

DOCUMENTAȚIE PROIECT

CUPRINS

1.Introducere.....	2
2. Descrierea aplicației.....	2
3. Descrierea implementării.....	2
4. Descrierea structurală.....	21
4.1. Descriere arhitecturală.....	21
4.2. Descriere funcțională.....	22
5. Evaluare performanțe.....	23
6. Concluzii.....	23
7. Bibliografie.....	23

Realizat de: Drăghici Andrei 333AA

Universitatea Politehnica București
Facultatea de Automatică și Calculatoare

1. Introducere

Am ales să lucrez la tema cu numărul 10, “**Image Mirroring**”, deoarece mi s-a părut foarte interesantă ideea de a face un cod care să îmi oglindească o imagine așa cum vreau eu. Recunosc că nu am mai făcut până acum un cod(nici în C++) care să realizeze acest lucru(oglindire de imagini), și fiind momentan neobișnuit să lucrez în **JAVA**, mi s-a părut un mod bun de a pune în practică noțiunile studiate atât la curs, cât și la laborator.

2. Descrierea aplicației

Aplicația intitulată “Image Mirroring” presupune citirea din memorie a unei imagini și trecerea ei prin mai multe etape de prelucrare cu scopul de a o oglindi pe o anumită axă (Ox și/sau Oy). După ce a fost oglindită, imaginea este salvată pe disc cu o denumire setată de utilizator.

3. Descrierea implementării

Din punct de vedere al implementării am făcut astfel:

1. În main – ul clasei **MyMain**, apelez funcția **settings**, funcție care îmi generează un frame cu următoarele:
 - Un câmp de text cu label-ul “Fisier Intrare”.
 - Un câmp de text cu label-ul “Fisier Iesire”
 - Un checkbox cu label-ul “X AXIS”.
 - Un checkbox cu label-ul “Y AXIS”.
 - Un buton numit “SUBMIT”.

În codul funcției settings facem următoarele:

- a) Creez un nou frame:

```
frame = new JFrame("CHOOSE AXIS");  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

b) Mă ocup de casetele de text, cât și de label-urile acestora.

```
// Creează JLabel-uri pentru casetele de text
JLabel inputFileLabel = new JLabel("Fisier Intrare:"); //label-urile casutelor de text
JLabel outputFileLabel = new JLabel("Fisier Iesire:");

// Creează casetele de text
JTextField inputTextField = new JTextField(10); //casutele de text in care scriu
//denumirea fisierului/imaginii de intrare,
JTextField outputTextField = new JTextField(10); //respectiv de iesire
```

c) Checkbox-urile

```
// Creează checkbox-uri
JCheckBox xAxisCheckBox = new JCheckBox("X AXIS"); //cele 2 checkbox -uri
JCheckBox yAxisCheckBox = new JCheckBox("Y AXIS");
```

d) Butonul de submit

```
JButton submit = new JButton("SUBMIT"); //butonul de submit
```

e) Creez un panel în care adaug toate cele 5 elemente de mai sus.

Adaug panel-ul în frame și setez frame-ul să fie vizibil, adică să apară pe ecran.

```
// Creează un JPanel pentru a conține componentele
JPanel panel = new JPanel(); //imi creez un jpanel in care bag tot continutul de mai sus

//adaug toate elementele specificate mai sus in panel

panel.add(inputFileLabel);
panel.add(inputTextField);
panel.add(outputFileLabel);
panel.add(outputTextField);
panel.add(xAxisCheckBox);
panel.add(yAxisCheckBox);
panel.add(submit);

frame.add(panel);

frame.pack();
frame.setVisible(true);
```

- f) De fiecare dată când se apasă butonul de submit, iau textul scris din cele doua casuțe de text și verific ce checkbox-uri au fost selectate pentru a parametriza procesarea imaginii. De asemenea, dacă cele doua câmpuri de text nu au fost completate și niciu axa nu a fost selectata, atunci fereastra nu se poate inchide.

```
submit.addActionListener(new ActionListener() { //atunci cand este apasat butonul de submit...

    public void actionPerformed(ActionEvent e) {

        inputStringFile = inputTextField.getText(); //iau textul din casutele de text
        outputStringFile = outputTextField.getText();

        if(xAxisCheckBox.isSelected() && yAxisCheckBox.isSelected()){ //in functie de ce casute(checkbox-uri) sunt selectate
            //setez string-ul axis pe care o sa il folosesc pentru
            axis = "xAxis and yAxis"; //alegerea tipului de oglindire(adica axa/axele pe care
            //fac oglindirea)
        }

        else if(xAxisCheckBox.isSelected() && !yAxisCheckBox.isSelected()){
            axis = "xAxis";
        }

        else if(!xAxisCheckBox.isSelected() && yAxisCheckBox.isSelected()){
            axis = "yAxis";
        }

        if(axis != "" && inputStringFile != "" && outputStringFile != "") //numai atunci cand am toate campurile completate/selectate
            frame.dispose(); //pot sa inchid JFrame-ul in care preiau datele de la utilizat

    }

});
```

În continuare o să explic ce am facut în cadrul metodei main din clasa MyMain.

În main apelez metoda settings(). După care bag programul în așteptare până când fereastra în care parametrizăm prelucrarea imaginii a fost închisă. Acest lucru îl realizez cu ajutorul unei bucle while care nu se opreste decat atunci cand frame.isVisible() == 0. Dacă frame-ul este vizibil , pun thread-ul curent pe sleep 0.5 secunde, după care verific, din nou, daca fereastra mai este vizibilă.

```
settings();

// Programul asteapta pana cand fereastra este inchisa
while (frame.isVisible()) { //atata timp cat fereastra este deschisa
    try {
        Thread.sleep(500); //pune Thread-ul pe pauza si verifica iar dupa
        //500 ms daca fereastra a fost inchisa
    } catch (InterruptedException e) {
        // Nu trebuie sa facem nimic daca sleep-ul este intrerupt
    }
}
```

Mai departe, tot în cadrul main, realizez legătura dintre Producer și Consumer prin intermediul clasei Buffer și legătura dintre Consumer și WriteResult prin intermediul unui pipe.

Setăm pipe-ul propriu zis, adică definim capetele acestuia, care sunt doua obiecte care specifică acest lucru. Adică **out** este un flux prin care ies date și **in** este un flux prin care vin date. De asemenea, trebuie să setăm și să filtrăm tipul de date trimise sau primite prin pipe. De aceea o sa folosim si obiecte de tipul `DataOutputStream` și `DataInputStream`.

Declar și aloc un obiect de tip Buffer.

După care declar si aloc obiectele:

- Producer, ce primește ca parametrii: buffer-ul prin care se realizeaza comunicarea cu Consumer, denumirea thread-ului, “Producer”, și un string prin care îi specific care este imaginea și de unde să o ia.
- Consumer, ce primește ca parametrii: buffer-ul prin care se realizeaza comunicarea cu Producer, denumirea thread-ului, “Consumer”, îi specific ce capat al pipe-ului este și îi dau un string prin care îi spun pe ce axe să facă oglindirea.
- WriteResult, căreia îi specific ce capăt al pipe-ului este și îi spun unde să salveze imaginea primită prin pipe de la Consumer și cu ce denumire.

După care, pornesc cele 3 thread-uri specifice claselor mentionate mai sus.

```
//realizam conexiunea prin pipe
PipedOutputStream pipeOut = new PipedOutputStream();
PipedInputStream pipeIn = new PipedInputStream(pipeOut);

//tipul de data trimis/primit prin pipe
DataOutputStream out = new DataOutputStream(pipeOut);
DataInputStream in = new DataInputStream(pipeIn);

//buffer ne ajuta sa transmitem pixel cu pixel imaginea de la producator la consumator
//si sincronizeaza cele 2 thread-uri, adica Producer si Consumer
Buffer b = new Buffer();

Producer p1 = new Producer(b, "Producer", inputPath + inputStringFile);
Consumer c1 = new Consumer(b, "Consumer", out, axis);
WriteResult wr1 = new WriteResult(in, outputPath + outputStringFile);
p1.start();
c1.start();
wr1.start();
```

2. În clasa Producer:

- Definesc constructorul.

Ii atribui obiectului meu bufferul, calea de unde sa ia imaginea și îi creez un thread de executie.

```
//-----CONSTRUCTORUL CLASEI PRODUCER-----
public Producer(Buffer b, String threadName, String fisier) { //constructor clasei Producer

    buffer = b;
    this.fisier = fisier;
    t = new Thread(this, threadName); //folosim unul din constructorii clasei Thread,
                                     //constructor ce primeste ca parametru un string
                                     //pentru a denumi thread-ul
    System.out.println("Constructor thread = " + threadName);

}

//-----
```

- Metoda de start a thread-ului creat anterior

```
//-----START METHOD-----  
  
//functia care declanseaza thread-ul clasei Producer  
  
public void start() {  
  
    System.out.println("Startin thread = " + t.getName());  
    t.start(); //pornim efectiv thread-ul  
  
}  
  
//-----
```

- Metoda run(ceea ce face efectiv thread-ul definit mai sus)

În această metodă:

- ✓ Citesc imaginea și o salvez într-un obiect de tipul BufferedImage. După care îi salvez dimensiunile și le trimit prin intermediul buffer-ului către Consumer.

```
//Deschidem imaginea si o salvam intr-n obiect de tipul BufferedImage  
  
System.out.println("Deschidem imaginea " + fisier);  
  
image = ImageIO.read(new File(fisier));  
  
//-----WIDTH-----  
  
//iau latimea imaginii si o transmit catre Consumer  
  
int width = image.getWidth();  
  
buffer.put(width);  
  
System.out.println("Latime imagine = " + width);  
  
//-----  
  
//-----HEIGHT-----  
  
//iau inaltimea imaginii si o transmit catre Consumer  
  
int height = image.getHeight();  
  
buffer.put(height);  
  
System.out.println("Inaltime imagine = " + height);  
  
//-----
```

- ✓ După care parcurg imaginea pe linii și coloane și transmit către Consumer valoarea RGB a fiecarui pixel. Când am parcurs un

sfert de imagine, atunci pun thread-ul producer pe sleep și scriu în consolă acest lucru. Cand am trimis toate valorile RGB, îi spun buffer-ului să se oprească(buffer.finished = true).

```
//-----  
  
int limit = height/4; //aici imi setez linia pana la care sa merg  
                        //adica imi delimitez segmentele de  
                        //imagine(4 la numar) pe care le transmit catre Consumer  
//System.out.println("Limita = " + limit);  
  
//-----ITERARE PRIN IMAGINE-----  
  
//incepem sa iteram prin matricea de pixeli si sa transmitem pachete catre Consumer  
//odata ce am citit si transmitem pixel cu pixel o bucata(adica un sfert de imagine),  
//pun thread-ul pe sleep 2000ns, adica Producer intra in starea de Not Runnable  
  
int flag = 0;  
  
for (int y = 0; y < height; y++) { //parcure imaginea pe linii  
  
    if(flag == limit-1){ //cand am ajuns la limita impusa pentru pachete, adica la o bucata de height  
        try {  
            t.sleep(2000); //punem thread-ul in starea de Not Runnable pentru 2000 ns  
            System.out.println("\n\nProducatorul a intrat in starea Not Runnable\n\n");  
        }  
        catch (InterruptedException e) {  
        }  
        flag = 0; //si setez flag-ul la 0 pentru a incepe parcurgerea unui nou sfert din imagine  
    }  
    else flag++;  
  
    for (int x = 0; x < width; x++) { //aici fac parcurgerea pe coloane  
  
        buffer.put(image.getRGB(x, y)); //si transmit, prin intermediul unui buffer catre consumator  
        // valoarea RGB a pixelului de pe pozitia data de x si y  
    }  
  
}  
  
buffer.finished = true; //odata ce am terminat de transmis toti pixelii, buffer-ul se inchide
```

3. În clasa Buffer:

- Am o variabila de tipul Boolean care îmi spune dacă bufferul este închis sau nu.

- Am o metodă care îmi ia tot ce vine de la Producer.

```
//-----GET METHOD-----

public synchronized int get() { //metoda prin care luam tot ce vine de la Producer
                                //si dam mai departe catre Consumer,
                                //adica valoarea RGB a pixelilor, cat si latimea
                                //si inaltimea imaginii

    while (!available) {
        try {
            wait();
            // Asteapta producatorul sa puna o valoare
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    available = false;

    notifyAll ();

    return valoare; //trimitem valoarea pe care o primim
                   // de la Producer catre Consumer

}

//-----
```

- Am o metodă care trimite mai departe către Consumer ceea ce a fost primit de la Producer.

```
//-----PUT METHOD-----  
  
public synchronized void put (int valoare) {  
  
    while (available) {  
  
        try {  
  
            wait();  
  
            // Asteapta consumatorul sa preia valoarea  
  
        } catch (InterruptedException e) {  
  
            e.printStackTrace();  
  
        }  
    }  
  
    this.valoare = valoare; //luam valoarea trimisa de la Producer  
  
    available = true;  
  
    notifyAll ();  
}  
  
//-----
```

4. În clasa Consumer am:

- Constructorul

```
//-----CONSUMER CONSTRUCTOR-----  
  
public Consumer (Buffer b, String threadName, DataOutputStream out, String axis){  
    this.axis = axis;  
    buffer = b;  
    this.out = out;  
    t = new Thread(this, threadName);  
    System.out.println("Constructor thread = " + threadName);  
  
}  
  
//-----
```

- start

```
//-----START METHOD-----  
  
public void start(){  
  
    System.out.println("Starting thread = " + t.getName());  
    t.start();  
  
}  
  
//-----
```

- run(adică ceea ce face thread-ul Consumer)

În această metodă iau dimensiunile imaginii și îmi definesc un vector de dimensiune height*width pentru a băga toate valorile RGB (ale pixelilor) venite de la Producer.

```
width = buffer.get(); //iau valorile latimii si inaltimei pe care  
                        // le-am transmis prin buffer din Producer  
height = buffer.get();  
  
//System.out.println("Latime imagine afis in Consumer = " + width);  
  
//System.out.println("Inaltime imagine afis in Consumer = " + height);  
  
vector = new int[height*width]; //imi definesc un vector in care o sa salvez  
                                // valorile RGB ale pixelilor primite  
int i = 0;  
  
while(buffer.finished == false){ //atata timp cat buffer-ul este activ  
  
    vector[i] = buffer.get(); //iau valoarea si o salvez in vector  
  
    if(i == (height*width - 1))  
        System.out.println("Nu mai vine nimic catre Consumer");  
  
    //System.out.println("Consumatorul a primit: pixel = " + vector[i]);  
  
    i++;  
  
}
```

După care, pe baza acestui vector, reconstruiesc imaginea și o afișez.

```
this.recreateImage(vector); //pe baza vectorului recream imaginea primita  
  
this.showImage(); //afisam imaginea pentru a ne asigura ca totul este in regula
```

Apoi, în funcție de ce axă am ales la început, oglindesc imaginea. Pentru asta am folosit un switch.

```
switch(axis){  
    case "xAxis":{ //oglindire pe axa X  
        ImageMirroringHorizontally mirrorImageHorizontally = new ImageMirroringHorizontally();  
        image = mirrorImageHorizontally.mirrorImage(image);  
        mirrorImageHorizontally.displayImage(image); //afisez imaginea oglindita  
    }break;  
    case "yAxis":{ //oglindire pe axa Y  
        ImageMirroringVertically mirrorImageVertically = new ImageMirroringVertically();  
        image = mirrorImageVertically.mirrorImage(image);  
        mirrorImageVertically.displayImage(image); //afisez imaginea oglindita  
    }break;  
    case "xAxis and yAxis":{ //oglindire atat pe axa X, cat si pe axa Y  
        ImageMirroringHorizontallyAndVertically mirrorImageHorizontallyAndVertically = new ImageMirroringHorizontallyAndVertically();  
        image = mirrorImageHorizontallyAndVertically.mirrorImage(image);  
        mirrorImageHorizontallyAndVertically.displayImage(image); //afisez imaginea oglindita  
    }break;  
}
```

Si la final, transmit imaginea pixel cu pixel prin pipe către WriteResult.

- showImage este o funcție care îmi crează un frame în care se află o imagine primită ca parametru.

```
//-----SHOW IMAGE-----  
  
//aceasta functie are rolul de a afisa o imagine  
  
private void showImage() {  
    JFrame frame = new JFrame();  
    frame.setContentPane(new JLabel(new ImageIcon(image)));  
    frame.setTitle("IMAGINEA PRIMITA DE LA PRODUCATOR");  
    frame.pack();  
    frame.setVisible(true);  
}  
  
//-----
```

- recreateImage este o funcție care primește ca parametru un vector și îmi reface imaginea.

```
//-----RECREATE IMAGE-----  
  
//pe baza unui vector in care am salvat valorile RGB ale pixelilor venite din Producer,  
//recreez imaginea  
  
private void recreateImage(int[] vector){  
    System.out.println("Reconstituim imaginea");  
    int l = 0;  
    image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);  
    for(int k = 0; k < height; k++){  
        for(int j = 0; j < width; j++){  
  
            image.setRGB(j, k, vector[l]);  
            l++;  
  
        }  
    }  
}
```

- `transmitImage` este o funcție ce se folosește de pipe-ul definit anterior pentru a transmite pe același principiu ca la buffer imaginea de la Consumer la WriteResult. Adică, trimitem prin pipe dimensiunile imaginii, apoi parcurgem imaginea și transmitem valorile RGB una câte una. Când am parcurs fiecare sfert de imagine, thread-ul intră în starea de “Not Runnable”, acest lucru fiind specificat și în consolă.

```
private void transmitImage(){

    int flag = 0;
    int limit = height/4;

    try{ //transmit prin pipe inaltimea imaginii
        out.writeInt(height);
    } catch(IOException e){
        e.printStackTrace();
    }

    try{ //transmit prin pipe latimea imaginii
        out.writeInt(width);
    } catch(IOException e){
        e.printStackTrace();
    }

    for(int k = 0; k < height; k++){ //parcure imaginea pe linii

        if(flag == limit-1){ //când am citit un sfert de imagine
            try {
                t.sleep(2000); //punem thread-ul în starea de Not Runnable pentru 2000 ns
                System.out.println("\n\nConsumatorul a intrat în starea Not Runnable\n\n");
            }
            catch (InterruptedException e) {

            }

            flag = 0;
        }
        else flag++;

        for(int j = 0; j < width; j++){ //ma deplasez pe coloane
            try{
                out.writeInt(image.getRGB(j, k)); //trimit prin pipe valoarea
                                                    //RGB a pixelului curent
            } catch(IOException e){
                e.printStackTrace();
            }
        }
    }
}
```

5. În clasa WriteResult:

- Constructorul

```
//-----WRITE RESULT CONSTRUCTOR-----  
  
public WriteResult(DataInputStream in, String fisier){  
    this.in = in;  
    this.fisier = fisier;  
  
}  
  
//-----
```

- run (adică ceea ce face thread-ul nostru)

Aici iau tot ce vine prin pipe și băgăm într-un vector.

Pe baza vectorului recreăm imaginea și apoi o salvăm pe disk.

```
try{    //citesc valoarea inaltimei(venita prin pipe)  
    height = in.readInt();  
}catch (IOException e){  
    e.printStackTrace();  
}  
  
try{    //citesc valoarea latimii(venita prin pipe)  
    width = in.readInt();  
}catch (IOException e){  
    e.printStackTrace();  
}  
  
valori = new int[height*width]; //in acest vector stochesz valorile pixelilor venite prin pipe  
  
//fac stocarea efectiva  
  
for(int i = 0; i < height*width; i++){  
  
    try{  
        valori[i] = in.readInt();  
    }catch (IOException e){  
        e.printStackTrace();  
    }  
  
}  
  
this.recreateImage(valori); //pe baza valorilor venite prin pipe, creez imaginea  
this.showImage(); //afisez imaginea pentru a ma asigura ca totul este in regula  
this.saveImage(); //stochesz imaginea pe disc
```

- saveImage este o metodă care îmi scrie pe disk imaginea mea

```
//-----SAVE IMAGE-----  
  
//aceasta metoda scrie pe disc imaginea prelucrata ce a venit prin pipe  
  
private void saveImage(){  
    File outputFile = new File(fisier);  
    try {  
        ImageIO.write(image, "bmp", outputFile);  
  
        System.out.println("Imagiea a fost scrisa in memorie");  
    } catch (IOException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}  
  
//-----
```

- recreateImage și showImage sunt la fel ca la clasa Consumer

```
//-----RECREATE IMAGE-----  
  
//aceasta metoda imi reconstituie imaginea pe baza a tot ceea ce am primit prin pipe  
  
private void recreateImage(int[] vector){  
    System.out.println("Reconstituim imaginea");  
    int l = 0;  
    image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);  
    for(int k = 0; k < height; k++){  
        for(int j = 0; j < width; j++){  
            image.setRGB(j, k, vector[l]);  
            l++;  
        }  
    }  
}  
  
//-----  
  
//-----SHOW IMAGE-----  
  
//metoda care imi afiseaza efectiv imaginea pe ecran  
  
private void showImage(){  
    JFrame frame = new JFrame();  
    frame.setContentPane(new JLabel(new ImageIcon(image)));  
    frame.setTitle("IMAGINEA PRIMITA DE LA CONSUMATOR");  
    frame.pack();  
    frame.setVisible(true);  
}  
  
//-----
```


6. Interfața ImageMirroringInterface are o metodă neimplementată displayImage

```
public interface ImageMirroringInterface {  
  
    public void displayImage(BufferedImage image);  
  
}
```

7. Clasa abstractă ImageMirroringAbstractClass are metoda de mai sus implementată și încă o metodă mirrorImage căreia îi vom da override în clasele de pe nivelul inferior.

```
public abstract class ImageMirroringAbstractClass implements ImageMirroringInterface{  
  
    //-----DISPLAY IMAGE-----  
  
    //aceasta metoda face afisarea unei imagini  
  
    public void displayImage(BufferedImage image) {  
        // Creează o fereastră cu imaginea  
        JFrame frame = new JFrame();  
        frame.setTitle("IMAGINEA OGLINDITA");  
        frame.setContentPane(new JLabel(new ImageIcon(image)));  
        frame.pack();  
        frame.setVisible(true);  
    }  
  
    //-----  
  
    abstract public BufferedImage mirrorImage(BufferedImage image); //metoda abstracta  
  
}
```

8. Clasele ImageMirroringHorizontally, ImageMirroringvertically și ImageMirroringHorizontallyAndVertically sunt derivate din clasa ImageMirroringAbstractClass și fac override la clasa mirrorImage.
9. Clasa ImageMirroringHorizontally are metoda mirrorImage și metoda displayImage.

- displayImage a fost implementată în ImageMirroringAbstractClass și face afișarea unei imagini primite ca parametru.
- mirrorImage face oglindirea unei imagini primite ca parametru pe axa OX.

În această metodă:

- ✓ îmi definesc un obiect BufferedImage gol numit „mirroredImage”, de aceeași dimensiune primită ca parametru

- ✓ parcurgem imaginea și luăm fiecare pixel în parte. La pixelul curent luăm coordonata x-1 și o scădem din width-ul imaginii, urmând ca mai apoi această valoare să fie salvată într-o variabilă de tip int numită „mirroredX”.
- ✓ Iau valoarea RGB a pixelului de la pixelul de pe linia x si coloana y din imagine și o salvez în variabila „pixel” de tip int
- ✓ Mă duc în mirroredImage și pun valoarea „pixel” pe poziția mirroredX și y

```
@Override
public BufferedImage mirrorImage(BufferedImage image) {
    // TODO Auto-generated method stub

    int width = image.getWidth();
    int height = image.getHeight();
    BufferedImage mirroredImage = new BufferedImage(width, height, image.getType());

    // Iterăm prin pixelii imaginii originale
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            // Reflectăm coordonatele pixelului pe axa x
            int mirroredX = width - x - 1;
            //System.out.println(x+" "+y);
            // Copiem pixelul din imaginea originală în imaginea reflectată
            int pixel = image.getRGB(x, y);
            mirroredImage.setRGB(mirroredX, y, pixel);
        }
    }
    System.out.println("Imaginea a fost oglindita.");
    return mirroredImage;
}
```

10. Clasa ImageMirroringVertically are metoda mirrorImage și metoda displayImage.

- displayImage a fost implementată în ImageMirroringAbstractClass și face afișarea unei imagini primite ca parametru.
- mirrorImage face oglindirea unei imagini primite ca parametru pe axa OY.

În această metodă:

- ✓ îmi definesc un obiect BufferedImage gol numit „mirroredImage”, de aceeași dimensiune primită ca parametru
- ✓ parcurgem imaginea și luăm fiecare pixel în parte. La pixelul curent luăm coordonata y-1 și o scădem din height-ul

imaginii, urmând ca mai apoi această valoare să fie salvată într-o variabilă de tip `int` numită „`mirroredY`”.

- ✓ Iau valoarea RGB a pixelului de la pixelul de pe linia `x` și coloana `y` din imagine și o salvez în variabila „`pixel`” de tip `int`
- ✓ Mă duc în `mirroredImage` și pun valoarea `pixel` pe poziția `x` și `mirroredY`.

```
@Override
public BufferedImage mirrorImage(BufferedImage image) {
    // TODO Auto-generated method stub
    int width = image.getWidth();
    int height = image.getHeight();
    BufferedImage mirroredImage = new BufferedImage(width, height, image.getType());

    // Iterăm prin pixelii imaginii originale
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            // Reflectăm coordonatele pixelului pe axa y
            int mirroredY = height - y - 1;
            //System.out.println(x+" "+y);
            // Copiem pixelul din imaginea originală în imaginea reflectată
            int pixel = image.getRGB(x, y);
            mirroredImage.setRGB(x, mirroredY, pixel);
        }
    }
    System.out.println("Imaginea a fost oglindita.");
    return mirroredImage;
}
```

11. Clasa `ImageMirroringHorizontallyAndVertically` are metoda `mirrorImage` și metoda `displayImage`.

- `displayImage` a fost implementată în `ImageMirroringAbstractClass` și face afișarea unei imagini primite ca parametru.
- `mirrorImage` face oglindirea unei imagini primite ca parametru atât pe axa `OX`, cât și pe axa `OY`.

În această metodă:

- ✓ îmi definesc un obiect `BufferedImage` gol numit „`mirroredImage`”, de aceeași dimensiune primită ca parametru
- ✓ parcurgem imaginea și luăm fiecare pixel în parte. La pixelul curent luăm coordonata `x-1` și o scădem din `width`-ul imaginii,

urmând ca mai apoi această valoare să fie salvată într-o variabilă de tip `int` numită „`mirroredX`”.

- ✓ parcurgem imaginea și luăm fiecare pixel în parte. La pixelul curent luăm coordonata `y-1` și o scădem din `height`-ul imaginii, urmând ca mai apoi această valoare să fie salvată într-o variabilă de tip `int` numită „`mirroredY`”.
- ✓ Iau valoarea RGB a pixelului de la pixelul de pe linia `x` și coloana `y` din imagine și o salvez în variabila „`pixel`” de tip `int`
- ✓ Mă duc în `mirroredImage` și pun valoarea pixel pe poziția `mirroredX` și `mirroredY`.

```
@Override
public BufferedImage mirrorImage(BufferedImage image) {
    // TODO Auto-generated method stub

    int width = image.getWidth();
    int height = image.getHeight();
    BufferedImage mirroredImage = new BufferedImage(width, height, image.getType());

    // Iterăm prin pixelii imaginii originale
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            // Reflectăm coordonatele pixelului pe axa x
            int mirroredX = width - x - 1;
            int mirroredY = height - y - 1;
            //System.out.println(x+" "+y);
            // Copiem pixelul din imaginea originală în imaginea reflectată
            int pixel = image.getRGB(x, y);
            mirroredImage.setRGB(mirroredX, mirroredY, pixel);
        }
    }
    System.out.println("Imaginea a fost oglindita.");
    return mirroredImage;
}
```

Ultimele 3 clase descrise mai sus sunt folosite în cadrul metodei `run` a clasei `Consumer`, unde folosim un `switch` pentru a alege modul în care oglindim imaginea. Dacă vrem să oglindim o imagine pe OX, atunci o să definim un obiect de tipul `ImageMirroringHorizontally`, aplicăm metoda `mirrorImage` și apoi metoda `displayImage` pentru a vedea ca oglindirea a reușit.

La fel și pentru celelalte 2 case-uri din `switch`-ul menționat anterior.

4. Descrierea structurală

4.1 Descriere arhitecturală

Din punct de vedere arhitectural, avem în proiectul nostru 2 pachete: Pachetul 1(“**package1**”), în care se află un singur fișier, fișierul de test numit “**MyMain**”, și Pachetul 2(“**package2**”), în care am celalalte fișiere .java care realizează oglindirea propriu-zisă a imaginii.

În Pachetul 2(“**package2**”), am următoarele:

1. Pentru prelucrarea efectivă a imaginii:
 - **ImageMirroringInterface**
 - **ImageMirroringAbstractClass**
 - **ImageMirroringHorizontally**
 - **ImageMirroringVertically**
 - **ImageMirroringHorizontallyAndVertically**
2. Pentru realizarea celorlalte cerințe de implementare:
 - **Consumer**
 - **Buffer**
 - **Producer**
 - **WriteResult**

Când vorbim de prelucrarea efectivă a imaginii, am o arhitectură pe 3 niveluri. Pe primul nivel am interfața **ImageMirroringInterface** ce conține o singură metodă(metodă neimplementată în cadrul interfeței) **displayImage** ce primește ca parametru un obiect de tipul **BufferedImage**.

Pe cel de-al doilea nivel am **ImageMirroringAbstractClass** care “**implements**” **ImageMirroringInterface** și în care implementăm funcționalitatea metodei precizate mai sus. De asemenea, tot în această clasă, mai am o metodă abstractă numită **mirrorImage**, căreia îi vom face **override** în clasele de pe nivelul 3.

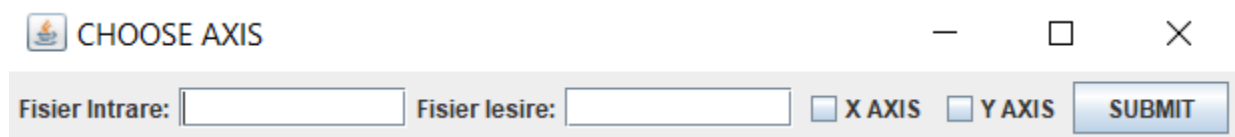
Pe cel de-al treilea nivel, am 3 clase: **ImageMirroringHorizontally**, **ImageMirroringVertically** și **ImageMirroringHorizontallyAndVertically**. Fiecare dintre aceste clase “**extends**” clasa **ImageMirroringAbstractClass**. În

fiecare dintre aceste clase am făcut **override** la metoda abstractă din clasa super, metoda numită **mirrorImage**.

Pentru realizarea celorlalte cerințe de implementare, am **Consumer** și **Producer** care “**implements**” **Runnable**, clasa **Buffer** și clasa **WriteResult** care “**extends**” **Thread**.

4.2 Descriere funcțională

În urma apăsării butonului de “Run”, pe ecran apare următoarea fereastră:



În această fereastră, selectăm denumirea fișierului de intrare, setăm denumirea fișierului de ieșire și axele pe care vrem să facem oglindirea. După ce am setat parametrii aplicației, apăsăm “SUBMIT” și aplicația începe treaba.

Prin “aplicația începe treaba” mă refer la următoarele:

1. Producer citește imaginea din fișierul “inputImages” și o salvează într-un **BufferedImage**.
2. Tot în cadrul clasei **Producer**, trimitem dimensiunile imaginii (height și width) prin **Buffer** către **Consumer** și parcurgem imaginea pixel cu pixel și transmitem tot prin intermediul lui **Buffer** valoarea fiecărui pixel către **Consumer**. Când am reușit să trimitem un sfert de imagine către **Consumer**, **Producer** intră în starea de “Not Runnable” pentru o secundă. De fiecare dată când **Producer** intră în starea menționată mai sus, un mesaj va apărea pe ecran.
3. **Consumer** ia tot ce vine prin **Buffer** de la **Producer** și reasamblează/reconstruiește imaginea exact așa cum era atunci când **Producer** a citit-o din fișier. După care, folosind funcția **mirrorImage**, parcurgem imaginea și facem oglindirea.
4. După ce am făcut oglindirea în cadrul clasei **Consumer**, începem să trimitem pixel cu pixel imaginea către **WriteResult** prin intermediul unui pipe.

Trimiterea valorilor pixelilor imaginii oglindite către **WriteResult** se face pe același principiu cu cel menționat anterior (trimiterea valorilor pixelilor

de la Producer la Consumer). Practic, parcurgem imaginea oglindită și de fiecare dată când am reușit să trimitem un sfert de imagine, Consumer intră în starea de “Not Runnable” pentru 1 secundă, după care continuă să trimită pixelii în continuare(valorile RGB ale pixelilor de fapt). De asemenea, un mesaj va apărea de fiecare dată în consolă când Consumer e în starea “Not Runnable”.

Ceea ce am uitat să specific aici, este că înainte de a trimite valorile RGB ale pixelilor către WriteResult, trimit dimensiunile imaginii(pentru a o putea reconstrui în WriteResult).

5. Pe baza dimensiunilor și a valorilor furnizate de Consumer prin pipe, reconstruim imaginea în cadrul funcției WriteResult și apoi o salvăm, cu denumirea specificată la început în cadrul interfeței grafice, în folder-ul “outputImages”.

5. Evaluare performanțe

Pentru evaluarea performanțelor am salvat în fiecare metodă run a claselor: Producer, Consumer și Writeresult, atât timpul de început al execuției, cât și timpul de final de execuție al metodei, după care am scăzut din momentul final, momentul de început și am împărțit rezultatul la 1000000 pentru a obține durata în milisecunde.

6. Concluzii

Mi-a plăcut foarte mult să lucrez la acest proiect, deoarece am putut să pun în aplicare noțiunile studiate de Java și am putut să creez ceva frumos ce are utilitate în viața de zi cu zi. Pe scurt, am realizat o aplicație ușor de utilizat care folosește thread-uri de execuție și pipe-uri pentru a oglindi o imagine pe care utilizatorul o dă ca input.

7. Bibliografie

- ✓ Cursul de la disciplina Aplicații Web cu Suport Java, mai exact am folosit:
 - ❖ Cursul 9 și Cursul 11 pentru realizarea cerințelor 15 și 16.
- ✓ Sursele lucrate la laborator pentru a-mi face o idee despre cum ar trebui să folosesc conceptele de OOP.

- ✓ <https://www.geeksforgeeks.org/image-processing-in-java-creating-a-mirror-image/>