# 16th homework; JAVA, Academic year 2017./2018.; FER

This homework has two problems. Solve them as two maven projects. You will export each project separately into different ZIP file and you will upload both ZIP files. You will not be able to lock the upload before two ZIP archives are uploaded.

## *Problem 1.*

Zadatak rješavate kao Maven projekt čiji je groupID jednak **`hr.fer.zemris.java.jmbag0000000000.cmdapps`** (zamijenite 0000000000 Vašim JMBAG-om), a artefactID jednak **`trazilica`**.

Jeste li se ikada zapitali kako radi Google? Kako rade drugi sustavi koji korisniku omogućavaju pretraživanje tekstovnih baza podataka? U okviru ovog zadatka upoznat ćemo se s jednostavnom implementacijom tražilice tekstovnih dokumenata. Naša implementacija imat će svojih mana i neće biti savršena, ali daje teorijsku podlogu na čijoj su nadogradnji implementirani mnogi kompleksniji sustavi.

**Predstavljanje dokumenata**

Da bismo mogli mjeriti sličnost odnosno različitost tekstnih dokumenata (ili zadanog upita pretraživanja i tekstnih dokumenata), nestrukturirani tekst potrebno je u nekom obliku strukturirati odnosno svaki dokument predstaviti u nekom formatu pogodnom za primjenu željenih algoritama. Najčešća reprezentacija tekstnih dokumenata je tzv. reprezentacija *vrećom riječi* (engl. *bag of words*) kod koje je bitno koje se riječi (i potencijalno koliko puta) pojavljuju u dokumentu, dok se poredak riječi u dokumentu efektivno zanemaruje. Iako na predstavljanjem dokumenata "vrećama riječi" gubimo informacije na sintaktičkoj, diskursnoj i pragmatičkoj razini, pokazalo se da je takva reprezentacija sasvim prikladna za zadatke koji ne zahtijevaju duboku semantičku analizu teksta, kao što su grupiranje, kategorizacija i pretraživanje tekstnih dokumenata.

Da bismo dokumente mogli predstaviti u obliku vreća riječi, moramo prvo odrediti vokabular s kojim raspolažemo. Vokabular čine sve riječi koje se nalaze u svim dokumentima kolekcije dokumenata za koju treba provesti grupiranje. Kada odredimo koje se sve riječi nalaze u tekstnoj kolekciji, dokumente kolekcije predstavljamo kao vektore tih riječi. Svaki element vektora predstavlja jednu riječ iz vokabulara. Pri tome je jako bitno naglasiti da se iz vokabulara, osim interpunkcije i drugih specijalnih znakova, najčešće izbacuju tzv. zaustavne riječi (engl. *stopwords*) – semantički siromašne riječi poput zamjenica, prijedloga, veznika, pomoćnih glagola, čestica i sl. Empirijski je pokazano da izbacivanje zaustavnih riječi ima pozitivan učinak. Neka nam je, primjerice, dana kolekcija sa sljedeća tri dokumenta:

- $d_1$ = "*Frodo i Sam nalaze se usred Mordora. Frodo i Sam.*"
- $d_2$ = "*Frodo kao Frodo nije pozdravio Gandalfa.*"
- $d_3$ = "*Sam se naljutio na Gandalfa. Ah Sam.*"

Vokabular kolekcije koja se sastoji od ova tri dokumenta jest skup:

$$\{Frodo, Sam, nalaze, Mordora, pozdravio, Gandalfa, naljutio\}.$$

Ostale riječi koje se pojavljuju u dokumentima (npr. *i*, *nije*, *usred*) smatramo zaustavnim riječima i zbog toga nisu uvrštene u vokabular. U nastavku su ukratko opisana tri često korištena načina predstavljanja dokumenata kao vreća riječi (za dani vokabular).

Binarni vektori dokumenata

Kod binarnog predstavljanja dokumenata za svaku se riječ iz vokabulara bilježi samo je li se ili nije pojavila u dokumentu. Pretpostavimo li gornju kolekciju od 3 dokumenta i prethodno izgrađeni vokabular, binarne reprezentacije dokumenata $d_1$-$d_3$ dane su kako slijedi:

- $d_1$ = [1, 1, 1, 1, 0, 0, 0]
- $d_2$ = [1, 0, 0, 0, 1, 1, 0]
- $d_3$ = [0, 1, 0, 0, 0, 1, 1]

pri čemu svaki indeks vektora predstavlja prisutnost u dokumentu one riječi koja se nalazi na istom indeksu nalazi u vokabularu. Primjerice, jedinica na drugom elementu vektora $d_1$ i $d_3$ znači da se riječ *Sam* nalazi u dokumentima $d_1$ i $d_3$, dok nula na istome mjestu u $d_2$ znači da se riječ *Sam* ne nalazi u dokumentu $d_2$.

Frekvencijski vektori dokumenata

Ukoliko se neka riječ nalazi u dokumentu više puta, ona je značajnija za dokument od neke druge riječi koja se manje puta pojavi u dokumentu. Ovu intuiciju možemo u vektor dokumenta ugraditi na način da umjesto binarnih vektora (je li riječ prisutna ili nije u dokumentu) dokument predstavimo cjelobrojnim vektorima kod kojih svaki element označava koliko se puta riječ pojavila u dokumentu. Prehodni dokumenti (uz nepromijenjen vokabular) u ovoj bi shemi bili predstavljeni kao:

- $d_1$ = [2, 2, 1, 1, 0, 0, 0]
- $d_2$ = [2, 0, 0, 0, 1, 1, 0]
- $d_3$ = [0, 2, 0, 0, 0, 1, 1]

TF-IDF vektori dokumenata

Ukoliko se neka riječ pojavljuje često u nekom dokumentu, ali čisto kao posljedica toga što se ta riječ pojavljuje često u svim dokumentima (odnosno generalno često u danome jeziku), onda ta riječ nije posebno informativna za dani dokument. Takve su upravo zaustavne riječi pa ih upravo zato početno izbacujemo iz vokabulara. Kod tzv. TF-IDF (engl. *term frequency-inverse document frequency*) sheme uvažavaju se dvije intuitivne pretpostavke:

- Riječ je važnija za semantiku dokumenta što se češće u njemu pojavljuje (*TF komponenta*)
- Riječ je manje važna za semantiku dokumenta što se češće pojavljuje po drugim dokumentima (*IDF komponenta*)

Za svaki par riječ-dokument (w,d), TF-IDF vrijednost se računa prema sljedećoj formuli:

$$tfidf(w,d) = tf(w,d) \cdot idf(w,D)$$

gdje je *w* riječ iz vokabulara, *d* pojedinačni dokument, a *D* kolekcija svih dokumenata s kojima raspolažemo. Pri tome je *tf(w, d)* frekvencija (tj. broj) pojavljivanja riječi *w* u dokumentu *d*, dok se IDF komponenta računa kao:

$$idf(w,D) = \log \frac{|D|}{|\{d \in D : w \in d\}|}$$

tj. kao logaritam omjera ukupnog broja dokumenata u kolekciji i broja dokumenata koji sadrže riječ *w*. Vrijednost *tfidf*(*Frodo*,$d_1$) primjerice iznosi:

$$tfidf(Frodo, d_1) = tf(Frodo, d_1) \cdot idf(Frodo, \{d_1, d_2, d_3\}) = 2 \cdot \log \frac{3}{2} \approx 0.81$$

Naime, *Frodo* se u dokumentu $d_1$ pojavljuje 2 puta (TF komponenta), radimo sa skupom od 3 dokumenta, i od ta tri dokumenta, njih 2 (nije bitno jednom ili više puta) sadrže riječ *Frodo* (IDF komponenta).

Vektori dokumenata u ovoj shemi sadrže TF-IDF pripadnih riječi.

Semantička sličnost dokumenata

Jednom kada smo dokumente predstavili kao vektore nad vokabularom, njihovu sličnost mjerimo računanjem kosinusa kuta između pripadnih im vektora u višedimenzionalnom prostoru čija je dimenzija određena veličinom vokabulara tekstne kolekcije. Neka je $v_{d_i}$ vektorska repezentacija (binarna, frekvencijska ili TF-IDF) dokumenta $d_i$. Tada se sličnost dvaju dokumenata mjeri kao:

$$\text{sim}(d_i, d_j) = \frac{v_{d_i} \cdot v_{d_j}}{||v_{d_i}|| \cdot ||v_{d_j}||}$$

što nije ništa drugo doli skalarni produkt vektora dokumenata podijeljen umnoškom normi tih istih vektora. Ako malo razmislite, brzo ćete uočiti da je to direktno definicija kosinusa kuta koji u vektorskom prostoru zatvaraju ta dva vektora: što su vektori bliži, kosinus kuta će biti veći (bliži 1).

Ovako definiranu mjeru sličnosti između dokumenata možete izravno koristiti u algoritmima grupiranja, ako nam je cilj u kolekciji dokumenata pronaći slične dokumente nekom zadanom dokumentu. Međutim, ova mjera je direktno primjenjiva i na postupak pretraživanja: kada korisnik zada upit (kao niz riječi), taj se upit procesira na isti način kao i dokumenti i izgradi se njegova vektorska reprezentacija. Potom se gleda sličnost vektora upita sa svim vektorima dokumenata i vraćaju se dokumenti najsličniji unesenom upitu.

Vaš zadatak

Na Ferku u repozitoriju imate ZIP arhivu sa 60 tekstovnih dokumenata te tekstovnu datoteku sa zaustavnim riječima za hrvatski jezik. Tekstovne datoteke spremljene su uporabom kodne stranice UTF-8. Napravite program koji s korisnikom komunicira kroz konzolu (glumi jednostavnu ljusku).

Evo primjera interakcije s tim programom. Pogledajte način pozivanja programa za informacije o nazivu i paketu.

```
java -cp target/classes hr.fer.zemris.java.hw16.trazilica.Konzola  d:\clanci

Veličina riječnika je 10896 riječi.

Enter command > bla
Nepoznata naredba.

Enter command > query darovit glumac zadnje akademske klase
Query is: [darovit, glumac, zadnje, akademske, klase]
Najboljih 10 rezultata:
[ 0] (0.1222) d:\clanci\vjesnik-1999-12-7-kul-5
[ 1] (0.0151) d:\clanci\vjesnik-1999-12-3-kul-3
[ 2] (0.0104) d:\clanci\vjesnik-1999-9-5-kul-2

Enter command > type 0
----------------------------------------------------------------
```

Dokument: d:\clanci\vjesnik-1999-12-7-kul-5
---

Predstava dobre vjere

Dječje kazalište u Osijeku / Praizvedba »Badnjak u garaži« Davora Špišića i Saše Anočića, po motivima Kozarčeve priče / Nov iskaz osječkih glumaca, praksom pretežito lutkara

...

Predstavu »Badnja u garaži« likovno je vrlo dobro osmislio Željko Zorica, čijoj sceni su se pridružili izvanredni kostimi Mirjane Zagorec. Osobito valja istaknuti prinos maskera i vlasuljara Josipa Saboa. Igor Valeri je autor primjerena glazbe, a Ksenija Zec bila je odlična redateljeva suradnica za koreografiju i scenski pokret. Premijerna publika ispratila je »Badnjak u garaži« s odobravanjem, u kojem se osjetio i tračak nedoumice o budućem životu predstave. Uostalom, najbolje da sama predstava dokaže svoju opstojnost.

---

Enter command > results
[ 0] (0.1222) d:\clanci\vjesnik-1999-12-7-kul-5
[ 1] (0.0151) d:\clanci\vjesnik-1999-12-3-kul-3
[ 2] (0.0104) d:\clanci\vjesnik-1999-9-5-kul-2

Enter command > exit

Program treba podržati sljedeće naredbe.

“query”: obavlja pretraživanje. Iz primljenog upita izbacuje sve riječi koje nisu u vokabularu izgrađenom na temelju analize baze dokumenata. Što je ostalo od riječi, ispisuje na zaslon. Potom radi pretraživanje. Ispisuje 10 najboljih rezultata (dokumenata najsličnijih upitu prema TF-IDF kriteriju), ili manje ako se prije pojave dokumenti sličnosti 0. Ispisana lista je rangirana po sličnosti. U svakom retku je ispisan redni broj, sličnost (na četiri decimale) te staza do dokumenta.

“type”: prima redni broj rezultata i na zaslon ispisuje dokument. Naredbu ima smisla pokretati tek nakon što je zadan neki upit. Naredba ispisuje puni naziv datoteke te tekst koji pohranjen u datoteci (u gornjem primjeru, skratio sam prikazani tekst zbog uštede mjesta; naredba ispisuje kompletan tekst na zaslon).

“results”: ponovno na zaslon ispisuje rang-listu rezultata koje je pronašao zadnje-izvršena naredba “query”.

“exit”: izlazi iz konzole.

Napomena: postupak izgradnje vektorske reprezentacije svih dokumenata (koristeći tf-idf) potrebno je napraviti pri pokretanju programa. Upiti više ne rade direktno s dokumentima, već samo gledaju izgrađene vektore koje čuvate u memoriji. Postupak obrade baze tekstova mora rezultirati

1. vokabularom,
2. izgrađenim vektorima dokumenata (koji su na neki način povezani sa stazom do dokumenta na disku) te
3. pomoćnim vektorom idf(w,D).

Pomoćni vektor (treća točka) koristit ćete kod pretraživanja dokumenata. Kada korisnik zada upit, isti ćete tretirati kao novi dokument na temelju kojeg ćete izgraditi tf-vektor (zato je važno da sačuvate vokabular kako biste znali što od riječi iz upita trebate zanemariti; ako upit donese novu riječ koja ne postoji u prikupljenom vokabularu, ta se riječ iz upita zanemaruje), i potom ćete komponente tf-vektora pomnožiti

(element-s-elementom-po-parovima) s komponentama pomoćnog vektora (točka 3, idf-vektor). Naime, nema smisla upit stvarno tretirati kao novi dokument pa nanovo računati idf-vektor nad čitavim skupom dokumenata proširenog novim "dokumentom" (upitom).

Kako radimo s pretpostavkom da dokumenata ima jako puno, ne možemo ih sve istovremeno učitati u memoriju. Stoga ćete izgradnjutraženih vektora morati napraviti u dva prolaza: u prvom čitate dokument po dokument, vadite riječi i gradite vokabular. Kad je vokabular izgrađen, još jednom ćete proći kroz sve dokumente, svaki učitati, povaditi riječi i stvoriti tražene vektore.

Naredba "type" otvara datoteku s diska i ispisuje ju.

Prilikom pokretanja, korisnik programu zadaje stazu do direktorija u kojem su tekstovi. Program pretražuje taj direktorij i analizira sve dokumente koje pronađe u tom direktoriju (ili bilo kojem poddirektoriju).

Pri ekstrakciji riječi iz dokumenata, "riječ" je kontinuirani niz slova (tj. sve za što Character.isAlphabetic(c) daje *true*). Sve što nisu slova, zanemarite. Posljedica je da ćete "pero27strašni" isparsirati kao dvije riječi "pero" i "strašni". To je OK.

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!).

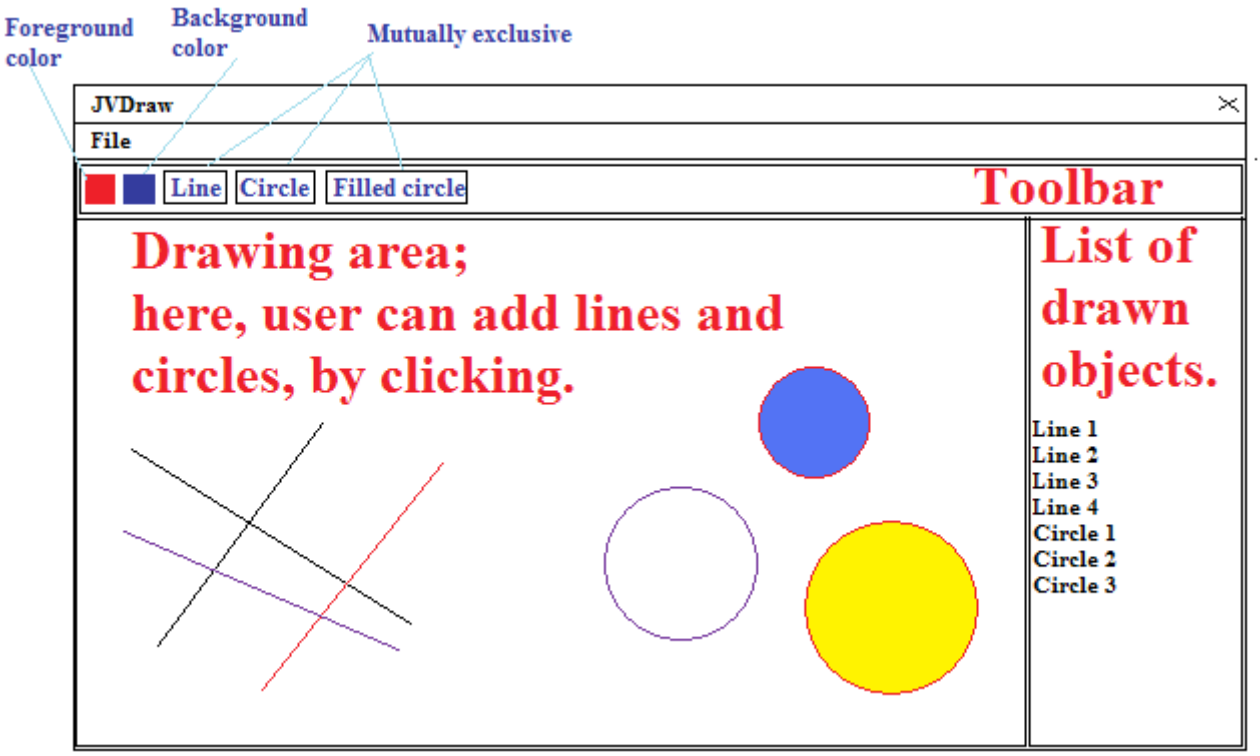
## *Problem 2.*

Important: please read the whole text of this problem before trying to solve it. Identify all classes and interfaces you are required to create. To help yourself, sketch a class diagram and relations among the classes – who extends who, who implements who, and who has a reference to who. Try to 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.

Create a new Maven projekt, groupId **`hr.fer.zemris.java.jmbag0000000000.cmdapps`** (replace `0000000000` with your actual JMBAG), artifactId **`jvdraw`**.

All classes developed as part of this problem must go into package `hr.fer.zemris.java.hw16.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.hw16.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 the necessary listeners in component's constructor – this is the behavior that the component is responsible for, and not any outside component. Considering the property `selectedColor`, this component must behave as a *Subject* in the *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 the color. Define an interface:

```
public interface IColorProvider {
  public Color getCurrentColor();
  public void addColorChangeListener(ColorChangeListener l);
  public void removeColorChangeListener(ColorChangeListener l);
}
```

Modify `JColorArea` so that it implements this interface and offers selected color through `getCurrentColor()` method, as well as the methods for observer registration and deregistration.

When user changes the selected color, notify all of the registered listeners about the change. **Add a 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.

If this component is called `X`, it should have a constructor such as:
`X(IColorProvider fgColorProvider, IColorProvider bgColorProvider) {…}`
so that it can "remember" color providers and register itself as listener on them. In `initGUI()`, you would setup this components with the code like this:

```
JColorArea fgColorArea = new JColorArea(…);
JColorArea bgColorArea = new JColorArea(…);
X bottomColorInfo = new X(fgColorArea, bgColorArea);
```

**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. In order to produce clean and maintainable code, utilize *State* Design Pattern (https://en.wikipedia.org/wiki/State_pattern). Model the State as interface:

```
interface Tool {
  public void mousePressed(MouseEvent e);
  public void mouseReleased(MouseEvent e);
  public void mouseClicked(MouseEvent e);
  public void mouseMoved(MouseEvent e);
  public void mouseDragged(MouseEvent e);
  public void paint(Graphics2D g2d);
}
```

Create a separate classes which implement this interface for each tool (adding lines, adding circle, adding filled circles); you can build more elaborate hierarchy to minimize code duplication. These concrete States (State1, State2, … in design pattern diagram) can receive through constructor any needed references required to implement the desired functionality (e.g. reference to `DocumentModel`, references to `IColorProvider` objects, etc). Your main program should have a single reference representing the "current state"; click on tool buttons in toolbar should change what current state is. Add to `JDrawingCanvas` appropriate mouse listeners which should forward received notifications to the current state. Implement `paintComponent` in `JDrawingCanvas` to render all elements from the `DocumentModel` and then to forward paint request to the current state; this way, if the current state wants to add something on image, it can.

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, but do not implement it like show in this image; later in this text, a concrete specification which you should follow is given).

Devise appropriate class hierarchy for modeling geometrical shapes so that this list can treat all object equally. Assume the top class to be `GeometricalObject`. For each geometrical object you should support several operations: implement them using *Visitor* Design Pattern (https://en.wikipedia.org/wiki/Visitor_pattern). In order to do so, model Visitor as interface GeometricalObjectVisitor:

```
interface GeometricalObjectVisitor {
  public abstract void visit(Line line);
  public abstract void visit(Circle circle);
  public abstract void visit(FilledCircle filledCircle);
}
```

Then GeometricalObject will have appropriate accept method:

```
abstract class GeometricalObject {
  ...
  public abstract void accept(GeometricalObjectVisitor v);
  ...
}
```

You should implement two visitors: `GeometricalObjectPainter` which receives through constructor a reference to Graphics2D and then can be used to paint all geometrical objects; and `GeometricalObjectBBCalculator` which can be used to calculate the bounding box for complete image (you will need it for export described later). When implemented correctly, the following pseudocode should work:

```
GeometricalObjectBBCalculator bbcalc = new GeometricalObjectBBCalculator();
for(GeometricalObject o : allObjectsFromModel) {
 o.accept(bbcalc);
}
Rectangle boundingBox = bbcalc.getBoundingBox();
```

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.showConfirmDialog` 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. When user press DEL-key while the list has focus, the object selected in list must be deleted; if user press '+' (plus) key, the selected object must be shifted one place up; if user press '-' (minus) key, the selected object must be shifted one place down (this operations can have effect of rendered image for overlapping objects).

To support described `GeometricalObject`'s property editing, define a new class:

```
abstract class GeometricalObjectEditor extends JPanel {
  public abstract void checkEditing();
  public abstract void acceptEditing();
}
```

For each concrete geometrical object define appropriate subclass (for example, for Line define LineEditor) which accepts in constructor a reference to appropriately subtyped geometrical object, creates appropriate swing components and initializes them with values retrieved from given geometrical object. When checkEditing is called, the method must check if fields are correctly filled and if not, must throw some exception. When acceptEditing is called, the values from all fields must be written back into given geometrical object. With this implementation, our geometrical object should offer a method for creating a `GeometricalObjectEditor` for itself:

```
abstract class GeometricalObject {
  ...
  public abstract GeometricalObjectEditor createGeometricalObjectEditor();
  ...
}
```

With this code in place (and appropriate subclasses created), you will be able to write double-click JList handler like this (treat it as pseudocode):

```
GeometricalObject clicked = ask-jlist-who-was-double-clicked
GeometricalObjectEditor editor = clicked.createGeometricalObjectEditor();
if(JOptionPane.showConfirmDialog(…, editor, …, OK_CANCEL) == OK) {
  try {
    editor.checkEditing();
    editor.acceptEditing();
  } catch(Exception ex) {
    tell-user-that-data-was-invalid (ex.getMessage())
    alternative: if checkEditing got us here, show message and reopen editor
  }
}
```

`GeometricalObject` class must be a *Subject* (Observer Design Pattern) which can notify interested *Observers* that its state has been changed. Model the *Observer* with `GeometricalObjectListener` interface.

```
interface GeometricalObjectListener {
  public void geometricalObjectChanged(GeometricalObject o);
}
```

The class `GeometricalObject` should have appropriate registration and deregistration methods:

```
class GeometricalObject {
  ...
  public void addGeometricalObjectListener(GeometricalObjectListener l);
  public void removeGeometricalObjectListener(GeometricalObjectListener l);
  ...
}
```

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)`.

When adding `GraphicalObject` to the `DrawingModel`, the `DrawingModel` must register itself (or anonymous object) as a `GeometricalObjectListener` on added `GraphicalObject`. In doing so, whenever any of `GeometricalObject`'s properties is changed, it will notify `DrawingModel`, and the `DrawingModel` will then notify its observers that object at appropriate position has changed (which will result, for example, with canvas repaint so that the shown image is accurate).

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), don't 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* (Design Pattern!) 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 the 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).

Important: the text shown in Jlist for each graphical object should be:

- for lines: "Line (3,5)-(20,30)", that is, start point – end point
- for unfilled circle: "Circle (20,10), 7", that is, center point and radius,
- for filled circle: "Filled circle (20,10), 7, #FF0000", that is, center point, radius, color

The image shown at the start of this problem text is different, so ignore that part of the image.

**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:

```
GeometricalObjectBBCalculator bbcalc = …
...use bbcalc to determine bounding box
Rectangle box = bbcalc.getBoundingBox();
BufferedImage image = new BufferedImage(
    box.width, box.height, BufferedImage.TYPE_3BYTE_BGR
);
Graphics2D g = image.createGraphics();
 … apply translation transform to g for -box.x, -box.y
 … draw objects using GeometricalObjectPainter initialized with g …
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. So before drawing, you would apply translation transform on g for (-10, -20). 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.

**Please note.** You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

**If you need any help, I have reserved a slot for consultations as follows: Tuesday at 10 AM, Wednesday at 9 AM. Feel free to drop by my office (after e-mail).**

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are not required to unit test code in this homework.

When your complete homework is done, pack each project into separate zip archive with name `hw16-0000000000-1.zip` and `hw16-0000000000-2.zip` (replace zeros with your JMBAG); make sure there is **NO** `target` directory present. Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is June 28th 2018. at 07:00 AM.

Upload checklist:

1. HW16-0012345678-1.zip
2. HW16-0012345678-2.zip

You have to upload both files in order to be able to lock the upload.