

# Práctica 2: Interfaz gráfica para el simulador de ecosistema

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

**Fecha de entrega:** 22 de Abril 2024, 08:30h

- Control de Copias..... 2**
- Instrucciones Generales.....2**
- Descripción General de la Interfaz Gráfica del Simulador..... 2**
- Cambios en el Modelo y el Controlador.....2**
  - Método reset en el la clase Simulator..... 3
  - Método toString() en las clases de regiones..... 3
  - El método fill\_in\_data de los builders..... 3
  - Iterador de regiones en la clase RegionManager.....3
  - La interfaz EcoSysObserver.....4
  - Envío de notificaciones.....5
  - Cambios en la clase Controller.....5
  - Factorías y delta-time públicos en la clase Main.....5
- La interfaz gráfica de usuario..... 5**
  - Ventana principal.....5
  - Barra de control.....6
  - Barra de estado.....8
  - Tablas de Información..... 9
    - Tabla de especies.....9
    - Tabla de regiones..... 10
  - Diálogo de cambio de regiones..... 10
  - Visor del mapa.....13
- Cambios en la clase Main.....14**
  - Nueva opción --mode..... 14
  - Método start\_batch\_mode.....15
- Figuras..... 16**
  - Ventana principal.....16
  - Diálogo de cambio de regiones..... 16
  - Visor del mapa.....17

## 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.
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. Descarga `extra.zip` y descomprimelo en la carpeta `src` (incluye ejemplos de `JTable` y `JDialog`).
7. Descarga `ViewUtils.java`, `AbstractMapViewier.java` y `MapViewier.java` y copialos al paquete `simulator.view`.
8. Descarga `icons.zip` y descomprimelo en la carpeta `resources` tal que tendrás que tener una carpeta `resources/icons` dónde están los iconos. No está permitido usar otra carpeta para los iconos.
9. No escribas errores con `System.out` ni con `printStackTrace()` de la excepción, todos los errores hay que mostrarlos usando `ViewUtils.showErrorMsg`.
10. Cuando entregues la práctica sube un fichero con el nombre **src.zip** que incluya solo la carpeta **src**. No está permitido llamarlo con otro nombre ni usar **7zip**, **rar**, etc. Si usas iconos adicionales, se puede incluir la carpeta `resources/icons` mientras que el tamaño total del **zip** no supera los **100k**.

## Descripción General de la Interfaz Gráfica del Simulador

En esta práctica vas a desarrollar una interfaz gráfica de usuario (GUI) para el simulador de ecosistema siguiendo el patrón de diseño modelo-vista-controlador (MVC). En el apartado [Figuras](#) 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 información sobre las especies y su estado; (3) una tabla que muestra el estado de todas las regiones; y (4) una barra de estado en la que aparece más información, que detallaremos después. Además, incluye un diálogo que permite cambiar las regiones, y una ventana (que se abre de forma separada) para dibujar el estado de la simulación (similar al visor 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.

## Método reset en el la clase Simulator

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

1. `public void reset(int cols, int rows, int width, int height)`: vacía la lista de animales (o crea una nueva), crea un nuevo RegionManager con tamaño adecuado, y pone el tiempo a 0.0 .

## Método toString() en las clases de regiones

Además, si no lo hiciste en la práctica 1, añade un método toString() a todas las clases que extienden la clase Region que devuelva una pequeña descripción correspondiente, por ejemplo:

- "Default region"
- "Dynamic region"

Esta información se usará en la GUI para mostrar la descripción de una región.

## El método fill\_in\_data de los builders

Completa el método fill\_in\_data en todos los builders para que rellene información correspondiente (por lo menos hacerlo para los builders de regiones, el resto no lo vamos a usar de momento). Por ejemplo, get\_info() de los builders de regiones tendrá que devolver los siguientes 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)"
  }
}
```

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

## Iterador de regiones en la clase RegionManager

Nuestro objetivo es dar acceso a las regiones desde fuera del modelo, de tal manera que no se pueda alterar sus estados.

Empezamos con la modificación de la interfaz RegionInfo para permitir el acceso a la lista de animales como List<AnimalInfo> en lugar de List<Animal>:

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

En la clase Region el método correspondiente será:

```
public List<AnimalInfo> getAnimalsInfo() {
```

```
return new ArrayList<>(_animals); // se puede usar Collections.unmodifiableList(_animals);
}
```

Las dos opciones en la línea del `return` son válidas, la primera es segura para programación concurrente mientras la segunda no (esto es importante para la tercera práctica). **Ejercicio muy importante (no para entregar, simplemente para entender): explicar por qué "`return _animals`" no compila, mientras las opciones de arriba sí compilan.**

Ahora vamos a modificar la clase `RegionManager` para que tenga un iterador que permite recorrer sobre las regiones (está prohibido añadir un método `get_region(int row, int col)` para consultar la región en la posición `(row,col)` desde fuera, es obligatorio hacerlo con un iterador para practicar los iteradores). Empezamos con la modificación de la interface `MapInfo` para incluir un record de información sobre las regiones e implementar la interfaz `Iterable`:

```
public interface MapInfo extends JSONable, Iterable<MapInfo.RegionData> {
    public record RegionData(int row, int col, RegionInfo r) {
    }
    // el resto de la interfaz es como antes.
}
```

El registro `RegionData` simplemente incluye la posición de la región y la región pero como `RegionInfo` en lugar de `Region` para asegurarnos que no se altera su estado desde fuera.

Ahora implementa un iterador correspondiente en la clase `RegionManager` tal que recorre la matriz de regiones (por filas, de izquierda a derecha) y para cada región devuelve una instancia correspondiente de `RegionData`.

## La interfaz `EcoSysObserver`

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

```
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 onAdvanced(double time, MapInfo map, List<AnimalInfo> animals, double dt);
}
```

Los nombres de los métodos dan información sobre el significado de los eventos que notifican. En cuanto a los parámetros: `map` es el gestor de regiones; `animals` es la lista de animales; `a` es un animal, `r` es una región, `time` es el tiempo actual de la simulación y `dt` es el delta-time usando en el paso de simulación correspondiente. Notase que usamos los tipos `MapInfo`, `AnimalInfo` y `RegionInfo` en lugar de `Animal`, `RegionManager` y `Region`, para no permitir alterar el estado de los objetos correspondientes desde fuera de la simulación.

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

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

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

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

## Envío de notificaciones

Modifica la clase Simulator 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 `add_animal` envía una notificación `onAnimalAdded` a **todos los observadores**.
4. Al final del método `set_region` envía una notificación `onRegionSet` a **todos los observadores**.
5. Al final del método `advance` envía una notificación `onAdvance` a **todos los observadores**.

Notase que la lista de animales hay que pasarla como `List<AnimalInfo>` a los observadores, esto se puede hacer usando `"new ArrayList<>(_animals)"` or `"Collections.unmodifiableList(_animals)"`, la diferencia entre las dos formas es como hemos explicado anteriormente. Por ejemplo, la notificación de advance se puede hacer usando el siguiente método:

```
private void notify_on_advanced(double dt) {
    List<AnimalInfo> animals = new ArrayList<>(_animals);
    // para cada observador o, invocar o.onAdvanced(_time, _region_mgr, animals, dt)
}
```

## Cambios en la clase 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(int cols, int rows, int width, int height)`: llama a `reset` del simulador.
2. `public void set_regions(JSONObject rs)`: suponiendo que `rs` es una estructura JSON que incluye la clave "regions" (como en la primera práctica), modifica las regiones correspondientes usando `set_regions` del simulador. Hay que hacer refactorización del código del `load_data` para que no haya duplicación de código (porque `load_data` ya hacía algo parecido).
3. `public void advance(double dt)`: llama a `advance` del simulador.
4. `public void addObserver(EcoSysObserver o)`: llama a `addObserver` del simulador.
5. `public void removeObserver(EcoSysObserver o)`: llama a `removeObserver` del simulador.

## Factorías y delta-time públicos en la clase Main

En la clase Main, hacer los atributos que corresponden a las factorías y el delta-time públicos porque se van a usar desde la GUI.

## La interfaz gráfica de usuario

En esta sección describiremos las distintas clases de nuestra GUI.

### Ventana principal

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

están implementadas. En lugar de BorderLayout se puede usar GridBagLayout o GridLayout.

```
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 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 BorderLayout vertical)
        JPanel contentPanel = new JPanel();
        contentPanel.setLayout(new BorderLayout(contentPanel, BorderLayout.Y_AXIS));
        mainPanel.add(contentPanel, BorderLayout.CENTER);

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

        // TODO crear la tabla de regiones.
        //      Usa setPreferredSize(new Dimension(500, 250)) para fijar su tamaño

        // TODO llama a ViewUtils.quit(MainWindow.this) en el método windowClosing
        addWindowListener( ... );

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

## Barra de control

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 (Ver el apartado [Figuras](#)). 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. El valor inicial que tiene que aparecer en el delta-time es el del atributo correspondiente en la clase Main.

```
class ControlPanel extends JPanel {

    private Controller _ctrl;
    private ChangeRegionsDialog _changeRegionsDialog;

    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();
}

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 Inicializar _fc con una instancia de JFileChooser. Para que siempre
    // abre en la carpeta de ejemplos puedes usar:
    //
    // _fc.setCurrentDirectory(new File(System.getProperty("user.dir") + "/resources/examples"));

    // TODO Inicializar _changeRegionsDialog con instancias del diálogo de cambio
    // de regiones
}
// 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(ViewUtils.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, cárgalo como `JSONObject`, resetea el simulador utilizando `_ctrl.reset(...)` con parámetros correspondiente, y cárgalo usando `_ctrl.load_data(...)`.
- Cuando se pulsa el botón  crea una instancia de `MapWindow` (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  llama a `_changeRegionsDialog.open(ViewUtils.getWindow(this))` para abrir el diálogo de regiones (recuerda que la instancia se crea sólo una vez en la constructora).
- 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) saca el valor del delta-time del correspondiente `UITextField`; y (3) llama al método `run_sim` con el valor actual de pasos, especificado en el correspondiente `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 llamar a ViewUtils.showErrorMsg con el mensaje de error
            // que corresponda
            // TODO activar todos los botones
            _stopped = true;
        }
    } 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 `run_sim` para incluir solo `for(int i=0;i<n;i++) _ctrl.advance(dt)` — 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 `ViewUtils.showErrorMsg`. No escribas errores con `System.out` o `System.err`, ni con `stackTrace()` de la excepción.

## Barra de estado

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 (Ver el apartado [Figuras](#)). Lee el código y completa las partes que faltan. Es obligatorio añadir el tiempo de simulación, el número total de animales, y la dimensión de la simulación (anchura, altura, filas, y columnas). Puedes añadir más información si lo deseas. Actualizar los distintos valores desde los métodos de `EcoSysObserver` cuando sea necesario,

```
class StatusBar extends JPanel implements EcoSysObserver {

    // TODO Añadir los atributos necesarios.

    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 varios JLabel para el tiempo, el número de animales, y la
        //      dimensión y añadirlos al panel. Puedes 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...
}

```

## Tablas de Información

Las tablas son las responsables de mostrar la información de los animales/regiones. Tendremos una clase `InfoTable` que incluya un `JTable`, que recibirá como parámetro el correspondiente modelo de la tabla, y dos clases `SpeciesTableModel` y `RegionsTableModel` para los modelos de las especies y las regiones.

Como las tablas tienen partes comunes, vamos a definir una clase que representa una tabla que recibe el modelo de tabla (que incluye los datos) como parámetro y usarla para ambas tablas:

```

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
    }
}

```

Usando `InfoTable`, la creación de las tablas en `MainWindow` se puede implementar así:

```

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

```

donde `SpeciesTableModel` y `RegionsTableModel` están descritas a continuación.

## Tabla de especies

El primer modelo de tabla representa la tabla de especies y será representado por la siguiente clase:

```

class SpeciesTableModel extends AbstractTableModel implements EcoSysObserver {

```

```
// TODO definir atributos necesarios

SpeciesTableModel(Controller ctrl) {
    // TODO inicializar estructuras de datos correspondientes
    // TODO registrar this como observador
}
// TODO el resto de métodos van aquí ...
}
```

La tabla incluye una fila para cada código genético con información sobre el número de animales en cada posible estado (Ver el apartado [Figuras](#)).

**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 cuáles son los posibles estados.

## Tabla de regiones

El segundo modelo de tabla representa la tabla de regiones y será representado por la siguiente clase:

```
class RegionsTableModel extends AbstractTableModel implements EcoSysObserver {

    // TODO definir atributos necesarios

    RegionsTableModel(Controller ctrl) {
        // TODO inicializar estructuras de datos correspondientes
        // TODO registrar this como observador
    }
    // TODO el resto de métodos van aquí...
}
```

La tabla incluye una fila para cada región con información sobre su fila y columna en la matriz de regiones, su descripción (lo que devuelve `toString()` de la región), y el número de animales en la región para cada tipo de dieta (Ver el apartado [Figuras](#)).

**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.

## Diálogo de cambio de regiones

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.

## Visor del mapa

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 `MapView` que hace la visualización (extiende una clase abstracta llamada `AbstractMapView` de tal forma que nos abstraemos de la implementación actual; notar que `AbstractMapView` 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 AbstractMapView _viewer;
    private Frame _parent;

    ViewerWindow(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 `MapView.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) mostrar información sobre el tiempo actual y el número de animales de cada código genético; y (3) permitir mostrar solo animales que tienen estado específico (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.

## Cambios en la clase Main

### Nueva opción `--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.
```

<code>-h,--help</code>	Print this message.
<code>-i,--input &lt;arg&gt;</code>	A configuration file (optional in GUI mode).
<code>-m,--mode &lt;arg&gt;</code>	Execution Mode. Possible values: 'batch' (Batch mode), 'gui' (Graphical User Interface mode). Default value: 'gui'.
<code>-o,--output &lt;arg&gt;</code>	A file where output is written (only for BATCH mode).
<code>-sv,--simple-viewer</code>	Show the viewer window in BATCH mode.
<code>-t,--time &lt;arg&gt;</code>	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.

### Método `start_batch_mode`

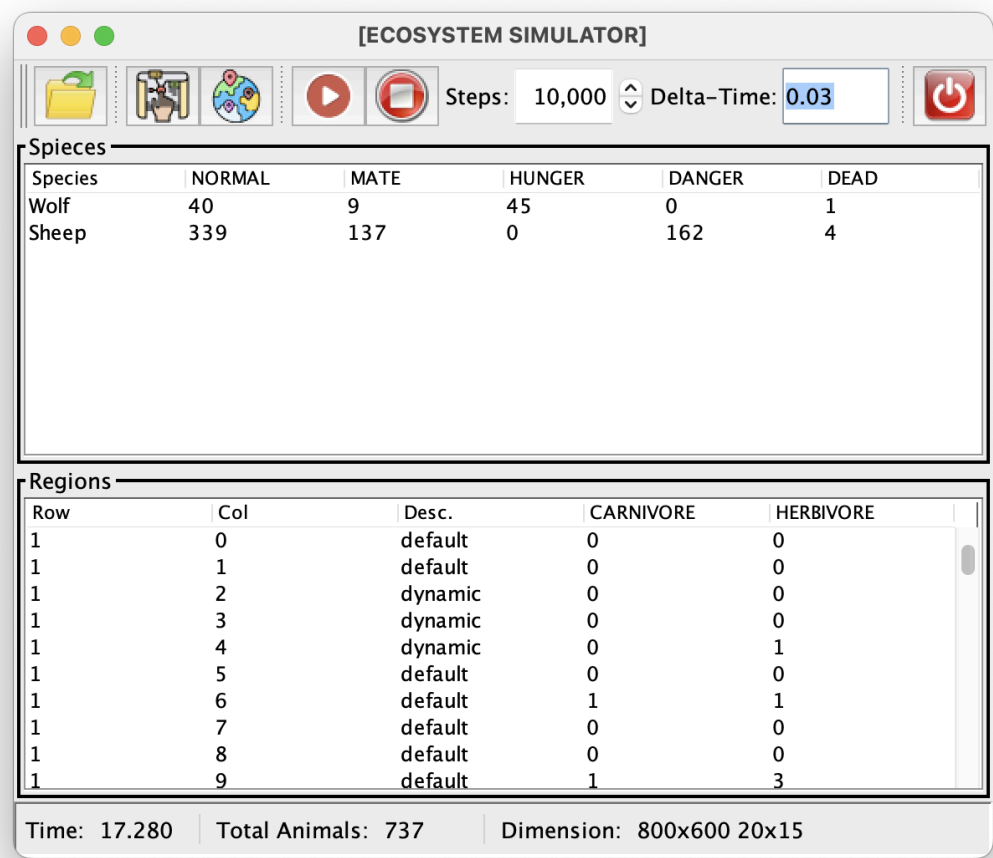
Completa el método `start_GUI_mode` de manera parecida a `start_BATCH_mode`, pero sin llamar al método `run` del controlador sino crear una ventana usando:

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

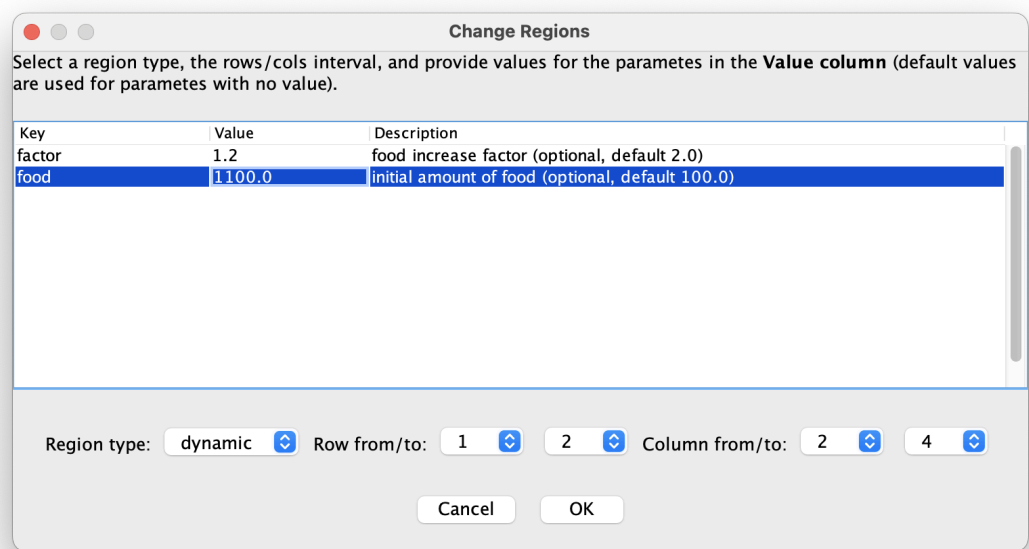
Recuerda que si el usuario ha proporciona un archivo de entrada, hay que usarlo para crear la instancia de `Simulator` y además añadir los animales y regiones usando `load_data` del controlador, y si no lo proporciona crea la instancia de `Simulator` con valores por defecto para la anchura, altura, filas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

# Figuras

## Ventana principal



## Diálogo de cambio de regiones





Visor del mapa

