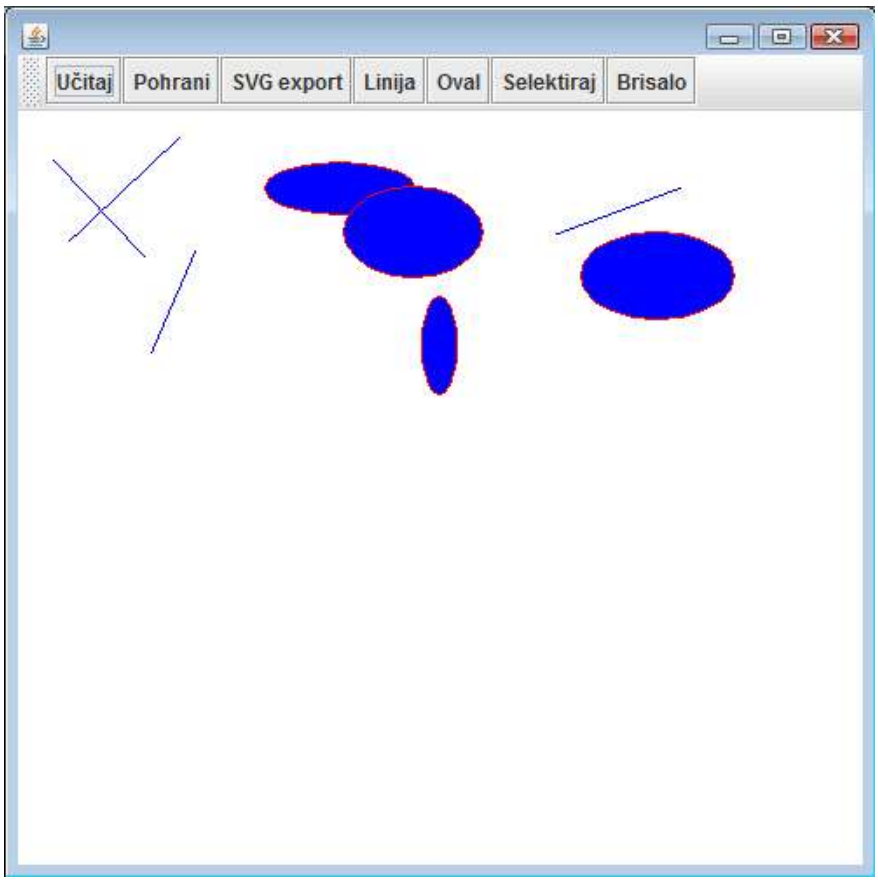
Četvrta laboratorijska vježba iz Oblikovnih obrazaca u programiranju:
Program za uređivanje vektorskih crteža

U okviru 4. laboratorijske vježbe razvijamo program za izradu vektorskih crteža. Program treba omogućiti interaktivno dodavanje geometrijskih oblika poput pravocrtnih linijskih segmenata i elipsa, njihovo grupiranje, brisanje, promjenu redosljeda iscrtavanja (što je važno kod prikaza preklapajućih objekata), naknadnu izmjenu te translatiranje. Program također treba omogućiti pohranu i učitavanje crteža u "nativnom" formatu kao i transparentno prikazivanje crteža na ekranu te eksportiranje u vektorski grafički format SVG (pogledajte **ovo**, **ovo**, **ovo**, **ovo**).

Prilikom izrade ovog rješenja koristit ćemo sljedeće oblikovne obrasce:

- *Promatrač* opisuje odnose između podatkovnog modela crteža i prikaznih komponenata.
- *Kompozit* omogućava transparentno provođenje operacija kako nad pojedinačnim, tako i nad grupiranim elementima.
- *Iterator* za obilazak elemenata crteža.
- *Prototip* omogućava izradu alatne trake za unos novih elemenata crteža (oblika) koja ne bi ovisila o podržanim konkretnim geometrijskim oblicima.
- *Tvornica* za stvaranje konkretnih oblika na temelju simboličkog naziva pri učitavanju crteža.
- *Stanje* omogućava dodavanje novih alata bez izmjena u komponenti koja obrađuje korisnički unos (pomaci miša, unos preko tipkovnice i slično).
- *Most* za transparentno iscrtavanje crteža i eksportiranje u različite slikovne formate poput formata SVG.

Ilustracija koja prikazuje izgled prozora programa s učitanom slikom `ooup-lab4-slika1.txt` prikazana je u nastavku.



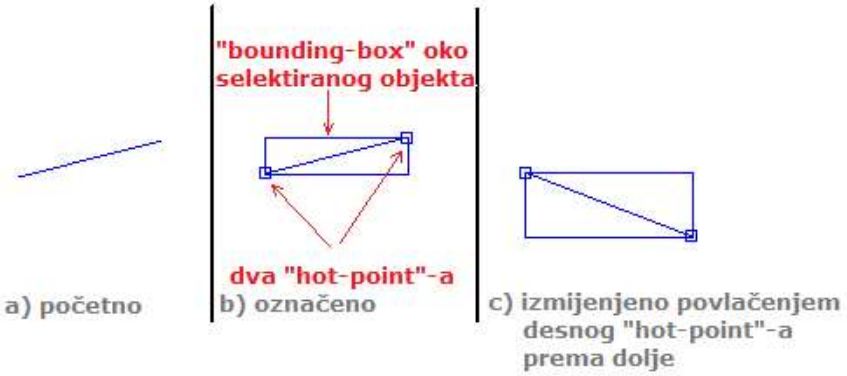
Glavni prozor programa sastoji se od dvije komponente: alatne trake te platna koje prikazuje sliku i omogućava njezino uređivanje. Za alatnu traku koristite gotovu komponentu koja se nudi u standardnim bibliotekama programskog jezika koji ćete koristiti za rješavanje ove vježbe. Platno za crtanje izvedite iz najjednostavnije komponente iste biblioteke koja Vam nudi mogućnost crtanja površine (kao i u prethodnoj vježbi: ako ste u Javi, to će biti `javax.swing.JComponent`, ako ste u C#-u, to će biti `System.Windows.Forms.Control`).

U okviru programa koji razvijamo, grafički objekti bit će korišteni u nekoliko različitih scenarija:

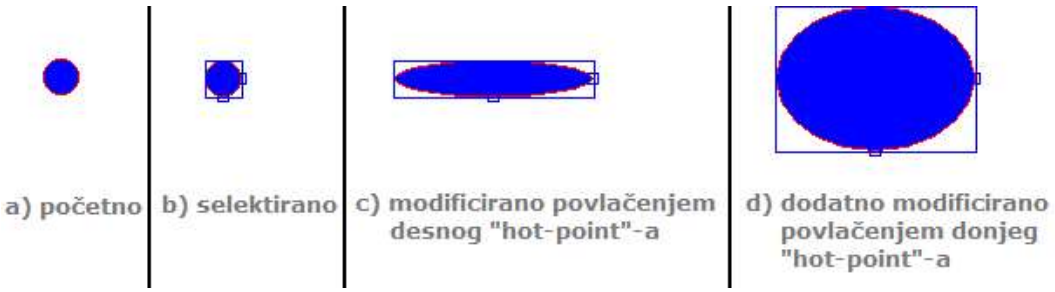
1. Grafičke objekte htjet ćemo nacrtati u grafičkoj komponenti koja sliku prikazuje korisniku.
2. Grafičke objekte htjet ćemo modificirati.
3. Grafičke objekte htjet ćemo grupirati te im mijenjati redoslijed iscrtavanja kako bismo kod preklapajućih objekata mogli definirati što će korisnik vidjeti.

4. Grafičke objekte htjet ćemo pohraniti u nativni format iz kojeg ćemo ih kasnije znati učitati.
5. Grafičke objekte htjet ćemo exportati u SVG format tako da sliku mogu koristiti drugi programi.

Konceptualno, svaki grafički objekt bit će definiran preko "hot-point"-a. Jedan "hot-point" predstavlja jednu karakterističnu točku objekta. Tako će linijski segment biti definiran s dva "hot-point"-a koji ujedno predstavljaju početnu i završnu točku linijskog segmenta. Slika u nastavku prikazuje (a) jedan linijski segment, (b) isti linijski segment kada je selektiran pa su prikazani i njegovi "hot-point"-i te bounding-box, te (c) modificirani linijski segment koji je dobiven tako što je korisnik mišem "uhvatio" desni "hot-point" i odvukao ga vertikalno prema dolje.



Svaki oval također je definiran preko dva "hot-point"-a (program treba podržati samo nerotirane ovale): desni "hot-point" određuje jednu polu-os, dok donji "hot-point" određuje drugu polu-os. Centar ovala je točka koja je presjecište okomitog pravca koji prolazi kroz donji "hot-point" te vodoravnog pravca koji prolazi kroz desni "hot-point".



1. Model grafičkog objekta

Model vektorskog crteža treba sadržavati listu referenci na grafičke objekte koje trebaju koristiti različiti drugi objekti. Predloženi model prikazan je u nastavku i nudi jedinstveno sučelje prema svim svojim korisnicima (što može i ne mora biti dobro -- razmislite je li ovakav unificirani pristup u skladu s postojećim načelima oblikovanja?).

```
public interface GraphicalObject {

        // Podrška za uređivanje objekta
        boolean isSelected();
        void setSelected(boolean selected);
        int getNumberOfHotPoints();
        Point getHotPoint(int index);
        void setHotPoint(int index, Point point);
        boolean isHotPointSelected(int index);
        void setHotPointSelected(int index, boolean selected);
        double getHotPointDistance(int index, Point mousePoint);

        // Geometrijska operacija nad oblikom
        void translate(Point delta);
        Rectangle getBoundingBox();
        double selectionDistance(Point mousePoint);

        // Podrška za crtanje (dio mosta)
        void render(Renderer r);

        // Observer za dojavu promjena modelu
        public void addGraphicalObjectListener(GraphicalObjectListener l);
        public void removeGraphicalObjectListener(GraphicalObjectListener l);

        // Podrška za prototip (alatna traka, stvaranje objekata u crtežu, ...)
        String getShapeName();
        GraphicalObject duplicate();
```

```
        // Podrška za snimanje i učitavanje
        public String getShapeID();
        public void load(Stack<GraphicalObject> stack, String data);
        public void save(List<String> rows);
}
```

Navedeni model oslanja se na pretpostavku da imate definirane i sljedeće razrede, koje dajemo u nastavku.

```
public class Point {

        private int x;
        private int y;

        public Point(int x, int y) {
                // ...
        }

        public int getX() {
                // ...
        }

        public int getY() {
                // ...
        }

        public Point translate(Point dp) {
                // vraća NOVU točku translatiranu za argument tj. THIS+DP...
        }

        public Point difference(Point p) {
                // vraća NOVU točku koja predstavlja razliku THIS-P...
        }
}

public class Rectangle {
        private int x;
        private int y;
        private int width;
        private int height;

        public Rectangle(int x, int y, int width, int height) {
                // ...
        };

        public int getX() {
                // ...
        }

        public int getY() {
                // ...
        }

        public int getWidth() {
                // ...
        }

        public int getHeight() {
                // ...
        }
}
```

Kako bi si olakšali pisanje ostatka koda, preporučamo da napravite i pomoćni razred `GeometryUtil` koji sadrži često korištene metode vezane u geometrijske proračune koje ćete trebati na različitim mjestima u kodu.

```
public class GeometryUtil {

        public static double distanceFromPoint(Point point1, Point point2) {
                // izračunaj euklidsku udaljenost između dvije točke ...
        }

        public static double distanceFromLineSegment(Point s, Point e, Point p) {
                // Izračunaj koliko je točka P udaljena od linijskog segmenta određenog
                // početnom točkom S i završnom točkom E. Uočite: ako je točka P iznad/ispod
                // tog segmenta, ova udaljenost je udaljenost okomice spuštene iz P na S-E.
                // Ako je točka P "prije" točke S ili "iza" točke E, udaljenost odgovara
                // udaljenosti od P do početne/konačne točke segmenta.
        }
}
```

Sučelje `GraphicalObject` predstavlja apstraktni model jednog grafičkog objekta. Sučelje predviđa da svaki grafički objekt bude subjekt čije stanje čine njegovi "hot-point"-i (pozicije te status selektiranosti) te njegov status selektiranosti. Zainteresirani promatrači moraju implementirati sučelje `GraphicalObjectListener` kako bi se mogli registrirati nad grafičkim objektom i dobivati obavijesti o njegovim promjenama. Ovo sučelje prikazano je u nastavku.

```
public interface GraphicalObjectListener {

        // Poziva se kad se nad objektom promjeni bio što...
        void graphicalObjectChanged(GraphicalObject go);
        // Poziva se isključivo ako je nad objektom promjenjen status selektiranosti
        // (baš objekta, ne njegovih hot-point-a).
        void graphicalObjectSelectionChanged(GraphicalObject go);

}
```

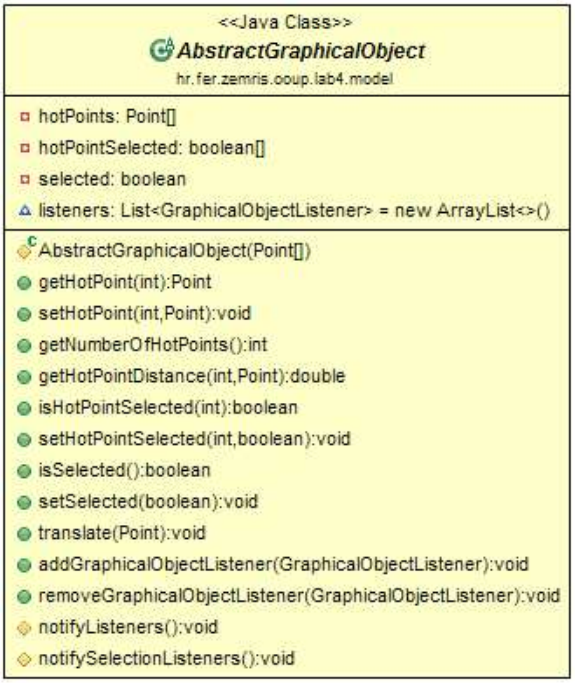
Konačno, za iscrtavanje grafičkog objekta predviđeno je sučelje `Renderer` koje je prikazano u nastavku.

```
public interface Renderer {
        void drawLine(Point s, Point e);
        void fillPolygon(Point[] points);
}
```

Zadatak

U okviru ove vježbe podržat ćemo dva grafička objekta: linijski segment te oval. Napravite sljedeće.

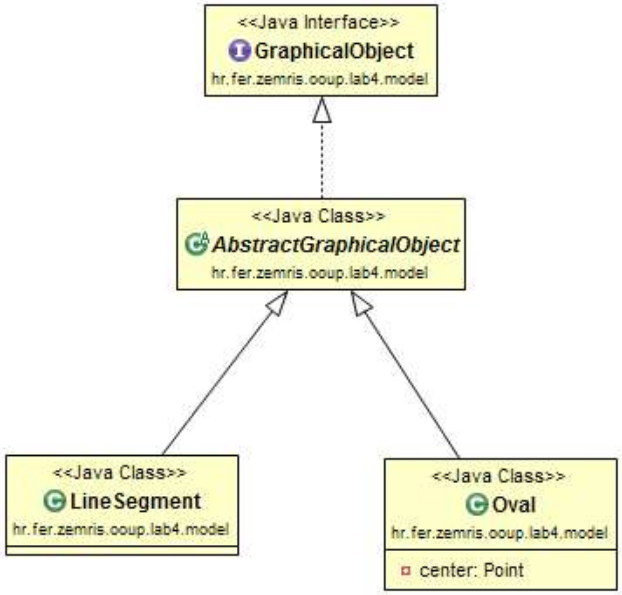
1. Prepišite sva prethodno opisana sučelja: `GraphicalObject`, `GraphicalObjectListener`, `Renderer`.Iz sučelja `GraphicalObject` za sada izostavite metode `render`, `getShapeID`, `load` te `save`.
2. Napišite i dovršite implementaciju razreda `Point`, `Rectangle` i `GeometryUtil`.
3. Napišite apstraktni razred `AbstractGraphicalObject` koji predstavlja djelomičnu implementaciju sučelja `GraphicalObject`.



   Razred implementira pohranu informacija o "hot-point"-ima (pozicija, status selektiranosti) te informaciju je li grafički objekt selektiran. Razred kroz konstruktor prima informaciju o broju i početnim položajima "hot-point"-a. U razredu je također implementirana funkcionalnost prijave i odjave promatrača i njihovog obavještavanja, kao i metode koje modificiraju "hot-point"-e te status selektiranosti i automatski obavještavaju promatrače.
4. Implementirajte razred `LineSegment` (izvedite ga iz razreda `AbstractGraphicalObject`). Opremite ga s dva konstruktora (jedan bez argumenata koji stvara linijski segment (0,0)-(10,0), te jedan koji prima početnu i konačnu točku). U njemu napišite metode koje niste mogli u apstraktnom nadrazredu: `selectionDistance`, `getBoundingBoxy`, `duplicate` (pazite: ne kopira se popis prijavljenih promatrača) i `getShapeName` (tako da vraća "Linija").
5. Implementirajte razred `Oval` (izvedite ga iz razreda `AbstractGraphicalObject`). Opremite ga s dva konstruktora (jedan bez argumenata koji stvara oval s desnim hot-pointom (10,0) i donjim hot-pointom (0,10), te jedan koji prima pozicije hot-pointa). U njemu napišite metode koje niste mogli u apstraktnom

nadrazredu: `selectionDistance`, `getBoundingBoxy`, `duplicate` (pazite: ne kopira se popis prijavljenih promatrača) i `getShapeName` (tako da vraća "Oval").

U ovom trenutku dijagram razreda grafičkih objekata (bez prikazanim članskih varijabli i metoda) izgledat će kao na slici u nastavku.



2. Model crteža

Čitav crtež modelirat ćemo razredom `DocumentModel` čije je okvirno sučelje prikazano u nastavku (slobodno doradite po potrebi).

```
public class DocumentModel {

        public final static double SELECTION_PROXIMITY = 10;

        // Kolekcija svih grafičkih objekata:
        private List objects = new ArrayList<>();
        // Read-Only proxy oko kolekcije grafičkih objekata:
        private List roObjects = Collections.unmodifiableList(objects);
        // Kolekcija prijavljenih promatrača:
        private List listeners = new ArrayList<>();
        // Kolekcija selektiranih objekata:
        private List selectedObjects = new ArrayList<>();
        // Read-Only proxy oko kolekcije selektiranih objekata:
        private List roSelectedObjects = Collections.unmodifiableList(selectedObjects);

        // Promatrač koji će biti registriran nad svim objektima crteža...
        private final GraphicalObjectListener goListener = new GraphicalObjectListener() {...};

        // Konstruktor...
        public DocumentModel() {...}

        // Brisanje svih objekata iz modela (pazite da se sve potrebno odregistrira)
        // i potom obavijeste svi promatrači modela
        public void clear() {...}

        // Dodavanje objekta u dokument (pazite je li već selektiran; registrirajte model kao promatrača)
        public void addGraphicalObject(GraphicalObject obj) {...}

        // Uklanjanje objekta iz dokumenta (pazite je li već selektiran; odregistrirajte model kao promatrača)
        public void removeGraphicalObject(GraphicalObject obj) {...}

        // Vrati nepromjenjivu listu postojećih objekata (izmjene smiju ići samo kroz metode modela)
        public List list() {...}

        // Prijava...
        public void addDocumentModelListener(DocumentModelListener l) {...}

        // Odjava...
        public void removeDocumentModelListener(DocumentModelListener l) {...}

        // Obavještavanje...
        public void notifyListeners() {...}
```

```
        // Vrati nepromjenjivu listu selektiranih objekata
        public List getSelectedObjects() {...}

        // Pomakni predani objekt u listi objekata na jedno mjesto kasnije...
        // Time će se on iscrtati kasnije (pa će time možda veći dio biti vidljiv)
        public void increaseZ(GraphicalObject go) {...}

        // Pomakni predani objekt u listi objekata na jedno mjesto ranije...
        public void decreaseZ(GraphicalObject go) {...}

        // Pronađi postoji li u modelu neki objekt koji klik na točku koja je
        // predana kao argument selektira i vrati ga ili vrati null. Točka selektira
        // objekt kojemu je najbliža uz uvjet da ta udaljenost nije veća od
        // SELECTION_PROXIMITY. Status selektiranosti objekta ova metoda NE dira.
        public GraphicalObject findSelectedGraphicalObject(Point mousePoint) {...}

        // Pronađi da li u predanom objektu predana točka miša selektira neki hot-point.
        // Točka miša selektira onaj hot-point objekta kojemu je najbliža uz uvjet da ta
        // udaljenost nije veća od SELECTION_PROXIMITY. Vraća se indeks hot-pointa
        // kojeg bi predana točka selektirala ili -1 ako takve nema. Status selekcije
        // se pri tome NE dira.
        public int findSelectedHotPoint(GraphicalObject object, Point mousePoint) {...}

}
```

Prokomentirajmo malo predloženi opis. Model dokumenta, tj. crteža, omogućava zainteresiranim klijentima da doznaju sve informacije o crtežu (koliko ima objekata, koji su, koji su od njih selektirani i slično). Također, model mora omogućiti svim zainteresiranim klijentima da budu obaviješteni i kada se u model dodaju novi odnosno uklone postojeći grafički objekti. Stoga je u okviru ove vježbe predložena uporaba ulančanih promatrača:

- Svaki grafički objekt je subjekt koji klijentima omogućava dojavu promjena nad njime.
- `DocumentModel` je subjekt koji svojim klijentima omogućava dojavu informacija o dodavanju i uklanjanju grafičkih objekata te informacija o promjenama u samim grafičkim objektima. Kako bi to bilo moguće, sam `DocumentModel` prijavit će se kao promatrač nad svakim grafičkim objektom koji mu pripada, i u situacijama kada ga grafički objekt obavijesti da je došlo do promjene u grafičkom objektu, `DocumentModel` će o tome obavijestiti svoje promatrače. Na ovaj način osigurano je da je dovoljno da se platno za crtanje prijavi samo na `DocumentModel`.
- Jedna od usluga koje `DocumentModel` treba ponuditi svojim klijentima je dostava kolekcije postojećih grafičkih objekata koji pripadaju crtežu. Pri tome treba osigurati da pri manipuliranju tom kolekcijom klijent ne može modificirati stanje crteža. To se može osigurati tako da se klijentu svaki puta vrati nova kopija liste što će vrlo skupo i neefikasno rješenje. Stoga je u predloženom rješenju stvoren proxy objekt koji omata originalnu listu i onemogućava bilo koju operaciju koja bi mijenjala sadržaj liste.
- Još jedna od usluga koju bi `DocumentModel` treba ponuditi jest dohvat popisa selektiranih objekata. I opet, jedna moguća implementacija bila bi da se pri svakom pozivu te metode pretražuje cjelokupna kolekcija objekata i provjerava status selektiranosti. U okviru ove vježbe predloženo je rješenje u kojem sam `DocumentModel` čitavo vrijeme održava kolekciju selektiranih objekata (`DocumentModel` je promatrač svih objekata pa kad se god promjeni status selektiranosti nekog od objekata `DocumentModel` ažurira svoju kolekciju) a pozivateljima vraća read-only proxy na tu listu.

Sučelje promatrača dokumenta prikazano je u nastavku.

```
public interface DocumentModelListener {

        void documentChange();

}
```

Zadatak

Prepišite definiciju sučelja `DocumentModelListener`. Napišite cjelovitu implementaciju razreda `DocumentModel`.

3. Metoda `main`, glavni program i crtanje objekata

Program za crtanje želimo napisati na način koji će osigurati da jednom napisani program može transparentno raditi s proizvoljnim grafičkim objektima, a bez da se u tom dijelu koda mora mijenjati ijedan redak koda. Stoga želimo da metoda `main` konceptualno izgleda kako je navedeno u nastavku:

```
void main(...) {

        List objects = new ArrayList<>();

        objects.add(new LineSegment());
        objects.add(new Oval());
```

```
        GUI gui = new GUI(objects);
        gui.setVisible(true);
}
```

Glavni prozor programa modeliran je razredom `GUI` koji u konstruktoru dobiva listu koja se sastoji od po jednog primjerka grafičkih objekata s kojima će se moći graditi crtež. Potom se u konstruktoru prozora ta lista pamti, i za svaki objekt liste stvara jedan gumb u alatnoj traci (tekst koji se ispisuje u gumbu odgovara onome što vrati metoda `getName()` grafičkog objekta. Prozor također definira jedan primjerak modela dokumenta te primjerak platna za crtanje. Platno za crtanje u konstruktoru treba dobiti referencu na model dokumenta koji je stvorio prozor.

U platnu za crtanje potrebno je nadjačati metodu koja se poziva kada je potrebno nacrtati površinu te komponente. Primjerice, u Javi bi to bila metoda `paintComponent(Graphics g)`. Zadaća te metode jest crtanje crteža. Da bismo to omogućili, dodajte sada u `GraphicalObject` metodu `void render(Renderer r);` (bila je prikazana u sučelju na početku ove upute). U svakom konkretnom grafičkom objektu (`LineSegment`, `Oval`) napišite njezinu implementaciju. Pazite: jedino što Vam stoji na raspolaganju su dva primitiva koja smo deklarirali u sučelju `Renderer`.

Napišite implementaciju sučelja `Renderer` koja crtanje obavlja na površini grafičke komponente. Primjerice, u Javi bi takvu implementacija okvirno bila sljedećeg oblika.

```
public class G2DRendererImpl implements Renderer {

        private Graphics2D g2d;

        public G2DRendererImpl(Graphics2D g2d) {
                // ...
        }

        @Override
        public void drawLine(Point s, Point e) {
                // Postavi boju na plavu
                // Nacrtaj linijski segment od S do E
                // (sve to uporabom g2d dobivenog u konstruktoru)
        }

        @Override
        public void fillPolygon(Point[] points) {
                // Postavi boju na plavu
                // Popuni poligon definiran danim točkama
                // Postavi boju na crvenu
                // Nacrtaj rub poligona definiranog danim točkama
                // (sve to uporabom g2d dobivenog u konstruktoru)
        }

}
```

Sada bismo u platnu za crtanje metodu koja crta površinu komponente mogli napisati na sljedeći način.

```
void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D)g;
        Renderer r = new G2DRendererImpl(g2d);
        za svaki objekt o modela:
          o.render(r);
}
```

Da biste isprobali radi li napisani kod, dodajte (privremeno) u konstruktoru prozora kod koji u stvoreni model ručno doda nekoliko objekata: ti bi se objekti morali ispravno prikazati u platnu za crtanje. Kad se uvjerite da to radi, uklonite taj dio koda tako da model po stvaranju bude prazan.

4. Modeliranje stanja u programu za crtanje

Sada kada imamo izgrađen elementarni dio funkcionalnosti prikaza objekata, vrijeme je da program obogatimo različitim alatima. Primjerice, htjeli bismo alat koji dodaje linije, alat koji dodaje ovale, alat koji omogućava selekciju i pomicanje objekata, njihovo grupiranje i promjenu Z-poretka te alat za brisanje. Što će se točno u programu dogoditi kada korisnik klikne mišem ili pritisne tipku na tipkovnici ovisit će o odabranom alatu. Stoga ćemo ovaj dio koda oblikovati uporabom oblikovnog obrasca Stanje.

Neka je apstraktno stanje definirano sučeljem `State`, koje je dano u nastavku.

```
public interface State {
        // poziva se kad progam registrira da je pritisnuta lijeva tipka miša
```

```
        void mouseDown(Point mousePoint, boolean shiftDown, boolean ctrlDown);
        // poziva se kad progam registrira da je otpuštena lijeva tipka miša
        void mouseUp(Point mousePoint, boolean shiftDown, boolean ctrlDown);
        // poziva se kad progam registrira da korisnik pomiče miš dok je tipka pritisnuta
        void mouseDragged(Point mousePoint);
        // poziva se kad progam registrira da je korisnik pritisnuo tipku na tipkovnici
        void keyPressed(int keyCode);

        // Poziva se nakon što je platno nacrtalo grafički objekt predan kao argument
        void afterDraw(Renderer r, GraphicalObject go);
        // Poziva se nakon što je platno nacrtalo čitav crtež
        void afterDraw(Renderer r);

        // Poziva se kada program napušta ovo stanje kako bi prešlo u neko drugo
        void onLeaving();
}
```

Napišite razred `IdleState` koji je implementacija ovog sučelja i u kojem su sve metode prazne.

Potom promijenite glavni prozor: dodajte mu člansku varijablu `private State currentState;` koja inicijalno pokazuje na primjerak razreda `IdleState`; u to isto stanje aplikacija se treba vratiti svaki puta kada korisnik pritisne tipku ESC.

Dodajte u platno za crtanje potreban kod kojim će platno tražiti dojavu informacija o pritiscima tipaka i pomacima miša. Na svaku takvu dojavu platno će samo pozvati odgovarajuću metodu trenutnog stanja (osigurajte da platno za crtanje vidi koje je trenutno stanje).

Vratite se u metodu platna koja crta grafičke objekte i nakon crtanja svakog objekta pozovite još i odgovarajuću metodu stanja te isto napravite i nakon što je nacrtan čitav crtež.

5. Dodavanje grafičkih objekata u crtež

Dodavanje grafičkih objekata u crtež riješit ćemo implementacijom stanja `AddShapeState`. Skica razreda ovog stanja dana je u nastavku.

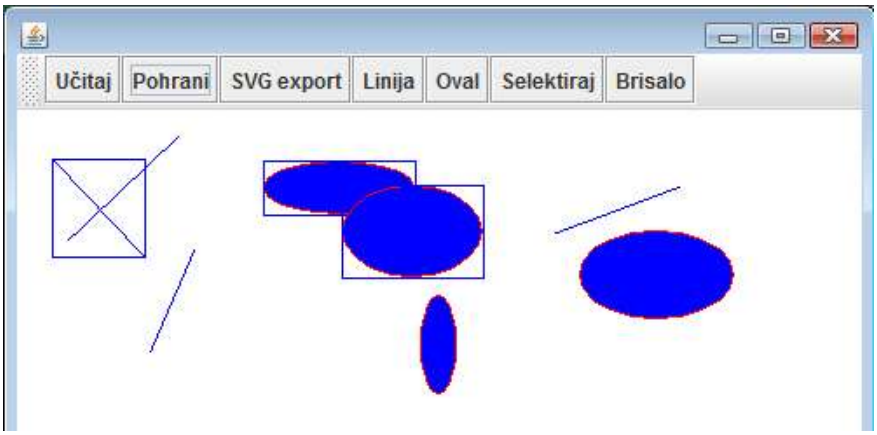
```
public class AddShapeState implements State {

        private GraphicalObject prototype;
        private DocumentModel model;

        public AddShapeState(DocumentModel model, GraphicalObject prototype) {
                // ...
        }

        @Override
        public void mouseDown(Point mousePoint, boolean shiftDown, boolean ctrlDown) {
                // dupliciraj zapamćeni prototip, pomakni ga na poziciju miša i dodaj u model
        }

        // ...
}
```

Dodavanje objekata u programu zamišljeno je po analogiji sa štambiljanjem: korisnik odabere s kojim grafičkim objektom želi raditi i onda na svaki klik miša na platnu na tom mjestu nastane jedan takav lik pretpostavljenih dimenzija.

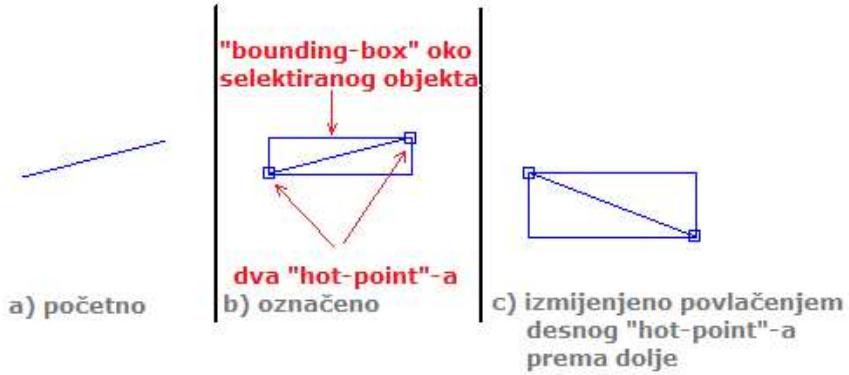
Sada se vratite u konstruktor prozora na mjesto gdje ste za svaki primljeni grafički objekt dodavali u alatnu traku po jedan gumb: modificirajte kod tako da se pritiskom na taj gumb promjeni trenutno stanje programa u primjerak stanja `AddShapeState` koje u konstruktoru dobije referencu na objekt koji gumb predstavlja.

6. Selektiranje objekata

Definirajte novo stanje: `SelectShapeState`. U tom stanju korisnik može mišem selektirati objekte. Ako drži pritisnutu tipku CTRL, objekti se dodaju u selekciju dok se bez pritisnute tipke CTRL uvijek selektira samo jedan objekt (ako je neki drugi bio selektiran, on se automatski treba odselektirati). Primjer koji prikazuje tri selektirana objekta prikazan je u nastavku.



Oko svakog selektiranog objekta prikazuje se njegov bounding-box. Kako platno za crtanje o ovome ništa ne zna, ovaj dio crtanja obavlja samo stanje u metodi `afterDraw` za svaki selektirani objekt. Dodatno, ako je selektiran samo jedan objekt, onda se za njega malim kvadratićima prikazuju i njegovi "hot-point"-i -- evo ponovno primjera s linijskim segmentom.

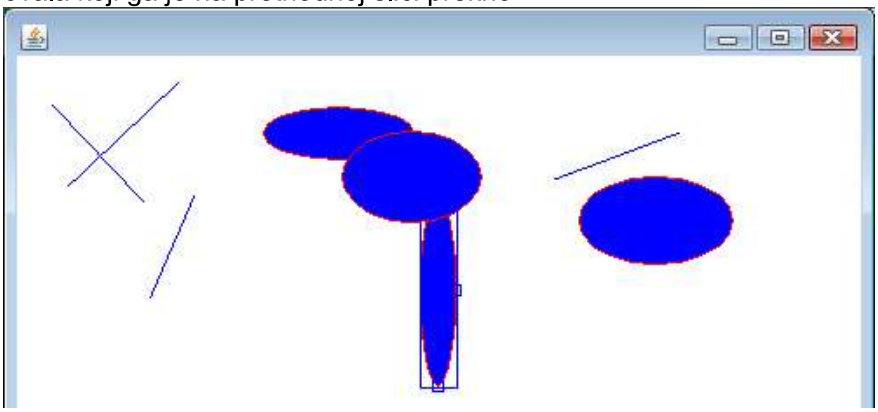


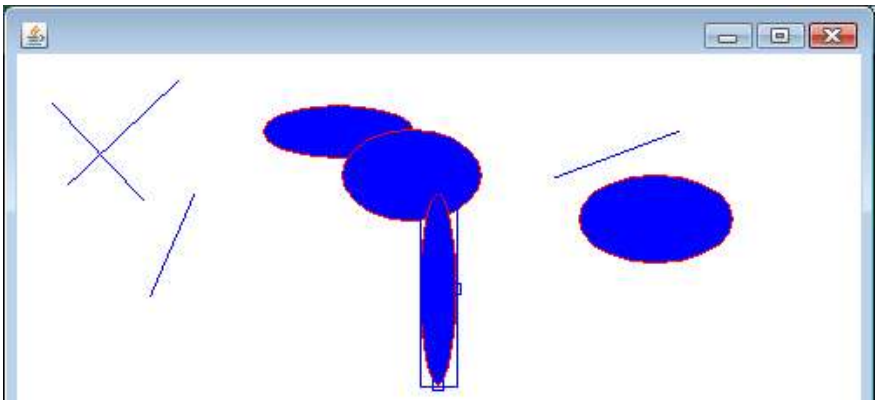
Samo u tom slučaju (selektiran jedan objekt) klik i povlačenje miša ne rade novu selekciju već pomiču koordinate selektiranog "hot-point"-a čime se modificira trenutni objekt.

U ovom stanju potrebno je implementirati i sljedeća djelovanja pritisaka tipki na tipkovnici.

- Kursorske tipke gore, dolje, lijevo, desno pomiču sve selektirane objekte za jedan piksel u odabranom smjeru.
- Tipka `+` pomiče objekt bliže prema promatraču (mijenja mu Z-poredak).
- Tipka `-` pomiče objekt dalje od promatraču (mijenja mu Z-poredak).

Utjecaj Z-poretka prikazan je na sljedeće dvije slike. Na prvoj slici selektirani objekt se crta prije drugog ovala pa ga on prekrije; na drugoj slici selektirani se objekt crta kasnije (povećan mu je Z-poredak) na se on crta preko ovala koji ga je na prethodnoj slici prekrio.





Napuštanjem ovog stanja automatski treba desektirati sve selektirane objekte.

Dodajte u alatnu traku gumb "Selektiraj" koji program prebacuje u stanje za selekciju.

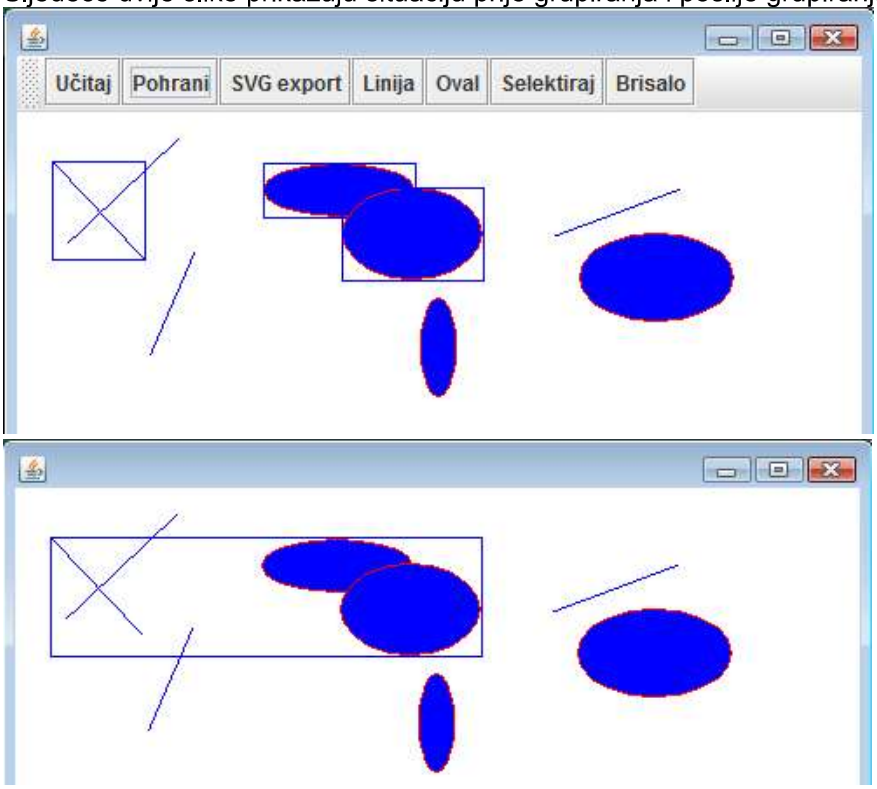
7. Grupiranje objekata

Napišite još jednu implementaciju grafičkog objekta: `CompositeShape`. To je objekt koji nema "hot-point"-a: on je kompozit čija su djeca drugi grafički objekti (potencijalno i drugi kompoziti). Pazite kako ćete u njemu implementirati metode poput one za izračun bounding-box-a (hint: unija) te pomicanja i crtanja (hint: delegiranje).

Omogućite sada u stanju za selekciju još dvije tipke.

- Pritisak na tipku G iz modela briše sve selektirane objekte, stvara novi kompozit čija to postaju djeca i samo taj kompozit ubacuje u model.
- Pritisak na tipku U (samo u slučaju da postoji jedan selektirani objekt i da je on kompozit) iz modela briše taj kompozit i ponovno dodaje u model njegovu djecu (i odmah ih ostavlja selektiranima tako da bi pritisak na G ponovno sve natrag zapakirao).

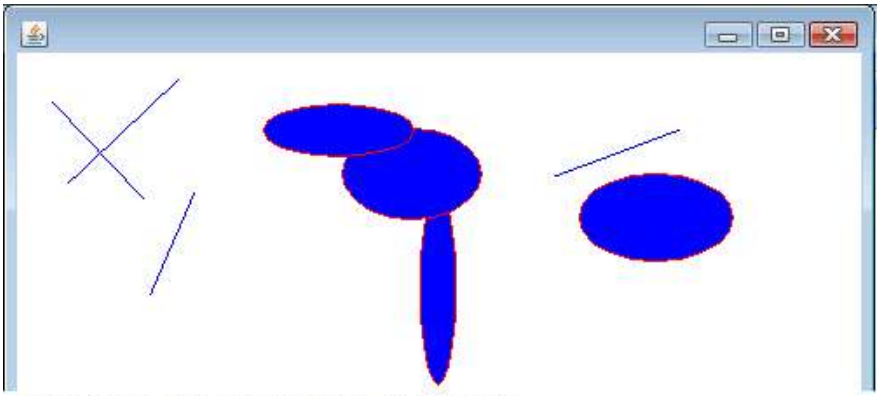
Sljedeće dvije slike prikazuju situaciju prije grupiranja i poslije grupiranja.



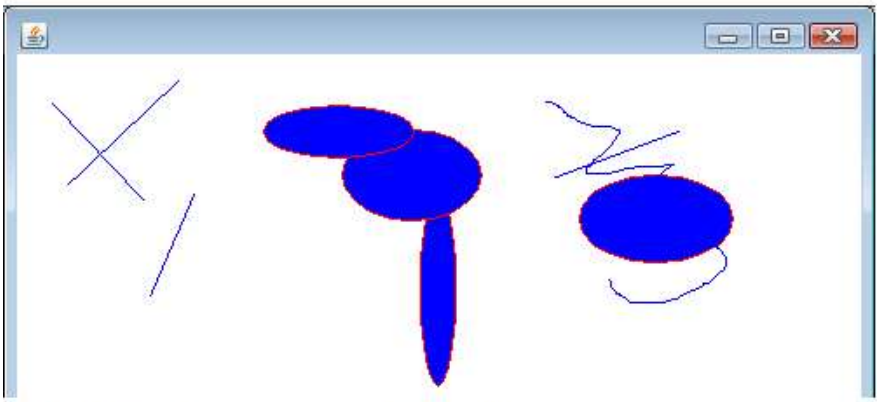
8. Brisanje objekata

Dodajte implementaciju novog stanja `EraserState` (i odgovarajući gumb u alatnu traku) koje će korisniku omogućiti provođenje brisanja objekata. Ideja je omogućiti korisniku da na klik miša opiše (i vizualizira) krivulju koju će na otpuštanje miša presjeći sa svim postojećim objektima. Svaki objekt koji nacrtana krivulja siječe bit

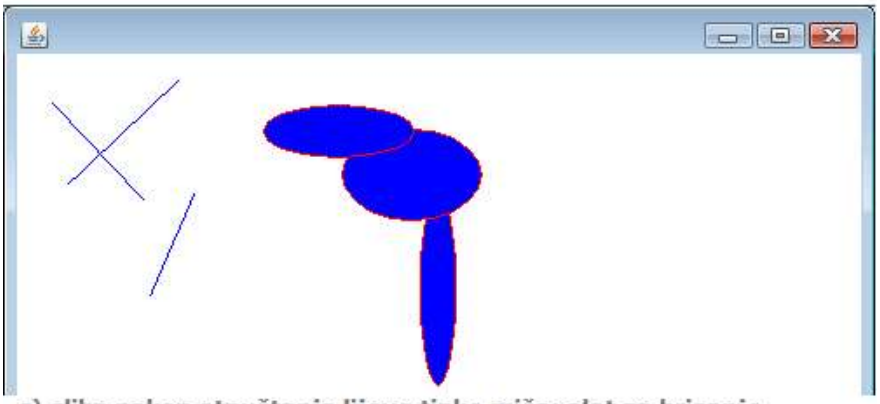
će obrisan. Slika u nastavku ovo ilustrira.



a) početna slika (ooup-lab4-slika2.txt)



b) trag miša nastao povlačenjem u alatu za brisanje



c) slika nakon otpuštanja lijeve tipke miša: alat za brisanje uklonio je sve objekte preko kojih je prešao trag miša

9. Export u SVG

Dodajte u alatnu traku gumb "SVG Export". Pritiskom na taj gumb korisnika će se pitati da odabere direktorij i ime datoteke koju želi stvoriti (*Save dialog*), i potom treba generirati SVG opis slike. Da biste to napravili, stvorite novu implementaciju renderera (pseudokod prikazan u nastavku).

```
public class SVGRendererImpl implements Renderer {

        private List lines = new ArrayList<>();
        private String fileName;

        public SVGRendererImpl(String fileName) {
                // zapamti fileName; u lines dodaj zaglavlje SVG dokumenta:
                // <svg xmlns=... >
                // ...
        }

        public void close() throws IOException {
                // u lines još dodaj završni tag SVG dokumenta: </svg>
                // sve retke u listi lines zapiši na disk u datoteku
                // ...
        }

        @Override
        public void drawLine(Point s, Point e) {
```

```
                // Dodaj u lines redak koji definira linijski segment:
                // <line ... />
        }

        @Override
        public void fillPolygon(Point[] points) {
                // Dodaj u lines redak koji definira popunjeni poligon:
                // <polygon points="..." style="stroke: ...; fill: ...;" />
        }

}
```

Na pritisak gumba "SVG export" događa se sljedeće:

```
fileName = pitajIme();
SVGRendererImpl r = new SVGRendererImpl(fileName);
za svaki objekt o modela:
  o.render(r);
r.close();
```

10. Dodavanje podrške za učitavanje i snimanje crteža

10.1. Dodavanje podrške za snimanje crteža

Nativni format datoteke za snimanje crteža u našem je slučaju niz redaka teksta. Pri tome svaki objekt zauzima jedan redak (uključivo i kompozite, koji se međutim mogu pozvati na prethodne retke). Da bismo pojasnili, zamislimo crtež koji se sastoji od objekta A, kompozita koji sadrži objekte B i C te još jednog objekta D. Retci u datoteci bili bi sljedeći:

```
A
B
C
kompozit - 2 prethodna
D
```

Svaki redak datoteke započinje identifikatorom objekta: @LINE ili @OVAL ili @COMP (za kompozit). Ovaj string odgovara identifikatoru vrste grafičkog objekta. Vratite se sada u sučelje `GraphicalObject` i dodajte metodu `public String getShapeID();`; implementirajte je u svakom od triju likova s kojima radimo da vraća prikladan identifikator. U retku dalje slijedi razmak pa niz argumenata koje "razumije" sam geometrijski oblik.

1. Ako je lik linijski segment, slijede x pa y početne točke i x pa y konačne točke.
2. Ako je lik oval, slijede x pa y desnog "hot-point"-a te x pa y donjeg "hot-point"-a.
3. Ako je lik kompozit, slijedi broj neposredno prethodno dostupnih objekata koji čine njegovu djecu.

Vratite se sada u sučelje `GraphicalObject` i dodajte metodu `public void save(List<String> rows);`; implementirajte je u svakom od triju likova s kojima radimo tako da jedan redak (linijski segment, oval) odnosno više redaka (kompozit) dodaju na kraj primljene liste `rows`.

Dodajte sada u alatnu trak gumb "Pohrani". Klikom na taj gumb treba pitati korisnika u koju datoteku da pohrani crtež, treba stvoriti praznu listu redaka, tražiti svaki objekt u modelu da se pohrani u tu listu i potom listu treba pohraniti u datoteku. Primjeri dviju datoteka kao i njihov grafički prikaz dostupni su na kraju ove upute.

10.2. Dodavanje podrške za učitavanje crteža

Dodajte u alatnu trak gumb "Učitaj". Klikom na taj gumb treba pitati korisnika iz koje datoteke se radi učitavanje. Retke odabrane datoteke treba učitajte u pomoćnu listu redaka. Stvorite prazan stog grafičkih objekata. Pripremite pomoćnu mapu koja identifikatore grafičkih objekata mapira na njihove prototipe.

Vratite se sada u sučelje `GraphicalObject` i dodajte metodu `public void load(Stack<GraphicalObject> stack, String data);`. Vratite se u linijski segment i oval i implementirajte ovu metodu tako da stvori **NOVI** grafički objekt s parametrima koje pročita iz drugog argumenta (`String data` - to će biti ostatak retka iz datoteke nakon uklanjanja početnog identifikatora). Taj novostvoreni objekt metoda na kraju izvođenja mora gurnuti na vrh primljenog stoga.

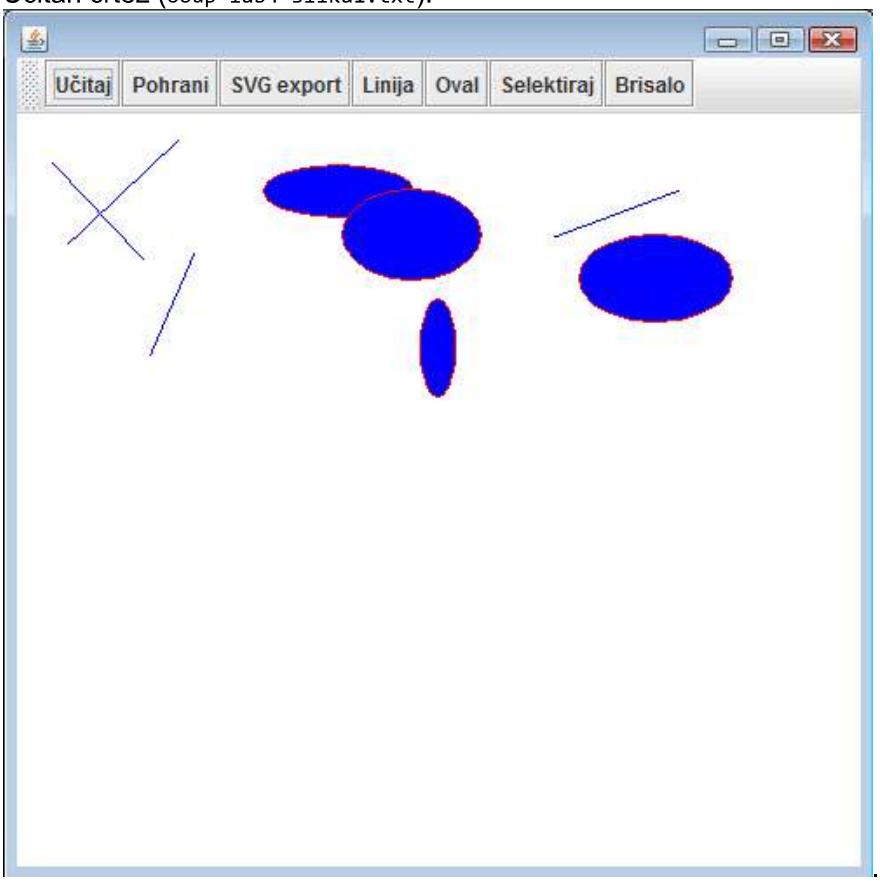
Implementirajte isto i u kompozitu: metoda mora iz stringa `data` pogledati koliko prethodno-stvorenih objekata na stogu čini djecu kompozita, mora stvoriti NOVI kompozit, sa stoga skinuti utvrđeni broj objekata i postaviti ih kao djecu novostvorenog kompozita, i potom taj kompozit treba gurnuti na stog.

Sada možemo definirati kako se obavlja nastavak učitavanja. Nakon što smo u pomoćnu listu pročitali sve retke datoteke, iteriramo redak po redak. Za svaki redak izvučemo prvi dio (identifikator) i u mapi pronađemo prototip

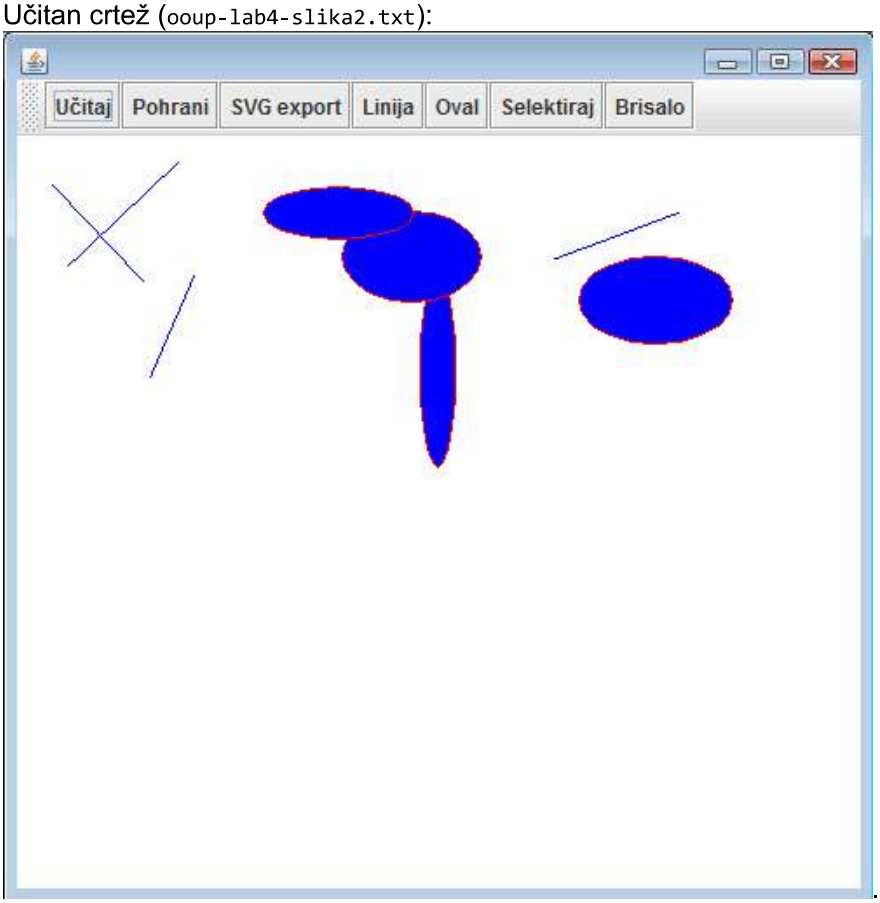
koji mu odgovara; nad tim prototipom pozovemo `load(...)` koji će rezultirati time da se odgovarajući objekt stavi na stog (uz moguće prethodno uklanjanje nekih drugih objekata ako smo učitali kompozit). Stanje koje na kraju ostane na stogu predstavlja popis objekata koje dodajemo u model. I time je učitavanje gotovo.

11. Primjeri datoteka

1. Snimljen crtež: **ooup-lab4-slika1.txt**.
2. Učitan crtež (`ooup-lab4-slika1.txt`):



3. Snimljen crtež: **ooup-lab4-slika2.txt**.
4. Učitan crtež (`ooup-lab4-slika2.txt`):



5. SVG renderiranje drugog crteža: **ooup-lab4-slika2.svg**.
