Isprobajte kako Vaša tražilica radi. Kvalitetu pretrage moguće je poboljšati na niz načina (koje ovdje NE trebate raditi). Primjerice, mogli biste sve različite oblike iste riječi svesti na osnovni oblik: {zgrade, zgradu, zgradom, zgradi, …} → zgrada. Trenutna implementacija sve ove riječi tretira kao potpuno različite i nepovezane. Više o ovoj tematici može doznati na kolegiju "Analiza i pretraživanje teksta" (diplomski studij, profil računarska znanost).

Sav programski kod mora biti u ovom projektu. Ako želite koristiti implementaciju vektora koju ste prethodno napisali za jednu od zadaća, iskopirajte relevantni dio razreda u ovaj projekt (i pazite je li kod dokumentiran!).

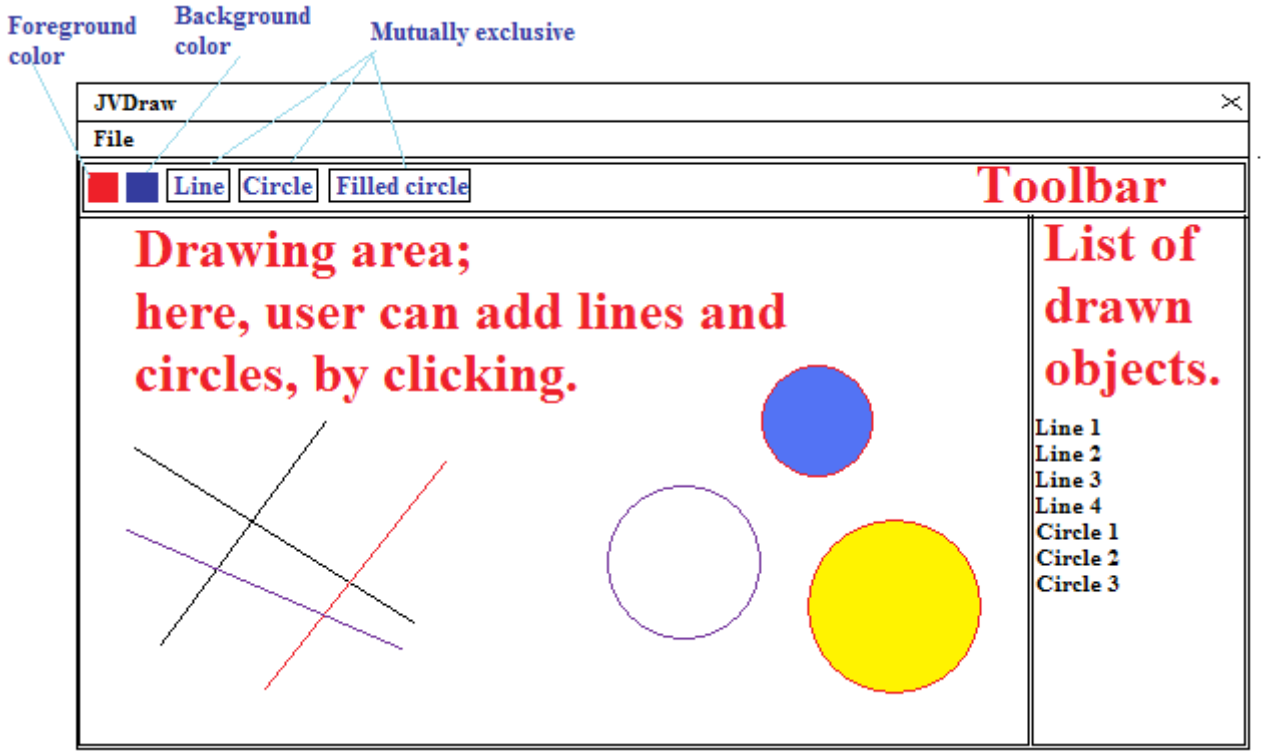
Problem 2.

Important: please read the whole text of this problem before trying to solve it. Identify all the classes and interfaces you are required to create. Sketch a class diagram and relations among the classes – who extends who, who implements who, and who has a reference to who. Identify the design patterns used; label each class/interface according to its purpose in the design pattern. Read carefully the rest of problem and then try to model the relationship among JColorArea, ColorChangeListener, IColorProvider, DrawingModel, DrawingModelListener, JDrawingCanvas, DrawingObjectListener. Show the relationship with class diagram. Once you understand the roles and the relationships among classes, start to implement your solution. Scan your drawings before you start any actual coding: you will have to upload them as well.

Create a new Maven projekt, groupId **`hr.fer.zemris.java.studentVASJMBAG.hw12`**, artifactId **`jvdraw`**.

All classes developed as part of this problem must go into package `hr.fer.zemris.java.hw12.jvdraw` (and its subpackages, if necessary). We are developing a GUI application called *JVDraw*, which is a simple application for vector graphics. The main class (a class which is used to start the program) must be `hr.fer.zemris.java.hw12.jvdraw.JVDraw`.

When started, program will show an empty canvas. Sketch of the program is shown below.



The developed program will allow user to draw lines, circles and filled circles. Program features menubar, toolbar, drawing canvas and object list.

Toolbar has five components: there are two `JColorArea` components and three mutually exclusive `JToggleButtons` (only one can be selected at any time).

JColorArea

Write the code for this component; extend it from `JComponent`. Override its method `getPreferredSize()` so that it always returns a new dimension object with dimensions 15x15. Add a property `Color selectedColor` and when painting the component, just fill its entire area with this color. Give it a constructor that accepts the initial value for `selectedColor`. When user clicks on this component, component must open color chooser dialog and must allow user to select color that will become new selected color. See:

http://docs.oracle.com/javase/8/docs/api/javax/swing/JColorChooser.html

Wire all necessary listeners in components constructor – this is behavior that the component is responsible for and not any outside component. Considering the `selectedColor`, this component must behave as a Subject in Observer pattern. So define a new interface for the observers:

```
public interface ColorChangeListener {
  public void newColorSelected(IColorProvider source, Color oldColor, Color newColor);
}
```

In `newColorSelected` method, the source is the component that has access to a color. Define an interface:

```
public interface IColorProvider {
  public Color getCurrentColor();
}
```

Modify `JColorArea` so that it implements this interface and offers selected color through `getCurrentColor()` method. Equip `JColorArea` with methods for observer registration and deregistration:

```
public void addColorChangeListener(ColorChangeListener l);
public void removeColorChangeListener(ColorChangeListener l);
```

When user changes the selected color, notify all registered listeners about the change. **Add single component to the JFrame's bottom** (for example, derive it from `JLabel`). This component must be a listener on both `JColorArea` instances. At all times, it must display text like this:

"`Foreground color: (255, 10, 210), background color: (128, 128, 0).`"

In parentheses are given red, green and blue components of the color.

Mutually exclusive buttons

Read:

http://docs.oracle.com/javase/8/docs/api/javax/swing/ButtonGroup.html
http://docs.oracle.com/javase/8/docs/api/javax/swing/JToggleButton.html
http://docs.oracle.com/javase/tutorial/uiswing/components/button.html

You must offer to user three tools: drawing of lines, drawing of circles (no filling) and drawing of filled circles (area is filled with background color, circle is drawn with foreground color). Objects are added using

mouse clicks. For example, if the selected tool is a line, the first click defined the start point for the line and the second click defines the end point for the line. Before the second click occurs, as user moves the mouse, the line is drawn with end-point tracking the mouse so that the user can see what will be the final result. Circle and filled circle are drawn similarly: first click defines the circle center and as user moves the mouse, a circle radius is defined. On second click, circle is added.

Right side of the window should be occupied by a list of currently defined objects. Each object must have its automatically generated name (an example is shown in given illustration; however, feel free to implement it any way you like).

Devise appropriate class hierarchy for modeling geometrical shapes so that this list can treat all object equally. Assume the top class to be `GeometricalObject`. When user double-clicks an object in this list, a property dialog must appear. Model the content of the dialog as a single `JPanel` and then use `JOptionPane.showConfirmMessage` giving the panel as the message. For lines, user must be able to modify start and end coordinate and line color. For (unfilled) circle, user must be able to modify center point, radius and color; for filled circle user must be able to modify center point, radius, circle outline and circle area color.

Define an interface `DrawingModel` as follows:

```
interface DrawingModel {

  public int getSize();
  public  GeometricalObject getObject(int index);
  public void add(GeometricalObject object);

  public void addDrawingModelListener(DrawingModelListener l);
  public void removeDrawingModelListener(DrawingModelListener l);

}
```

Define `DrawingModelListener` as this:

```
public interface  DrawingModelListener {
  public void objectsAdded(DrawingModel source, int index0, int index1);
  public void objectsRemoved(DrawingModel source, int index0, int index1);
  public void objectsChanged(DrawingModel source, int index0, int index1);
}
```

In `DrawingModel`, graphical objects have its defined position and it is expected that the image rendering will be created by drawing objects from first-one to last-one (this is important if objects overlap).
Interval defined by `index0` to `index1` is inclusive; for example, if properties of object on position 5 change, you are expected to fire `objectChanged(..., 5, 5)`.

Implement the central component as a component derived from the `JComponent`; name it `JDrawingCanvas`. Register it as a listener on the `DrawingModel`. Each time it is notified that something has changed, it should call a `repaint()` method. When in process of adding a new object by mouse, after a final click occurs (so you know all parameters for the object), dont call the `repaint()`; instead, add the newly created object to the `DrawingModel` – it will fire `objectsAdded(...)` method and then you will call the `repaint()` as a response to that notification.

When creating the object list component, use `JList` with a custom made list model. Develop a `ListModel` (for this you can extend your model from abstract class `AbstractListModel` which already implements listener registration/deregistration/notification functionality) named `DrawingObjectListModel`. This model

must not have its own list of object – drawing object are stored in `DrawingModel`. Implement `DrawingObjectListModel` to be an object adapter for the `DrawingModel`: it must store a reference to the `DrawingModel` and implement all methods in such a way that the information is retrieved from `DrawingModel`. Please note that you will want to make `DrawingObjectListModel` a listener on `DrawingModel`, so that, when user defines a new object by clicking in `JDrawingCanvas`, the `DrawingObjectListModel` can get a notification that the model has changed and that it can re-fire necessary notifications to it's own listeners (to the `JList` that shows the list of currently available objects).

Menus

Under the File menu you are required to implement several actions.

You must provide *Open*, *Save* and *Save As* actions. Each of these actions reads or writes a text file with extension `*.jvd`. An example for this file is given below.

```
LINE 10 10 50 50 255 255 0
LINE 50 90 30 10 128 0 128
CIRCLE 40 40 18 0 0 255
FCIRCLE 40 40 18 0 0 255 255 0 0
```

In file there is one row per object; space is used as element separator. Definitions of lines start with `LINE`, of unfilled circles with `CIRCLE` and of filled circles with `FCIRCLE`. The meaning of numbers is:

```
LINE x0 y0 x1 y1 red green blue
CIRCLE centerx centery radius red green blue
FCIRCLE centerx centery radius red green blue red green blue
```

For `FCIRCLE`, first three color components define outline color and last three color components fill color.

You also must provide an export action. When user selects export, you must ask him which format he wants (offer: JPG, PNG, GIF) and then ask him to select where he wants to save the image. You can make this in two steps, or you can immediately open save dialog with allowed extension `jpg`, `png` and `gif` and then determine what user selected by inspecting the file extension after the dialog is closed. Look at the objects in `DrawingModel` and find out the bounding box (the minimal box that encapsulates the whole image). Create an `BufferedImage` and the export procedure as follows:

```
BufferedImage image = new BufferedImage(
    box_width, box_height, BufferedImage.TYPE_3BYTE_BGR
);
Graphics2D g = image.createGraphics();
 … draw objects …
g.dispose();
File file = …;
ImageIO.write(image, "png", file);
Tell-user-that-images-is-exported.
```

For file formats (second argument) you must provide strings “png”, “gif” or “jpg”.

Lets see a simple example. Lets assume that the document model holds just a single line: (10, 100) to (100, 20). The bounding box is a box whose top-left corner is in (10, 20); it has width 100-10=90 and height 100-20=80. You would create an image 90x80 pixels. When drawing objects, you would translate all coordinates left for 10 pixels and up for 20 pixels. The idea is that the produced rendering corresponds to the interior of the bounding box, so that, if all of your object have x-coordinate grater than 100, you won't get 100 pixels of

blank image. If you determine that bounding box has negative coordinates, this procedure will shift objects to the right so that exported image will again be OK.

And finally, you must provide “Exit” action that will check to see if `DrawingModel` has changed since the last saving; if so, you must ask user if he wants to save the image, cancel the exit action or reject the changes.