer

    }
    // TODO the rest of the methods go here...
}
```

The table contain a fow for each region with information about its row and column number in the region matrix, its description (returned by the `toString()` method of the region) and the number of animlas in the region for each type of diet (see the Figures section).

IMPORTANTE: Si añadimos más tipos de dietas al simulador, la tabla tiene que seguir funcionando igual sin la necesidad de modificar nada de su código, y por eso está prohibido hacer referencia explícita a tipos de dietas como **`CARNIVORE`** y **`HERBIVORE`**. Hay que usar `Diet.values()` para saber cuales son las posibles dietas.

The region-change dialog

La clase `ChangeRegionsDialog` es la responsable de implementar la ventana de diálogo que permite modificar las regiones (Ver el apartado Figuras):

```
class ChangeRegionsDialog extends JDialog implements EcoSysObserver {

  private DefaultComboBoxModel<String> _regionsModel;
  private DefaultComboBoxModel<String> _fromRowModel;
  private DefaultComboBoxModel<String> _toRowModel;
  private DefaultComboBoxModel<String> _fromColModel;
  private DefaultComboBoxModel<String> _toColModel;
```

```
 private DefaultTableModel _dataTableModel;
 private Controller _ctrl;
 private List<JSONObject> _regionsInfo;

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

// TODO en caso de ser necesario, añadir los atributos aquí…
 ChangeRegionsDialog(Controller ctrl) {
 super((Frame)null, true);
 _ctrl = ctrl;
 initGUI();
 // TODO registrar this como observer;
 }

 private void initGUI() {
 setTitle("Change Regions");
 JPanel mainPanel = new JPanel();
 mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
 setContentPane(mainPanel);

 // TODO crea varios paneles para organizar los componentes visuales en el
      // dialogo, y añadelos al mainpanel. P.ej., uno para el texto de ayuda,
      // uno para la tabla, uno para los combobox, y uno para los botones.

 // TODO crear el texto de ayuda que aparece en la parte superior del diálogo
y
      // añadirlo al panel correspondiente diálogo (Ver el apartado Figuras)

 // _regionsInfo se usará para establecer la información en la tabla
 _regionsInfo = Main._regions_factory.get_info();

 // _dataTableModel es un modelo de tabla que incluye todos los parámetros de
      // la region
 _dataTableModel = new DefaultTableModel() {
 @Override
 public boolean isCellEditable(int row, int column) {
 // TODO hacer editable solo la columna 1
 }
 };
 _dataTableModel.setColumnIdentifiers(_headers);

 // TODO crear un JTable que use _dataTableModel, y añadirlo al diálogo

 // _regionsModel es un modelo de combobox que incluye los tipos de regiones
 _regionsModel = new DefaultComboBoxModel<>();

 // TODO añadir la descripción de todas las regiones a _regionsModel, para eso
      //      usa la clave “desc” o “type” de los JSONObject en
_regionsInfo,
      //      ya que estos nos dan información sobre lo que puede crear la
factoría.
```

```
  // TODO crear un combobox que use _regionsModel y añadirlo al diálogo.

  // TODO crear 4 modelos de combobox para _fromRowModel, _toRowModel,
        //       _fromColModel y _toColModel.

  // TODO crear 4 combobox que usen estos modelos y añadirlos al diálogo.

  // TODO crear los botones OK y Cancel y añadirlos al diálogo.

  setPreferredSize(new Dimension(700, 400)); // puedes usar otro tamaño
  pack();
  setResizable(false);
  setVisible(false);
  }

  public void open(Frame parent) {
       setLocation(//
      parent.getLocation().x + parent.getWidth() / 2 - getWidth() / 2, //
      parent.getLocation().y + parent.getHeight() / 2 - getHeight() / 2);
        pack();
        setVisible(true);
  }

  // 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 en la constructora, 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 `onReset` y `onRegister` de `EcoSysObserver` debes mantener la lista de opciones en los combobox de coordenadas actualizada – usa `removeAllElements` y `addElement` del modelo correspondiente. Así cuando cambia el número de fila/columnas cambian también en los combobox.
2. Cuando el usuario selecciona la i-ésima región (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 `_regionsInfo`, 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.
3. Si el usuario pulsa el botón `Cancel`, simplemente pon el `_status` a `0` y haz el diálogo invisible.
4. Si el usuario pulsa el botón `OK`
   a. Convierte la información en la tabla en un `JSON` que incluye la clave y el valor para cada fila en la tabla, sólo para la fila que incluyen valor no vacío – en el ejemplo `extra.dialog.ex3` hay un método que hace algo parecido. Nos referimos a este `JSON` como `region_data`.
   b. Sacar el tipo de la región seleccionado usando (usando el índice seleccionado puedes hacerlo desde `_regionsInfo`). Nos referimos a este valor como `region_type`.
   c. Sacar las coordenadas de los combobox correspondientes. Nos referimos a estos valores como `row_from`, `row_to`, `col_from`, `col_to`.
   d. Crear un `JSON` de la forma:

```
{
  "regions" : [ {
    "row" : [ row_from, row_to ],
```

```
        "col" : [ col_from, col_to ],
        "spec" : {
          "type" : region_type,
          "data" : region_data
        }
      ]
    }
```

y pasalo a `_ctrl.set_regions` para cambiar las regiones. Si la llamada acaba con éxito, pon `_status` a `1` y haz el diálogo invisible, en otro caso muestra el mensaje de la excepción correspondiente usando `ViewUtils.showErrorMsg`. No escribas errores con `System.out` o System.err, ni con `stackTrace()` de la excepción.

IMPORTANTE: Si añadimos más tipos de regiones a la factoría de regiones, el diálogo tiene que seguir funcionando igual sin la necesidad de modificar nada de su código, y por eso está prohibido hacer referencia explícita a tipos de regiones como "`default`" y "`dynamic`", ni a claves como "`factor`" y "`food`". Siempre hay que sacar la información usando `get_info()` de la factoría.

The map viewer

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

```
class MapWindow extends JFrame implements EcoSysObserver {

  private Controller _ctrl;
  private AbstractMapViewer _viewer;
  private Frame _parent;

  MapWindow(Frame parent, Controller ctrl) {
  super("[MAP 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

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

  // TODO en el método windowClosing, eliminar 'MapWindow.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);
  setResizable(false);
  setVisible(true);
  }
   // TODO otros métodos van aquí….
}
```

Notase que la ventana no se puede redimensionar para que el código que dibuje el estado en `_viewer` sea más sencillo.

Deberías completar el código de los métodos de `EcoSysObserver` de de forma que:

1. Los métodos `onRegister` y `onReset` llamen al `reset` del `_viewer` y cambien el tamaño de la ventana usando `pack()` porque el `_viewer` puede cambiar de tamaño. Esto se puede hacer usando `SwingUtilities.invokeLater(() -> { _viewer.reset(...); pack(); });`
2. El método `onAdvance` llame a update del `_viewer`. Esto se puede hacer usando `SwingUtilities.invokeLater(() -> { _viewer.update(...) });`

La clase `MapViewer.java` y `AbstractMapViwer.java` son dadas sin parte de su funcionalidad, lee todos los comentarios `TODO` dentro del código y completarlos – más información será explicada en las clases/laboratorios. En general el visor tiene que: (1) dibujar cada animal con un tamaño relativo a su edad y de color que corresponde a su código genético; (2) muestrar información sobre el tiempo actual y el número de animales de cada código genético; y (3) permetir mostrar solo animales que tienen estado especifico (pulsando la tecla `s` cambiamos de un estado a otro).

IMPORTANTE: Si añadimos más códigos genéticos y/o estados al simulador, la tabla tiene que seguir funcionando igual sin la necesidad de modificar nada de su código, y por eso (1) está prohibido hacer referencia explícita a códigos genéticos como “`sheep`” y “`wolf`”, esta información hay que sacarla de la lista de animales; (2) está prohibido hacer referencia a estados concretos como `NORMAL`, `DEAD`, etc. Hay que usar `State.values()` para saber cuales son los posibles estados.

Changes in the Main class

New option: --mode

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>] [-h] [-i <arg>] [-m <arg>] [-o
       <arg>] [-sv] [-t <arg>]

 -dt,--delta-time <arg>   A real number representing actual time, in
                          seconds, per simulation step. Default value:
                          0.03.
 -h,--help                Print this message.
 -i,--input <arg>         A configuration file (optional in GUI mode).
 -m,--mode <arg>          Execution Mode. Possible values: 'batch' (Batch
                          mode), 'gui' (Graphical User Interface mode).
                          Default value: 'gui'.
 -o,--output <arg>        A file where output is written (only for BATCH mode).
 -sv,--simple-viewer      Show the viewer window in BATCH mode.
```

```
  -t,--time <arg>          An real number representing the total simulation
                           time in seconds. Default value: 10.0. (only for BATCH
mode).
```

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. Las opciones `-o` y `-t` se ignoran en el modo `GUI`. Recuerda que las opciones -i y -o tienen que seguir siendo obligatorias en el modo `BATCH`.

The start_batch_mode method

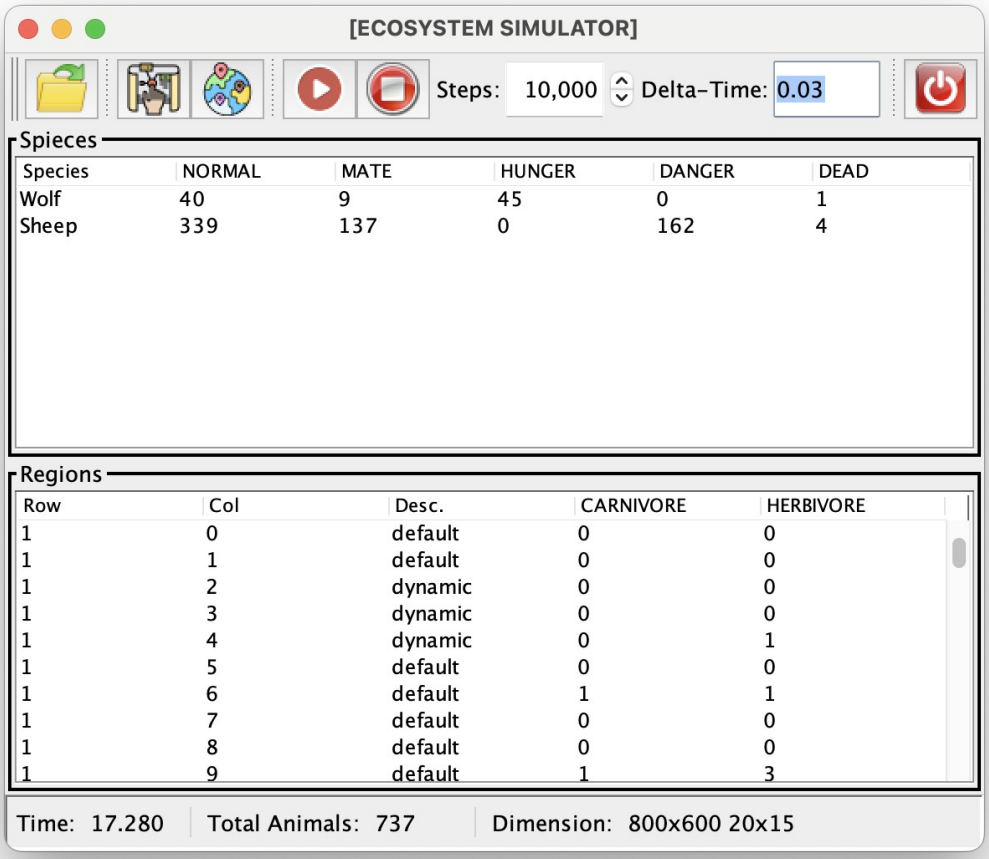
Completa el método `start_GUI_mode` de manera parecida a start_BATCH_mode, pero sin llamar al método `run` del controlador sino crear una ventana usando:

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

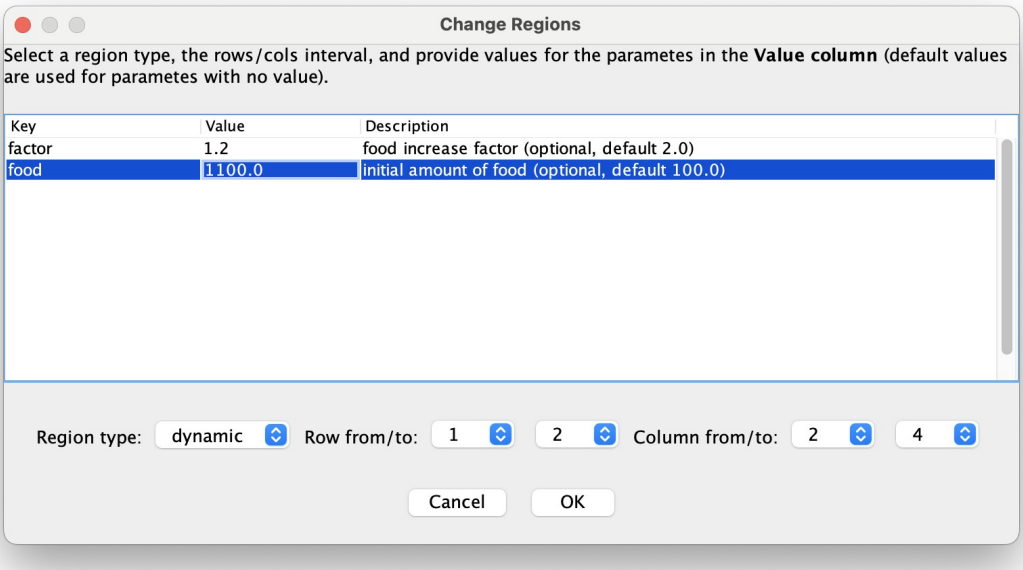
Recuerda que si el usuario ha proporciona un archivo de entrada, hay que usarlo para crear la instancia de `Simulator` y además añadir los animales y regiones usando `load_data` del controlador, y si no lo proporciona crea la instancia de `Simulator` con valores por defecto para la anchura, altura, filas y columnas (se puede usar `800`, `600`, `15`, `20`). Recuerda que no hay que usar archivo de salida en este modo.

Figures

The main window



The region-change dialog



The map viewer