Interfețe Grafice cu Utilizatorul (GUI) I

1. Scopul lucrării

Obiectivele de învățare ala acestei sesiuni de laborator sunt:

- Înțelegerea modului de realizare a unei interfețe grafice (GUI-Graphical User Interface)
- Acumularea cunoștințelor privind utilizarea claselor și interfețelor importante care sunt necesare în gestionarea aplicațiilor cu interfață grafică
- Acumularea de experiență de programare în gestiunea evenimentelor generate de controalele folosite într-o GUI

Mouse-ul este tratat automat de către majoritatea componentelor, astfel că, în general, nu trebuie să știți de el. Spre exemplu, dacă cineva dă clic pe un buton (`JButton`), veți recepționa un `ActionEvent`, dar nu este nevoie să știți (și n-ar trebui să vă pese) dacă aceasta s-a datorat unui clic cu mouse pe buton sau a fost cauzat de o apăsare de tastă "rapidă" (shortcut).

Grafica. Dacă desenați grafică proprie (d.e., într-un `JPanel`) și aveți nevoie să știți dacă utilizatorul face clic, atunci trebuie să știți despre evenimentele legate de mouse. Puteți adăuga cu ușurință un "ascultător" pentru mouse la un `JPanel`.

2. Containere si componente grafice

Pachetele care conțin elementele de grafică sunt:
`java.awt`
`javax.swing`

AWT este o interfață pentru codul nativ de GUI prezent în sistemul de operare, un învelitor (*wrapper*) al obiectelor grafice ale sistemului de operare, fiind astfel dependent de platforma pe care se implementează. În cazul componentelor grafice **Swing**, mașina virtuală Java este responsabilă de aspectul lor, oferind astfel independență față de sistemul de operare. În plus, Swing vine cu o gamă extinsă de componente și facilități. Pentru clasele AWT (din pachetul `java.awt`), clasele corespunzătoare componentelor Swing (din pachetul `javax.swing`) încep cu prefixul "J", ex.: `JButton`, `JTextField`, `JLabel`, `JPanel`, `JFrame`, sau `JApplet`.

Ierarhia de clase pentru containerele si componentele grafice Swing este ilustrată în Figura 1:

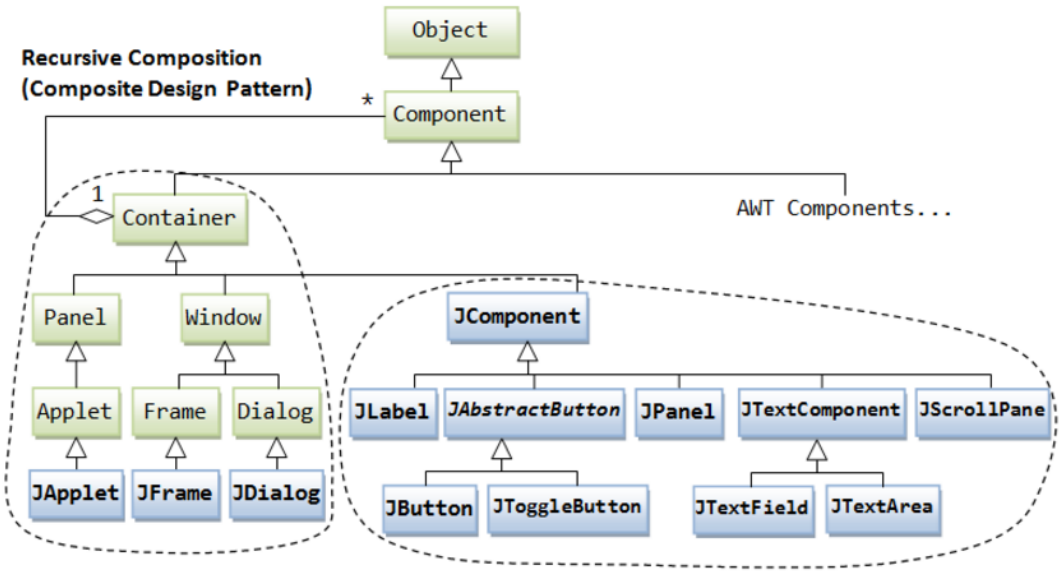

Figura 1. Ierarhia de clase pentru Swing (cu verde: AWT).

Elementele grafice sunt de două tipuri:

1. ***Componente***: componentele sunt obiecte care au o reprezentare grafică ce poate fi afișată pe ecran și care poate interacționa cu utilizatorul. Exemple: `JButtons, JLabels, JTextField` etc.)
2. ***Containere***: componente care pot conține alte componente. În Swing, containerele precum `JFrame, JPanel`, sunt folosite pentru a conține componente. Componentele se pot aranja într-un anume fel în interiorul unui container specificând tipul de *layout* dorit (`FlowLayout, GridLayout, BoxLayout` etc.). Un container poate conține alte containere (subcontainere). Se recomandă, pentru a putea aranja obiectele într-un anume fel în interiorul containerului să se folosească subcontainere.

În figura 2 este prezentat un exemplu pentru elementele grafice de bază:

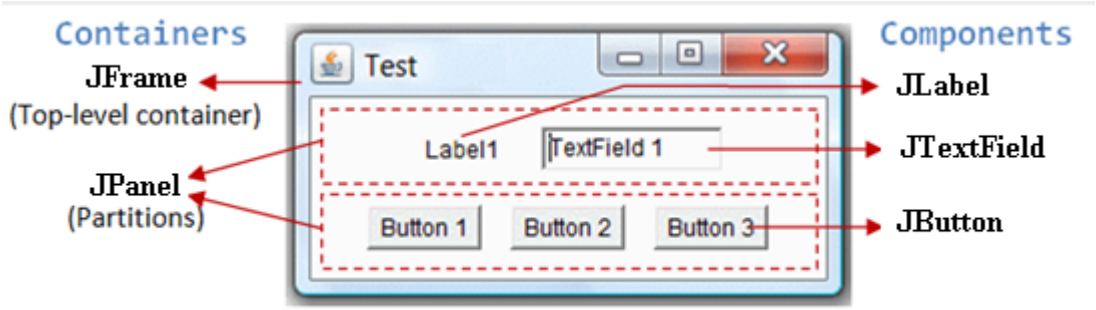

Figura 2. Exemplu de interfață grafică.

2.1 Aranjarea componentelor în containere

Pentru stabili locația componentelor în containere, trebuie specificat modul de aranjare al lor (*layout*). Există mai tipuri de gestionare de aranjare. Câteva dintre cele mai uzitate sunt prezentate în cele ce urmează:

- `FlowLayout` (aranjarea implicită pentru JPanel) – aranjează elementele unul după altul, ca în exemplul din figura 3.

Figura 3. Exemplu de aranjare cu FlowLayout.

Codul Java care poate produce aranjarea din figura 3 poate conține:

```
FlowLayout experimentLayout = new FlowLayout();
...
    compsToExperiment.setLayout(experimentLayout);

    compsToExperiment.add(new JButton("Button 1"));
    compsToExperiment.add(new JButton("Button 2"));
    compsToExperiment.add(new JButton("Button 3"));
    compsToExperiment.add(new JButton("Long-Named Button 4"));
    compsToExperiment.add(new JButton("5"));
```

- `GridLayout` – plasează componentele într-o matrice specificată prin numărul de linii și de coloane, ca în exemplul din figura 4.

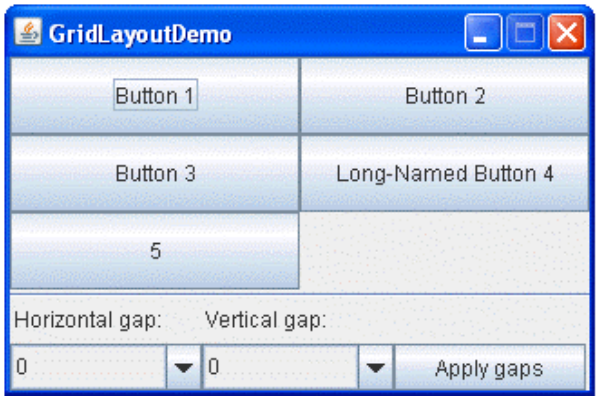

Figura 4. Exemplu de aranjare cu GridLayout.

Codul Java care poate produce aranjarea din figura 4 poate conține:

```
GridLayout experimentLayout = new GridLayout(0,2);
...
        compsToExperiment.setLayout(experimentLayout);

        compsToExperiment.add(new JButton("Button 1"));
        compsToExperiment.add(new JButton("Button 2"));
        compsToExperiment.add(new JButton("Button 3"));
        compsToExperiment.add(new JButton("Long-Named Button 4"));
        compsToExperiment.add(new JButton("5"));
```

- **`BorderLayout`** – plasează componentele în până la cinci zone diferite: est, vest, sud, nord și centru; spațiul extra este plasat în zona centrală.

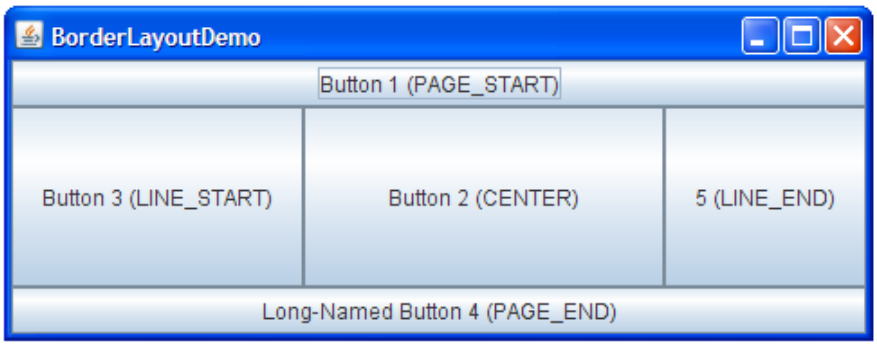

Figura 5. Exemplu de aranjare cu BorderLayout.

Codul Java care poate produce aranjarea din figura 3 poate conține:

```
...
Container pane = aFrame.getContentPane();
JButton button = new JButton("Button 1 (PAGE_START)");
pane.add(button, BorderLayout.PAGE_START);

//Make the center component big, since that's the
//typical usage of BorderLayout.
button = new JButton("Button 2 (CENTER)");
button.setPreferredSize(new Dimension(200, 100));
pane.add(button, BorderLayout.CENTER);
...
```

- Alte tipuri: **`BoxLayout`**, **`CardLayout`**, **`SpringLayout`** etc. (pentru detalii despre acestea citiți referința [1]);

2.2 Realizarea unei aplicații simple cu Interfață grafică

Un exemplu simplu, care nu conține codul de interacțiune, ci doar cel pentru aranjare, este prezentat mai jos.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MyPanel extends JPanel {

      public static void main() {
             JFrame frame  = new JFrame ("Simple Frame");
             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
             frame.setSize(300, 120);

            JPanel panel1 = new JPanel();
            JPanel panel2 = new JPanel();

            JLabel l = new JLabel ("Label1 ");
            JTextField tf = new JTextField("TextField1");
            panel1.add(l);
            panel1.add(tf);
            panel1.setLayout(new FlowLayout());

            JButton b1 = new JButton("Button 1");
            JButton b2 = new JButton("Button 2");
            JButton b3 = new JButton("Button 3");
            panel2.add(b1);
            panel2.add(b2);
            panel2.add(b3);

            JPanel p = new JPanel();
            p.add(panel1);
            p.add(panel2);
            p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));

            frame.setContentPane(p);
            frame.setVisible(true);
       }
}
```

3. Tratarea evenimentelor

3.1 Evenimente

Evenimentele vin de la controalele cu care interacționează utilizatorul. Când definim o interfață utilizator, e nevoie de obicei de o cale pentru a obține informație de la utilizator. Butoanele, meniurile, slider-ele, click-urile pe mouse, etc., generează evenimente atunci când utilizatorul interacționează cu ele. Obiectele eveniment din interfața utilizator sunt transmise de la o **sursă de evenimente** – cum sunt un buton sau un click pe mouse – la un **ascultător de evenimente** – o metodă a utilizatorului care va prelucra obiectele eveniment.
Fiecare control pentru intrare (`JButton, JSlider`, ...) are nevoie de un ascultător de evenimente.
Dacă doriți ca un control să facă ceva atunci când utilizatorul interacționează cu el, atunci trebuie să aveți un ascultător.

Există mai multe feluri de evenimente. Cele mai uzuale sunt prezentate în Tabelul 1.

Sursa evenimentului	**Metoda de adăugare a ascultătorului**	**Metoda ascultătorului**
JButton JTextField JMenuItem	addActionListener()	actionPerformed(actionEvent e)
JSlider	addChangeListener()	stateChanged(ChangeEvent e)
JCheckBox	addItemListener()	itemStateChanged()
Tasta pe componentă	addKeyListener()	keyPressed(), keyReleased(), keyTyped()
Mouse pe componenta	addMouseListener()	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
Mouse mișcat pe componentă	addMouseMotionListener()	mouseMoved(), mouseDragged(),
JFrame	addWindowListener()	windowClosing(WindowsEvent e), …

Tabelul 1. Surse de evenimente, metoda de adăugare a ascultătorului și metoda invocată în ascultător.

Pentru a folosi evenimente, aveți nevoie de clase predefinite, pe care le puteți include folosind **clauze import**:

```
import java.awt.* ;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

3.2 Ascultători

Un ascultător este apelat atunci când utilizatorul interacționează cu interfața, ceea ce provoacă un eveniment. Multe dintre evenimente provin de obicei din interfața utilizator, dar ele pot avea și alte surse (d.e., un contor de timp (Timer)). După crearea unui buton, adăugați-i un ascultător d.e.,

```
btn.addActionListener(obiect_ascultator);
//unde obiect_ascultator este de tipul ButtonListener definit mai jos
```

La clic pe buton se face invocă metoda `actionPerformed()` definită în clasa obiectului ascultător. Metodei i se transmite ca parametru un obiect `ActionEvent`.

Exemplu de clasa care implementează un ascultător:

```
class ButtonListener implements ActionListener{
     public void actionPerformed(ActionEvent e){
         //fa ceva cand se apasa butonul, de exemplu
         ++count;
         tf.setText(count + "");
     }
 }
```

Ascultătorii se pot defini și ca clase imbricate anonime. Exemplu:

```
btnCount.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
```

```
            //fa ceva cand se apasa butonul, de exemplu
            ++count;
            tf.setText(count + "");
        }
    });
```

4. Structura Model-View-Controller (MVC)

Multe interfețe se bazează pe șablonul **Model-View-Controller (MVC).** Ideea este separarea programelor în **Model**, **View** (vedere, vizualizare) (creează afișajul, interacționând cu Modelul după nevoi), și **Controller** (răspunde la cererile utilizatorului, interacționând atât cu Vizualizarea cât și cu Modelul) după nevoi.

Rolul elementelor din arhitectura MVC (potrivit sursei [2])**:**

- **Model** – încapsulează datele specifice unei aplicații și definește logica și computațiile care manipulează și procesează datele respective. În software de întreprindere, modelul servește adeseori ca o aproximare a proceselor din lumea reală. Modelul nu știe despre vederi și controloare. Când se schimbă, în mod tipic modelul notifică observatorii săi despre schimbare.
- **Vederea –** redă conținutul unui model. Specifică exact cum se prezintă utilizatorului datele din model. Dacă modelul se schimbă, vederea trebuie să-și actualizeze prezentarea după nevoi. Aceasta se poate obține folosind un *model push*, în care vederea se înregistrează la model pentru a fi notificată despre schimbări, fie folosind un *model pull*, în care vederea răspunde de apelarea modelului atunci când are nevoie să afișeze cele mai actuale date.
- **Controlorul** – traduce interacțiunile utilizatorului cu vederea în acțiuni pe care le va executa modelul. Într-un client GUI de sine stătător, interacțiunile pot fi click-uri pe butoane, selecții de meniu etc.

4.1 Exemplu de aplicație cu structura MVC: un calculator simplu.

Calculatorul este organizat conform șablonului Model-View-Controller (MVC). Ideea este separarea interfeței utilizator într-o View (vedere, vizualizare care creează afișajul, interacționând cu Modelul după nevoi), și un Controller (răspunde la cererile utilizatorului, interacționând atât cu Vizualizarea cât și cu Controlorul după nevoi). Literatura despre MVC permite o serie de variațiuni, dar toate urmează această idee de bază. Acest model este simplu și poate fi folosit cu apeluri de metodă simple. Dacă sunt interacțiuni mai complexe (d.e. Modelul este actualizat asincron), atunci poate fi necesar un șablon de tip Observer (observator) (cu ascultători). Aspectul său este ilustrat în figura 6.

Figura 6. GUI pentru calculatorul simplu.

Programul principal
Programul principal inițializează interfața și le leagă pe toate împreună.

```
// Calculator în sablonul MVC
//Fred Swartz -- December 2004
import javax.swing.*;

public class CalcMVC {
      // ... Creeaza modelul, vizulizarea, si controlorul. Aceste sunt
      // create aici o data si transmise partilor care au
      // nevoie de ele astfel ca exista o singura copie din fiecare.
```

```
      public static void main(String[] args) {
            CalcModel model = new CalcModel();
            CalcView view = new CalcView();
            CalcController controller = new CalcController(model, view);
            view.setVisible(true);
      }
}
```

Vizualizarea:

```
// Componenta Vizualizare
// Doar prezentare. Nu are acțiuni utilizator.
// Fred Swartz -- December 2004
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class CalcView extends JFrame {
    //... Components
    private JTextField m_userInputTf = new JTextField(5);
    private JTextField m_totalTf     = new JTextField(20);
    private JButton    m_multiplyBtn = new JButton("Multiply");
    private JButton    m_clearBtn    = new JButton("Clear");

    private CalcModel m_model;

    //========================================================= constructor
    /** Constructor */
    CalcView(CalcModel model) {
        //... Set up the logic
        m_model = model;
        m_model.setValue(CalcModel.INITIAL_VALUE);

        //... Initialize components
        m_totalTf.setText(m_model.getValue());
        m_totalTf.setEditable(false);

        //... Layout the components.
        JPanel content = new JPanel();
        content.setLayout(new FlowLayout());
        content.add(new JLabel("Input"));
        content.add(m_userInputTf);
        content.add(m_multiplyBtn);
        content.add(new JLabel("Total"));
        content.add(m_totalTf);
        content.add(m_clearBtn);

        //... finalize layout
        this.setContentPane(content);
        this.pack();

        this.setTitle("Simple Calc - MVC");
        // The window closing event should probably be passed to the
        // Controller in a real program, but this is a short example.
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    void reset() {
        m_totalTf.setText(CalcModel.INITIAL_VALUE);
    }
    String getUserInput() {
```

```
        return m_userInputTf.getText();
    }

    void setTotal(String newTotal) {
        m_totalTf.setText(newTotal);
    }

    void showError(String errMessage) {
        JOptionPane.showMessageDialog(this, errMessage);
    }

    void addMultiplyListener(ActionListener mal) {
        m_multiplyBtn.addActionListener(mal);
    }

    void addClearListener(ActionListener cal) {
        m_clearBtn.addActionListener(cal);
    }
}
```

Controlorul:

```
// Trateaza interactiunea utilizatorului folosind ascultatori.
// Apeleaza Vizualizarea si Modelul dupa nevoi.
// Fred Swartz -- December 2004

import java.awt.event.*;

public class CalcController {
    //... The Controller needs to interact with both the Model and View.
    private CalcModel m_model;
    private CalcView  m_view;

    //=============================================================
constructor
    /** Constructor */
    CalcController(CalcModel model, CalcView view) {
        m_model = model;
        m_view  = view;

        //... Add listeners to the view.
        view.addMultiplyListener(new MultiplyListener());
        view.addClearListener(new ClearListener());
    }


    ////////////////////////////////////// inner class
MultiplyListener
    /** When a mulitplication is requested.
     *  1. Get the user input number from the View.
     *  2. Call the model to mulitply by this number.
     *  3. Get the result from the Model.
     *  4. Tell the View to display the result.
     * If there was an error, tell the View to display it.
     */
    class MultiplyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String userInput = "";
            try {
                userInput = m_view.getUserInput();
                m_model.multiplyBy(userInput);
```

```
                m_view.setTotal(m_model.getValue());

            } catch (NumberFormatException nfex) {
                m_view.showError("Bad input: '" + userInput + "'");
            }
        }
    }//end inner class MultiplyListener


    ////////////////////////////////////////// inner class ClearListener
    /**  1. Reset model.
     *   2. Reset View.
     */
    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            m_model.reset();
            m_view.reset();
        }
    }// end inner class ClearListener
}
```

Modelul:

```
// Acest model este complet independent de interfata utilizator
// Poate fi folosit la fel de usor in linie de comanda sau cu interfata
Web.
// Fred Swartz - December 2004

import java.math.BigInteger;

public class CalcModel {
    //... Constants
    static final String INITIAL_VALUE = "1";

    //... Member variable defining state of calculator.
    private BigInteger m_total;  // The total current value state.

    //================================================================
constructor
    /** Constructor */
    CalcModel() {
        reset();
    }

    //================================================================
reset
    /** Reset to initial value. */
    public void reset() {
        m_total = new BigInteger(INITIAL_VALUE);
    }

    //================================================================
multiplyBy
    /** Multiply current total by a number.
    *@param operand Number (as string) to multiply total by.
    */
    public void multiplyBy(String operand) {
        m_total = m_total.multiply(new BigInteger(operand));
    }
```

```
    //============================================================
setValue
    /** Set the total value.
    *@param value New value that should be used for the calculator total.
    */
    public void setValue(String value) {
        m_total = new BigInteger(value);
    }

    //============================================================
getValue
    /** Return current calculator total. */
    public String getValue() {
        return m_total.toString();
    }
}
```

5. Mersul lucrării

5.1. Studiați și înțelegeți exemplele din laborator. Mai multe exemple găsiți în [3].

5.2. În exemplul de aplicație simplă din secțiunea 2.2 adăugați ascultători celor trei butoane astfel:
- primul buton (Button1) va contoriza de câte ori a fost apăsat acel buton și rezultatul se va actualiza în componenta `JLabel`.
- al doilea buton (Button2) va citi un text introdus de utilizator în `JTextField` și îl va afișa în etichetă. Pentru a citi șirul de caractere din `JTextField` se va apela metoda `getText()` care returnează un `String`.
- la apăsarea celui de-al treilea buton (Button3), se va genera câte o culoare aleatoare pentru cele două panouri.

5.3. Pornind de la exemplul cu Calculator din secțiunea 4.1, adăugați un buton nou pentru adunare. Noua GUI va arăta ca în figura 7.

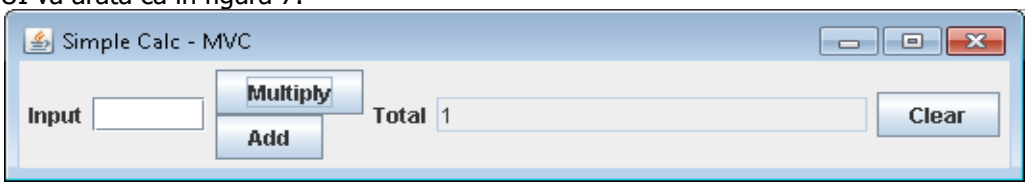

Figura 7. Noua interfața pentru Calculator.

5.4. Respectând structura MVC, implementați un program care să simuleze un convertor valutar. Ca exemplu de GUI folosiți site-ul oficial BNR http://www.cursbnr.ro/convertor-valutar (cf. figura 8)

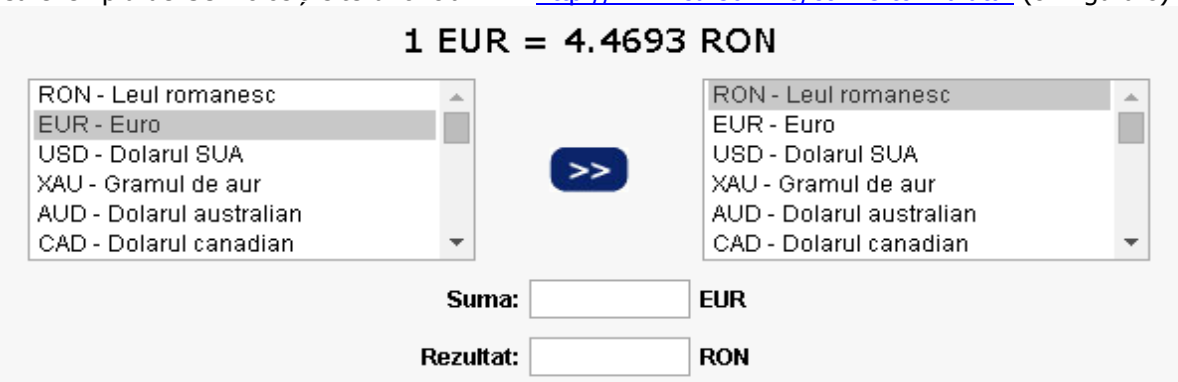

Figura 8. Interfața pentru convertorul valutar.

Indicații implementare: folosiți `JComboBox`-uri [4] pentru a selecta moneda de schimb. Implementați trei opțiuni: RON, EUR, USD.

6. Referințe

[1] Oracle, „Java Tutorials. A Visual Guide to Layout Managers,” 2015. [Interactiv]. Available: https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html. [Accesat 27 November 2016].

[2] R. Eckstein, „Java SE Application Design With MVC,” March 2007. [Interactiv]. Available: http://www.oracle.com/technetwork/articles/javase/index-142890.html. [Accesat 27 November 2016].

[3] C. H. Chuan, „Yet another insignificant programming notes,” January 2016. [Interactiv]. Available: http://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html. [Accesat 27 November 2016].

[4] Oracle, „Java Tutorials. How to Use Combo Boxes,” 2015. [Interactiv]. Available: https://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html. [Accesat 27 November 2016].