Interfețe Grafice cu Utilizatorul (GUI)

1. Scopul lucrării

Obiectivele de învățare ala acestei sesiuni de laborator sunt:

- Înțelegerea modului de folosire a mouse pentru interacțiunea cu utilizatorul
- Acumularea cunoștințelor privind utilizarea claselor și interfețelor importante care sunt necesare în gestionarea aplicațiilor cu infață grafică
- Acumularea de experiență de programare în gestiunea evenimentelor generate de mouse.

Mouse este tratat automat de către majoritatea componentelor, astfel că, în general, nu trebuie să ştiţi de el. Spre exemplu, dacă cineva dă clic pe un buton (**JButton**), veţi recepţiona un **ActionEvent**, dar nu este nevoie să ştiţi (şi 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).

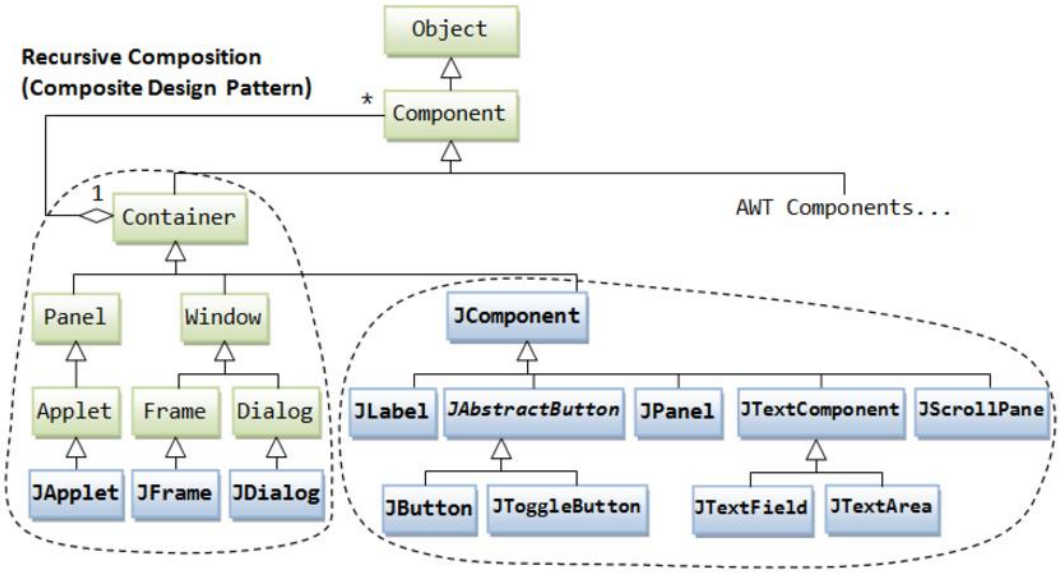
Grafica. Dacă desenaţi grafică proprie (d.e., într-un **JPanel**) şi aveţi nevoie să ştiţi daca utilizatorul face clic, atunci trebuie să ştiţi despre evenimentele legate de mouse. Puteţi dăuga cu uşurinţă un "ascultător" pentru mouse la un **JPanel**.

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 wraper al obiectelor grafice ale sistemului de operare, fiind astfel dependent de platforma pe care se implementează. În cazul coponentelor grafice **Swing**, java virtual machine este responsabilă de aspectul lor, oferind astfel o independență față de OS. În plus, Swing vine cu o gamă extinsă de componente și facilități. În comparație cu clasele AWT (din pachetul java.awt), clasele corespunzătoare componentelor Swing (din pachetul javax.swing) încep cu prefixul "J", ex.: **JButton**, **JTextField**, **JLabel**, **JPanel**, **JFrame**, or **JApplet**.

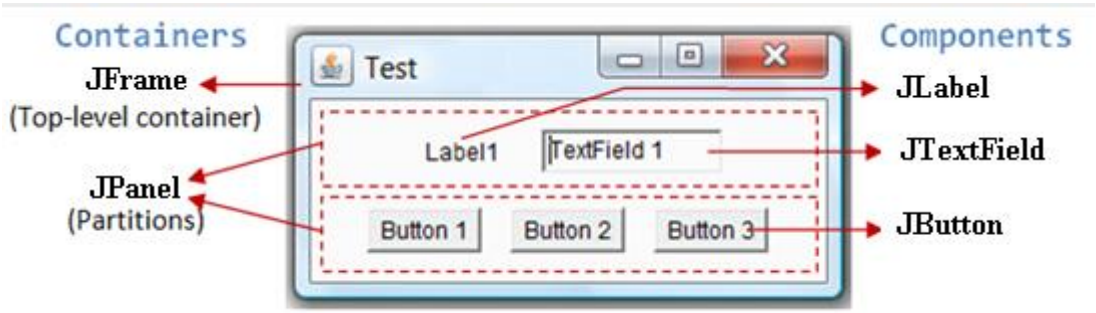
Ierarhia de clase pentru containerele si componentele grafice swing este următoarea:



Elementele grafice sunt de două tipuri:

1. ***Componente***: componenetele sunt entități grafice elementare, precum **JButtons**, **JLabels**, **JTextField** etc.)
2. ***Containere***: containerele precum **JFrame**, **JPanel**, sunt folosite pentru a conține componente. Componentele se pot aranja într-un anume fel în interiorul unui container specificânt tipul de layout dorit (flow, grid sau Box). Un container poate conține subcontainere. Este chiar recomandat, pentru a aputea aranja obiectele într-un anume fel în interiorul containerului să se folosească subcontainere.

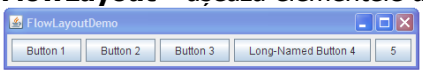
În figura de mai jos este prezentat un exemplu vizual pentru elementele grafice de bază:



2.1 Aranjarea componentelor în containere

Pentru a customiza locația componentelor in containere, trebuie spacificat Layout-ul. Există mai tipuri de layout, dintre care câteva cele mai utilizate:

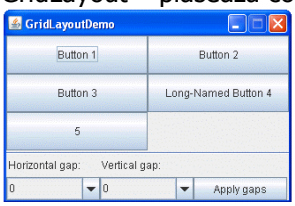
- **FlowLayout** – așează elementele unul după altul.



Ex:

```
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-un tablel specificat prin numărul de linii și de coloane.

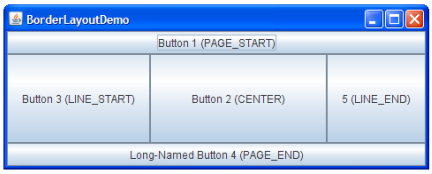


Ex:

```
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: sus, jos, stânga, dreapta și centru; spațiul extra este plasat în zona central.



Ex:

```
...//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. (pt detalii citiți [3]).

2.2 Realizarea unei aplicații simple cu Interfață grafică

```
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 utilizator. Când definim o interfață utilizator, e nevoie de obicei de o cale de a obține informație de la utilizator. Butoanele, meniurile, sliders, clickurile pe mouse, etc. toate generează evenimente atunci când utilizatorul face ceva 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. Figura de mai jos ilustrează ierarhia de moşteniri pentru evenimente.

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:

Control utilizator	***addXXXListener***	***Metoda în ascultător (listener)***

`JButton`
`JTextField`
`JMenuItem`
	`addActionListener()`	`actionPerformed(ActionEvent e)`

`JSlider`	`addChangeListener()`	`stateChanged(ChangeEvent e)`
`JCheckBox`	`addItemListener()`	`itemstateChanged()`
key on component	`addKeyListener()`	`keyPressed(), keyReleased(),` **`keyTyped`**`()`

mouse on component	`addMouseListener()`	**`mouseClicked`**`(), mouseEntered(),`
		`mouseExited(), mousePressed(),`
		`mouseReleased()`

mouse on component	`addMouseMotionListener()`	`mouseMoved(), mouseDragged()`
`JFrame`	`addWindowListener()`	`windowClosing(WindowEvent e), ...`

Clauze import. Pentru a folosi evenimente trebuie incluse aceste clauze import:
import java.awt.* ;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

3.2 Ascultători

Se apelează un ascultător atunci când utilizatorul interacționează cu interfața ceea ce provoacă un eveniment. Deşi evenimentele provin de obicei din interfața utilizator, ele pot avea şi alte surse (D.e., un contor de timp (Timer)). După crearea unui buton, adaugați-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 un apel la metoda actionPerformed() definită în clasa obiectului ascultător. Metodei i se transmite ca parametru un obiect ActionEvent.

Ex de clasa care implementează un ascultător:

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

Ascultătorii se pot defini și ca clase imbricate cu anonimi. Ex:

```
btnCount.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            //fa ceva cand se apasa butonul
            ++count;
            tf.setText(count + "");
         }
      });
```

4. Structura Model-View-Controller (MVC)

Calculatorul este organizat conform şablonului **Model-View-Controller (MVC).** Ideea este separarea interfeței utilizator într-un **Model**, un **View** (vedere, vizualizare) (crează afişajul, interacționând cu Modelul după nevoi), şi un **Controller** (răspunde la cererile utilizatorului, interacționând atât cu Vizualizarea cât şi cu Controlorul după nevoi.

Arhitectură:

- **Model** -Această parte a controlatorului manipulează operațiunile logice și de utilizare de informație (trimisă dinainte de către rangul său superior) pentru a rezulta de o formă ușor de înțeles. Modelul nu depinde de interfata cu utilizatorul. El nu ştie dacă este folosit de o interfața consolă, una grafică sau de Web.
- **View**- Acestui membru al familiei îi corespunde reprezentarea grafică, sau mai bine zis, exprimarea ultimei forme a datelor: interfața grafică ce interacționează cu utilizatorul final. Rolul său este de a evidenția informația obținută până ce ea ajunge la controlator.
- **Controller** - Cu acest element putem controla accesul la aplicația noastră. Pot fi fișiere, scripts sau programe, in general orice tip de informație permisă de interfață. În acest fel putem diversifica conținutul nostru de o formă dinamică și statică, în același timp.

4.1 Exemplu de aplicație cu structura MVC

Se prezintă un exemplu de implementare a unui calculator. Calculatorul este organizat conform şablonului Model-View-Controller (MVC). Ideea este separarea interfeței utilizator în View (vedere, vizualizare), Model (crează afişajul, interacționând cu Modelul după nevoi), şi un Controller (răspunde

la cererile utilizatorului, interacționând atât cu Vizualizarea cât şi cu Controlorul după nevoi). Literatura despre MVC permite o serie de variaŃiuni, dar toate urmează această idee de bază. Acest model este simplu şi poate fi folosit cu mapeluri de metodă simple. Dacă sunt interacțiuni mai complexe (d.e. Modelul este actualizat asincron), atunci poate fi necesar un şablon Observer (observator) (cu ascultători).

Programul principal
Programul principal inițializează totul şi le leagă pe toate împreună.

```
// stucture/calc-mvc/CalcMVC.java -- Calculator in sablonul MVC.
//Fred Swartz -- December 2004
import javax.swing.*;

public class CalcMVC {
	// ... Creaza modelul, vvizulizarea, 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(model);
		CalcController controller = new CalcController(model, view);
		view.setVisible(true);
	}
}}
```

Vizualizarea:

```
// structure/calc-mvc/CalcView.java – componenta Vizualizare
// Doar prezentare. Nu are acŃiuni utilizator.
// Fred Swartz -- December 2004
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class CalcView extends JFrame {
	// ... Constante
	private static final String INITIAL_VALUE = "1";
	// ... 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(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");
            // Evenimentul "window closing" ar trebui probabil transmis
            // Controlorului într-un program real, dar acesta este
            // un scurt exemplu.
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      }

      void reset() {
            m_totalTf.setText(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:

```
// stucture/calc-mvc/CalcController.java - Controlor
// Trateaza interactiunea utilizatorului folosind ascultatori.
// Apeleaza Vizualizarea si Modelul dupa nevoi.
// Fred Swartz -- December 2004
import java.awt.event.*;

public class CalcController {
      // ... Controlorul are nevoie sa interactioneze atât cu
      // Modelul cât si cu Vizualizarea.
      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());
	}

	////////////////////////////////////// clasa interna MultiplyListener
	/*** La solicitarea unei inmultiri. 1. Obtine numarul introdus de
	 * utilizator la Vizualizare. 2. Apeleaza modelul pentru a inmulti prin
	 * acest numar. 3.Obtine resultatul din Model. 4. Spune Vizualizarii sa
	 * afiseze rezultatul.Daca a aparut o eroare spune Vizualizarii sa o
	 * afiseze.
	 */
	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

	////////////////////////////////////// clasa interna ClearListener
	/**
	 * 1. Reseteaza Modelul. 2. Reseteaza Vizualizarea.
	 */
	class ClearListener implements ActionListener {
		public void actionPerformed(ActionEvent e) {
			m_model.reset();
			m_view.reset();
		}
	}// end inner class ClearListener
}
```

Modelul:

```
// structure/calc-mvc/CalcModel.java
// Fred Swartz - December 2004
// Model
// Acest model este complet independent de interfata utilizator
// Poate fi folosit la fel de usor in linie de comanda sau cu interfata Web.
import java.math.BigInteger;

public class CalcModel {
	//... Constante
	private static final String INITIAL_VALUE = "1";
	//... Variabila membru care defineste starea calculatorului.
	private BigInteger m_total; // Valoarea totala curenta

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

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

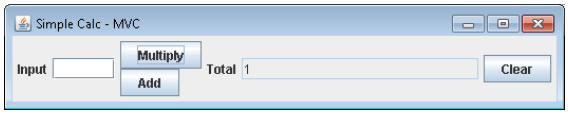
```
	}

	/** Inmulteste totalul current printr-un numar.
	*@param operand Numar (ca String) cu care sa se inmulteasca totalul.
	*/
	public void multiplyBy(String operand) {
		m_total = m_total.multiply(new BigInteger(operand));
	}

	/** Seteaza valoarea totalului.
	*@param value Noua valoare de folosit pe post de total al calculatorului.
	*/
	public void setValue(String value) {
		m_total = new BigInteger(value);
	}

	/** Returneza totalul current al calculatorului. */
	public String getValue() {
		return m_total.toString();
	}
}
```

5. Mersul lucrării

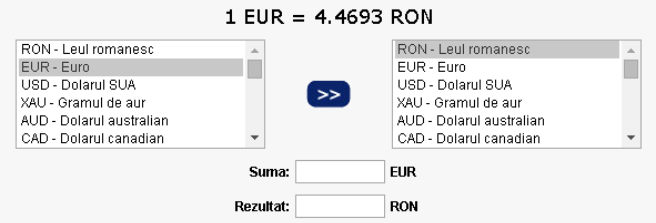
5.1 Studiați și înțelegeți exemplele din laborator

5.2 În exemplul de aplicație simplă din secțiunea 2.1 adăugați ascultători celor trei butoate astfel:
- primul buton va contoriza de câte ori a fost apăsat acel buton și rezultatul se va actualiza în componenta Jlabel.
- al doilea buton va citi un text introdus de utilizator în JtextField și îl va afișa în primul panel la niște coordonate date. Pentru a citi stringul din textField se va apela metoda *getText()* care returnează un String.
- la apăsarea celui de-al treilea buton, se va genera câte o culoare aleatoare pentru cele două paneluri

5.3 Pornind de la exemplul cu Calculator din secțiunea 4.1, adăugați un buton nou pentru adunare. Noul GUI va arăta astfel:



5.4 Respectând structura MVC, implementați un program care să simuleze un convertor valutar. Exemplu de GUI:
Site-ul oficial BNR http://www.cursbnr.ro/convertor-valutar



Indicatii implementare: folosiți JcomboBox-uri [2] pentru a selecta moneda de schimb. Folosiți trei opțiuni: RON, EUR, USD.

6. Referinte

[1] Java Programming Tutorial - Programming Graphical User Interface (GUI)
http://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html
[2] JComboBox Documentation
https://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html
[3] Layout https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html