Software Engineering 2
DEAD REPORT

Deadline Report

Team number:	0309

Team member 1	
Name:	Muhammed Akinci
Student ID:	11720479
E-mail address:	a11720479@unet.univie.ac.at

Team member 2	
Name:	Maxim Bogoutdinov
Student ID:	01468382
E-mail address:	a01468382@unet.univie.ac.at

Team member 3	
Name:	Ema Dupovac-Kilincarslan
Student ID:	01268309
E-mail address:	a01268309@unet.univie.ac.at

Team member 4	
Name:	Lala Mammadova
Student ID:	01652485
E-mail address:	a01652485@unet.univie.ac.at

1 Final Design

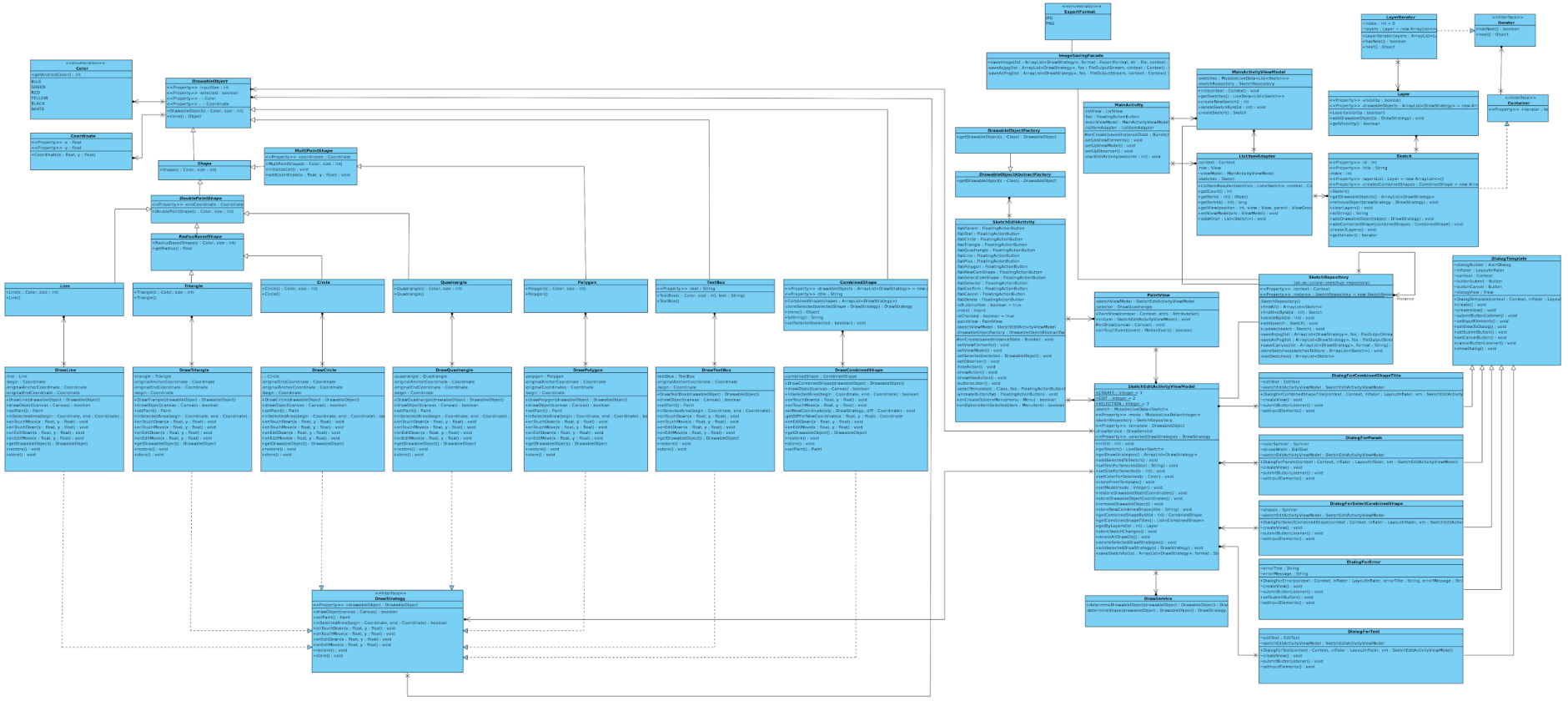
1.1 Design Approach and Overview

In our first online meeting we discussed the assignment sheet and talked about possible solutions for the project. It was a long discussion about the assignment sheet and understanding the requirements set there. So we used brainstorming as a technique to collect different ideas about functionalities and layout implementations that had to be done.

By next time we had a first version of the UML class diagram, which contained the basic logic and elements we had to implement. In the first iteration we just did the basic skeleton of our implementation where we defined classes with basic shapes. At this stage we decided to start coding and expand our project by actually working on it. So we divided the shapes that had to be implemented and started working on them. We always had regular meetings where we discussed our ideas and merged them. And we were parallel also working on our UML class diagram and expanding them.

Ater we got the feedback for the supd part and figured out that our approach to solve the project was ok, we decided not to make major changes in our approach and continue with the implementation as before. So in our next meeting we discussed possibilities how to incorporate all the other missing patterns in our implementation and finish the functional requirement that we were missing. So this time we divided the functional requirements that were left to implement and started working on them. We met regularly to present the progress and get the ideas by brainstorming them together. In the end we incorporated the patterns in our implementation and started doing tests and the needed documentation for the project.

1.1.1 Class Diagrams



Due to the amount of the classes this is just a thumbnail of the class diagram. For a readable version, please check the SVG file in the same gitlab folder.

1.1.2 Technology Stack

Besides the standard libraries (Appcompat, Test...) that were included in our project, we used following frameworks for our implementation:

- **Android Jetpack suite** - we use a pack of standard libraries for creating main components of application like ViewModel, Canvas, Paint, Activity.
- **Material components available through Google Maven Repository** - which we used for the layout and specifying the main design. The main UI, where the user can draw, contains a blank canvas with a central floating action button. This floating action button is the primary point of interaction where User can open the menu and use other floating action points to start actions like drawing lines or writing text and also modifying the shapes by changing attributes.

  We decided to use a floating action button as a modern approach to design our app and to have a different layout than the most used and popular paint apps have.

- **Lifecycle-aware components available through Google Maven Repository**

  We used the lifecycle class to process the information about the lifecycle state of the components like sketch itself, which holds all the elements.

- **JUnit**

  We used JUnit framework for the tests provided for the quality requirements.

1.2 Major Changes Compared to SUPD

We added five more patterns since the SUPD and have now a total of seven design patterns. We added Composite pattern for the CombinedShape, Iterator pattern for our Layer class, Template Method pattern for our Alert Dialogs, the Abstract Factory pattern for our Shape determination and the facade pattern for our image saving process next to our already existing Strategy pattern and Factory pattern. A more detailed information about the pattern follows in the next chapter.

According to the SUPD it was only possible to add Sketches but not delete, save or load them from the internal android storage. Now it is possible to do these kinds of operations. We added the concept of layers, which allows the user to select one of three layers to draw on them. It is really easy to hide layers as well. The variety of drawable objects increased. Now the user can add polygons and combined shapes. Operations such as selecting multiple drawable objects, changing their attributes and its coordinates are also possible. The user can now group drawable objects and create a combined shape out of them. The canvas can easily be exported as a PNG or JPEG file.

1.3 Design Patterns

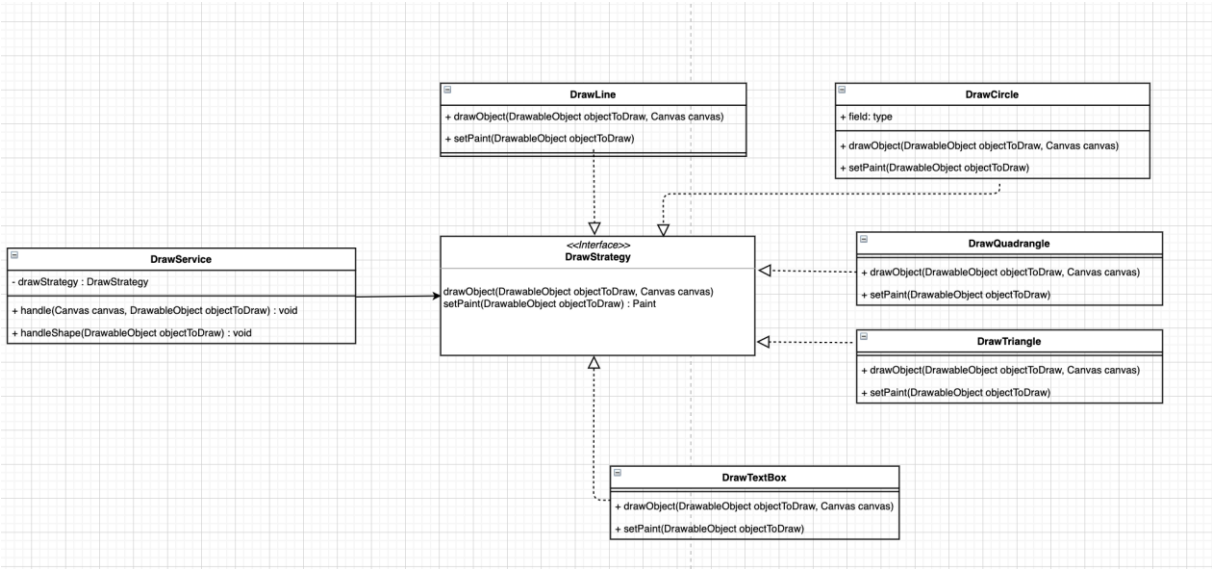
As the main application pattern, we decided to follow "Guide to app architecture" by Google and implement MVVM pattern with the LiveData library. It allows us to separate UI, business logic, data from each other. As an implementation for each Activity was created a separate ViewModel class which manages all logic/connection between UI and entities.

Also, we implemented Strategy and Factory patterns.

1.3.1 Strategy Pattern

We used the strategy pattern to implement drawing techniques in Canvas and draw such objects as textbox, circle, linie, triangle and rectangle. This helps us to extend drawing logic into a separate service where we can manage different drawing algorithms.

The main benefit of using a strategy pattern here is that we can easily create new drawing algorithms when adding new DrawableObjects without changing the main View.



- DrawService defines a type of object to draw and create related algorithms.

```
/**
 * Determines the DrawStrategy of the DrawableObject
 * @param drawableObject A DrawableObject, which generates its DrawStrategy
 * @return Returns a DrawStrategy that matches the DrawableObject
 * @throws CloneNotSupportedException The determination fails, CloneNotSupportedException will be thrown
 */
public DrawStrategy determineDrawStrategy(DrawableObject drawableObject) throws CloneNotSupportedException {
    DrawStrategy result = null;
    if (drawableObject instanceof TextBox)
        result = new DrawTextBox(drawableObject);
    else if (drawableObject instanceof Shape)
        result = determineFromShape(drawableObject);
    else if (drawableObject instanceof CombinedShape)
        result = new DrawCombinedShape(drawableObject);
    return result;
}
```

- Interface describe main functionality

```
public interface DrawStrategy {

    /**
     * Draws the DrawableObject of the DrawStrategy to the given Canvas
     * @param canvas Canvas in which the DrawableObject is drawn
     * @return Returns true if the drawing was successful
     */
    boolean drawObject(Canvas canvas);

    /**
     * Sets the Paint of the DrawStrategy
     * @return Returns the Paint of the DrawStrategy
     */
    Paint setPaint();

    /**
     * Checks if the DrawStrategy is in the selected area
     * @param begin Begin Coordinate of the selector
     * @param end End Coordinate of the selector
     * @return Returns the configured Paint
     */
    boolean inSelectedArea(Coordinate begin, Coordinate end);

    /**
     * First TouchDown of the created DrawStrategy
     * @param x X-Coordinate of Touch Down
     * @param y Y-Coordinate of Touch Down
     */
    void onTouchDown(float x, float y);
```

- And here a given example of drawing a circle in the DrawCircle class that implements the DrawStrategy interface.

```
@Override
public boolean drawObject(Canvas canvas) {
    canvas.drawCircle(
            this.circle.getAnchorCoordinate().getX(),
            this.circle.getAnchorCoordinate().getY(),
            this.circle.getRadius(),
            setPaint()
    );
    return true;
}

@Override
public Paint setPaint() {
    Paint paint = new Paint();
    paint.setColor(this.circle.getColor().getAndroidColor());
    paint.setAntiAlias(true);
    paint.setStyle(Paint.Style.STROKE);
    paint.setStrokeWidth(this.circle.getInputSize());
    if (this.circle.isSelected())
        paint.setPathEffect(new DashPathEffect(new float[]{2, 4}, phase: 50));
    return paint;
}
```

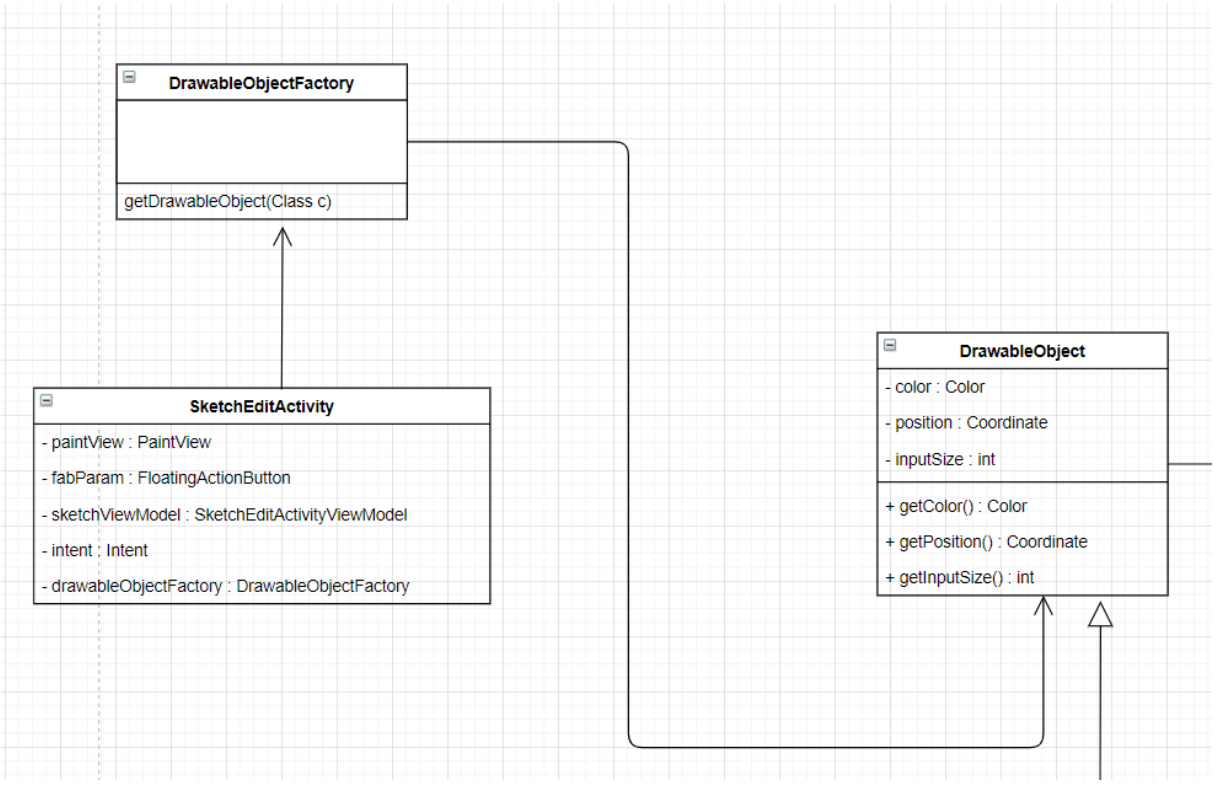
```
@Override
public boolean inSelectedArea(Coordinate begin, Coordinate end) {
    float beginCircleX = this.circle.getAnchorCoordinate().getX();
    float beginCircleY = this.circle.getAnchorCoordinate().getY();
    float beginX = Math.min(end.getX(), begin.getX());
    float beginY = Math.min(end.getY(), begin.getY());
    float endX = Math.max(end.getX(), begin.getX());
    float endY = Math.max(end.getY(), begin.getY());
    float radius = this.circle.getRadius();

    return (beginCircleX - radius > beginX && beginCircleY - radius > beginY &&
            beginCircleX + radius < endX && beginCircleY + radius < endY);
}

@Override
public void onTouchDown(float x, float y) {
    this.circle.setAnchorCoordinate(new Coordinate(x, y));
    this.originalAnchorCoordinate = this.circle.getAnchorCoordinate();
}
```

1.3.2 Factory Pattern

Factory Pattern being one of the most famous design patterns, is also implemented in our project and being the creational pattern it helps us to create an object in its best way.



- We are creating the object without exposing the creation logic to the client and refer to the newly created object. Here in the snapshot we create a DrawableObjectFactory class to generate objects of concrete class based on given information.

```
public class DrawableObjectFactory extends DrawableObjectAbstractFactory{
    public DrawableObject getDrawableObject(Class c) {
        if (c == Line.class) return new Line();
        else if (c == Circle.class) return new Circle();
        else if (c == Quadrangle.class) return new Quadrangle();
        else if (c == Triangle.class) return new Triangle();
        else if (c == Polygon.class) return new Polygon();
        else if (c == TextBox.class) return new TextBox();
        return new TextBox();
    }
}
```

- In this method called buttonListener() we get and create the object of the concrete classes, which are extending the abstract Shape class: Line, Circle, Quadrangle, Triangle by passing information of the Class Type through the DrawableObjectFactory method.

```
private void buttonsLister() {
    fabText.setOnClickListener(view -> {
        selectTemplate(TextBox.class, (FloatingActionButton) view);
        DialogForText dialog = new DialogForText( context: this, getLayoutInflater(), sketchViewModel);
        dialog.create();
    });

    fabCircle.setOnClickListener(view -> selectTemplate(Circle.class, (FloatingActionButton) view));

    fabTriangle.setOnClickListener(view -> selectTemplate(Triangle.class, (FloatingActionButton) view));

    fabQuadrangle.setOnClickListener(view -> selectTemplate(Quadrangle.class, (FloatingActionButton) view));

    fabLine.setOnClickListener(view -> selectTemplate(Line.class, (FloatingActionButton) view));

    fabPolygon.setOnClickListener(view -> selectTemplate(Polygon.class, (FloatingActionButton) view));
```

1.3.1 Iterator Pattern

For implementing Iterator pattern was created an Interface called Iterator, which has basic methods next() and hasnext().

```
package at.ac.univie.sketchup.model;

public interface Iterator {

    public boolean hasNext();

    public Object next();
}
```

```
package at.ac.univie.sketchup.model;

public interface Container {

    public Iterator getIterator();

}
```

Further, was created the class LayerIterator, which implements the Iterator pattern and overrides its methods.These methods were used in Sketch, which is a concrete class, that implements Container interface,by addDrawableObject method, where iterator pattern was used to iterate through the Layerlist in each Sketch.

```java
package at.ac.univie.sketchup.model;

import java.util.ArrayList;

public class LayerIterator implements Iterator {
    int index = 0;
    ArrayList<Layer> layers = new ArrayList<>();

    public LayerIterator(ArrayList<Layer> layers) {
        this.layers = layers;
    }

    @Override
    public boolean hasNext() {
        return index < layers.size();
    }

    @Override
    public Object next() {
        if (this.hasNext()) {
            return layers.get(index++);
        }
        return null;
    }
}
```

```java
public void addDrawableObject(DrawStrategy object) {
    Layer lastVisible = null;
    Layer layerZero = new Layer( visibility: true);
    LayerIterator layerIterator = (LayerIterator) getIterator();
    while (layerIterator.hasNext()) {
        Layer l = (Layer) layerIterator.next();
        if (l.getVisibility())
            lastVisible = l;
    }
    if (lastVisible != null)
        lastVisible.addDrawableObject(object);
    else
        layerZero.addDrawableObject(object);
}
```

1.3.2 Abstract Factory Pattern

Abstract Factory Pattern is being used to manage all the other factory patterns and has a role of a major factory pattern to handle other factories in the proposed way. Since we only implemented the Factory Pattern to create our objects as one factory (as described in 1.3.2. Factory Pattern) we decided to use Abstract Factory Pattern to manage our existing Factory Pattern.

```
public abstract class DrawableObjectAbstractFactory{

    //Calling the DrawableObjectFactory method to create a DrawableObject

    public abstract DrawableObject getDrawableObject(Class c);

}
```

For this purpose we created a DrawableObjectAbstractFactory to call the getDrawableObject method from the DrawableObjectFactory class. In this way in our SketchEditActivity we can create an object of DrawableObjectAbstractFactory class and use the method to create objects.

```
private PaintView paintView;
private boolean isButtonsHide = true;
private SketchEditActivityViewModel sketchViewModel;
private Intent intent;
private DrawableObjectAbstractFactory drawableObjectFactory;


private void selectTemplate(Class c, FloatingActionButton fab) {
    setSelected(drawableObjectFactory.getDrawableObject(c));
    animateButton(fab);
}
```

1.3.3 Template Pattern

Template Method Design Pattern is being used in our implementation to handle the different Android alert dialogues we are using in our UI. For this purpose we created an abstract class called DialogTemplate which defines a path of all the methods that we need to create dialog, where some steps are the same for all dialogs and other should be overridden for each alert dialogue, as needed in the further implementation.

```
public abstract class DialogTemplate {

    final AlertDialog dialogBuilder;
    LayoutInflater inflater;
    Context context;
    Button buttonSubmit;
    Button buttonCancel;
    View dialogView;

    public DialogTemplate(Context context, LayoutInflater inflater) {
        this.context = context;
        this.inflater = inflater;
        dialogBuilder = new AlertDialog.Builder(context).create();
    }


public void create() {
    createView();
    setViewToDialog();

    setCancelButton();
    setInputElements();
    setSubmitButton();

    submitButtonListener();
    cancelButtonListener();

    showDialog();
}
```

Each time we inflate the different view with the createView() method. So for example, to create the dialogue that has the function of getting text input from the user, we override methods of a DialogForText class that extends the DialogTemplate class, as seen in picture bellow:

```
@Override
void submitButtonListener() {
    buttonSubmit.setOnClickListener(view -> {
        sketchEditActivityViewModel.setTextForSelected(editText.getText().toString());
        dialogBuilder.dismiss();
    });
}

@Override
void setInputElements() { editText = dialogView.findViewById(R.id.edt_comment); }
```

1.3.4 Composite Pattern

Composite pattern was used, as defined, to handle the combined shapes of our individual objects. It extends from our DrawableObject class and has an ArrayList of DrawStartegy to process them. As CombinedShape is just an array of DrawableObjects, it can also include another CombinedShape, which also can have CombinedShapes in array. Hence, it will have a tree structure.

```
public class CombinedShape extends DrawableObject {

    private ArrayList<DrawStrategy> drawableObjects = new ArrayList<>();
    private String title;

    public CombinedShape(ArrayList<DrawStrategy> shapes) {
        super(Color.BLACK,  size: 5);
        shapes.forEach(selected -> drawableObjects.add(cloneSelected(selected)));
    }
```

And later when we work with CombinedShape, for example if we want to draw or move it(screen below) we can recursively go through such object and don't care about how many layers it has.

```
private void setNewCoordinate(DrawStrategy obj, Coordinate diff) {
    float newX;
    float newY;

    if (obj.getDrawableObject() instanceof DoublePointShape) {
        newX = ((DoublePointShape) obj.getDrawableObject()).getEndCoordinate().getX() + diff.getX();
        newY = ((DoublePointShape) obj.getDrawableObject()).getEndCoordinate().getY() + diff.getY();
        ((DoublePointShape) obj.getDrawableObject()).setEndCoordinate(new Coordinate(newX, newY));
    }

    if (obj.getDrawableObject() instanceof Polygon) {
        for (Coordinate c : ((Polygon) obj.getDrawableObject()).getCoordinates()) {
            c.setX(c.getX() + diff.getX());
            c.setY(c.getY() + diff.getY());
        }
    }

    if (obj.getDrawableObject() instanceof CombinedShape) {
        ((CombinedShape)obj.getDrawableObject()).getDrawableObjects().forEach(selected -> setNewCoordinate(selected, diff));
        return;
    }

    newX = obj.getDrawableObject().getAnchorCoordinate().getX() + diff.getX();
    newY = obj.getDrawableObject().getAnchorCoordinate().getY() + diff.getY();
    obj.getDrawableObject().setAnchorCoordinate(new Coordinate(newX, newY));
}
```

1.3.5 Facade Pattern

We implemented the Facade pattern to abstract complicated logic with a lot of third library dependencies. We used it to export our sketches as files by creating ImageSavingFacade which has only one public function saveImage. This class will keep all logic and dependencies for creating and saving JPG and PNG files.

```
public class ImageSavingFacade {

    public void saveImage(ArrayList<DrawStrategy> list, ExportFormat format, File dir, Context context) {

        try{
            String filename = "Sketch" + System.currentTimeMillis();
            File file= new File(dir, child: filename+"."+format);
            FileOutputStream fos= new FileOutputStream(file);
            if(format == ExportFormat.JPG) {
                saveAsJpg(list, fos, context);
            } else {
                saveAsPng(list, fos, context);
            }
        } catch (FileNotFoundException fileNotFoundException) {
            fileNotFoundException.printStackTrace();
        }
    }
```

2 Implementation

2.1 Overview of Main Modules and Components

For the drawing we created the abstract class DrawableObject and TextBox, CombinedShape and Shape are extending from this class. It has the base common attributes such as anchorCoordinate, color, size of shape or size of text and common methods such as getAnchorCoordinate which represents the objects beginning coordinate. Then we refined our objects by using the abstract Shape class. Basically all shape-like objects such as RadiusBasedShape, DoublePointShape or Polygon extend from this class. The next step was to determine what shape is radius based. So we created a Circle and a Triangle, since the canvas does not support triangles natively. We solved this issue by creating paths to form a triangle and use the radius to make it even on all ends. All shapes except for Polygon are DoublePointShapes, which means they have a point of a begin and a point of end. We used the point of end to calculate the distance of begin and end to get a radius. A Polygon is a special shape and required to store a list of drawn points, so we created a MultiPointShape in case we will add more classes in the future such as the Polygon.

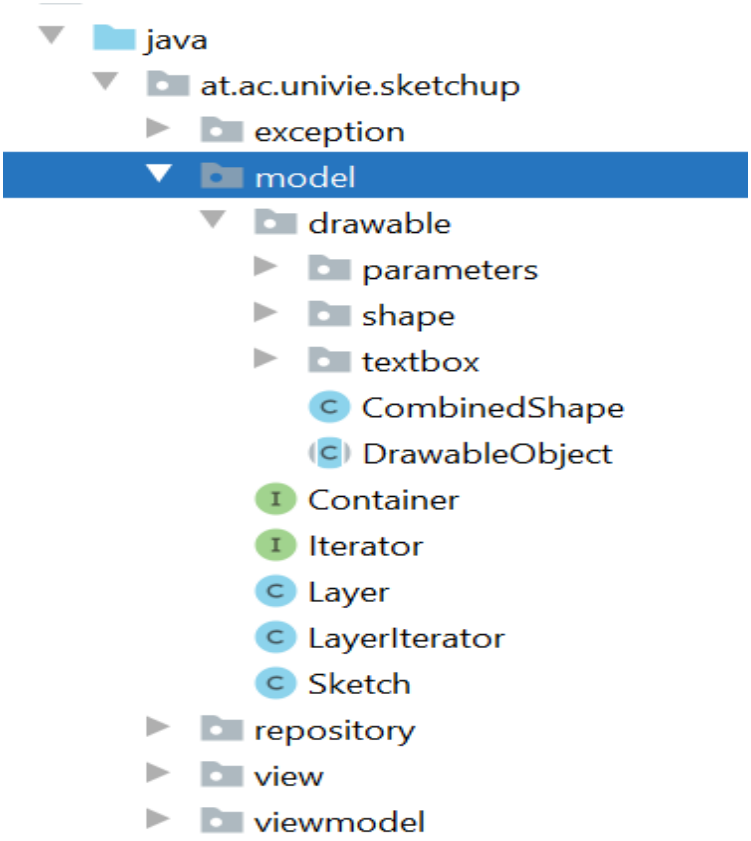
Every DrawableObject has its own way to draw it. To solve this problem we created a Strategy pattern. Our main concept was to pass the strategy a canvas in which we draw in and let it handle the rest. We created for each DrawableObject a class and implemented DrawStrategy a interface which has operations such as drawObject, onTouchDown, onTouchMove, onEditDown, onEditMove, inSelectedArea, getDrawableObject etc. These operations had a single task. To name a few, i. e. drawObject draws the DrawableObject in the strategy, inSelectedArea checks if the DrawableObject of the strategy is within the selected area, used to determine and move objects on the canvas.

We used a Sketch as a structure to store all the DrawStrategies on them. Working on several Sketches is also possible. The Sketches have Layers of the DrawStrategies inside. We also created a possibility to delete Sketches, store and load them. Finally we added a Facade pattern for exporting images.

How exactly this works is shown in the class diagram. For better reference, please see the SVG file and the source code.

2.2 Coding Practices

As the most common coding practice, we also decided to orient on Google Java Style Guide and this course learning material and implement them whenever possible. Our main goal was to make the code as readable as possible so any other user can easily understand its functionalities and purposes. We tried to do the code implementation as homogeneous as possible and also tried to use self-explaining names for classes and variables. We also tried to name our methods also in the same pattern, so that it is obvious what each of them does. Dividing our functionalities in packages that include the main goal or objects in its name, also helped us achieve this pattern bei coding. We also tried to briefly comment on every important part of the code and used Javadoc Guidelines for it. Following codes snippets are provided to show parts of these practices we mentioned above:



```
public class CombinedShape extends DrawableObject {

    private ArrayList<DrawStrategy> drawableObjects = new ArrayList<>();
    private String title;

    public CombinedShape(ArrayList<DrawStrategy> shapes) {
        super(Color.BLACK, 70);
        shapes.forEach(selected -> drawableObjects.add(cloneSelected(selected)));
    }
```

```
private void create3Layers() {
    Layer l1 = new Layer(true);
    Layer l2 = new Layer(true);
    Layer l3 = new Layer(true);

    layersList.add(l1);
    layersList.add(l2);
    layersList.add(l3);
}
```

```
    /**
     * Initiating the sketch toast with the hint
     */
    Context context = getApplicationContext();
    CharSequence text = "Long hold the buttons for the explanation";
    int duration = Toast.LENGTH_LONG;

    Toast toast = Toast.makeText(context, text, duration);
    toast.setGravity(Gravity.CENTER_HORIZONTAL, 0, 0);
    toast.show();
```

2.3 Defensive Programming

We used defensive programming to prevent users from trying to change attributes such as color or stroke width on non existing objects.

```
public void setSizeForSelected(int s) throws IncorrectAttributesException {
    if (this.template != null) {
        this.template.setInputSize(s);
        if (this.selectedDrawStrategies.size() > 0)
            this.selectedDrawStrategies.forEach(d -> d.getDrawableObject().setInputSize(s));
    } else {
        // Custom ExceptionClass Usage
        throw new IncorrectAttributesException("Select the element first to which size changes shou
    }
}
```

Every time this occurs the user is informed in a separate alert box that his action is not possible, as shown in picture below. It also prevents full app crash

```
@Override
void submitButtonListener() {
    buttonSubmit.setOnClickListener(view -> {
        try {
            sketchEditActivityViewModel.setSizeForSelected(Integer.parseInt(strokeWidth.getText().toString()));
            sketchEditActivityViewModel.setColorForSelected(((Color) colorSpinner.getSelectedItem()));
        } catch (Exception e) {
            DialogForError errorDialog = new DialogForError(context, inflater, "Wrong input", e.getMessage());
            errorDialog.create();
        }
        dialogBuilder.dismiss();
    });
```

Also in some places instead of returning null which can create unexpected exceptions late, we return object which will be not visible for a users(TextBox with empty string)

```
public DrawableObject getDrawableObject(Class c) {
    if (c == Line.class) return new Line();
    else if (c == Circle.class) return new Circle();
    else if (c == Quadrangle.class) return new Quadrangle();
    else if (c == Triangle.class) return new Triangle();
    else if (c == Polygon.class) return new Polygon();
    else if (c == TextBox.class) return new TextBox();
    return new TextBox();
}
```

To prevent invalid input data, which can crash the application we validate it and stop further changes.

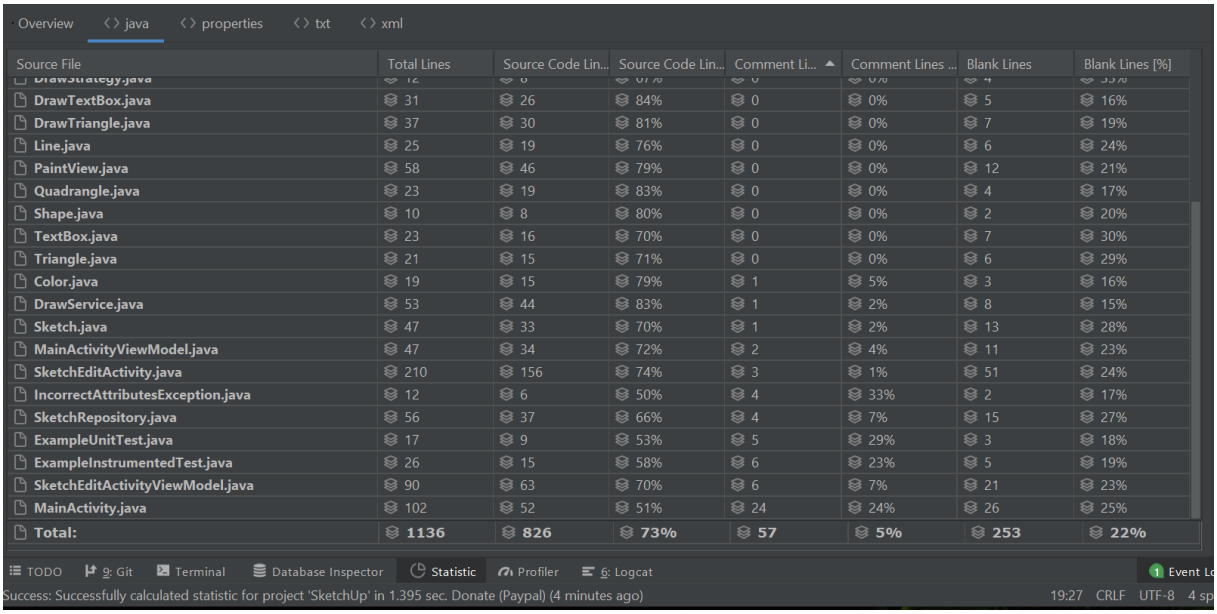
```
public void addSelectedToSketch() {
    if (template == null || this.selectedDrawStrategies.size() == 0) return;

    Sketch currentSketch = this.sketch.getValue();
    Objects.requireNonNull(currentSketch).addDrawableObject(this.selectedDrawStrategies.get(0));
    this.sketch.postValue(currentSketch);
    this.selectedDrawStrategies.clear();
}
```

3 Software Quality

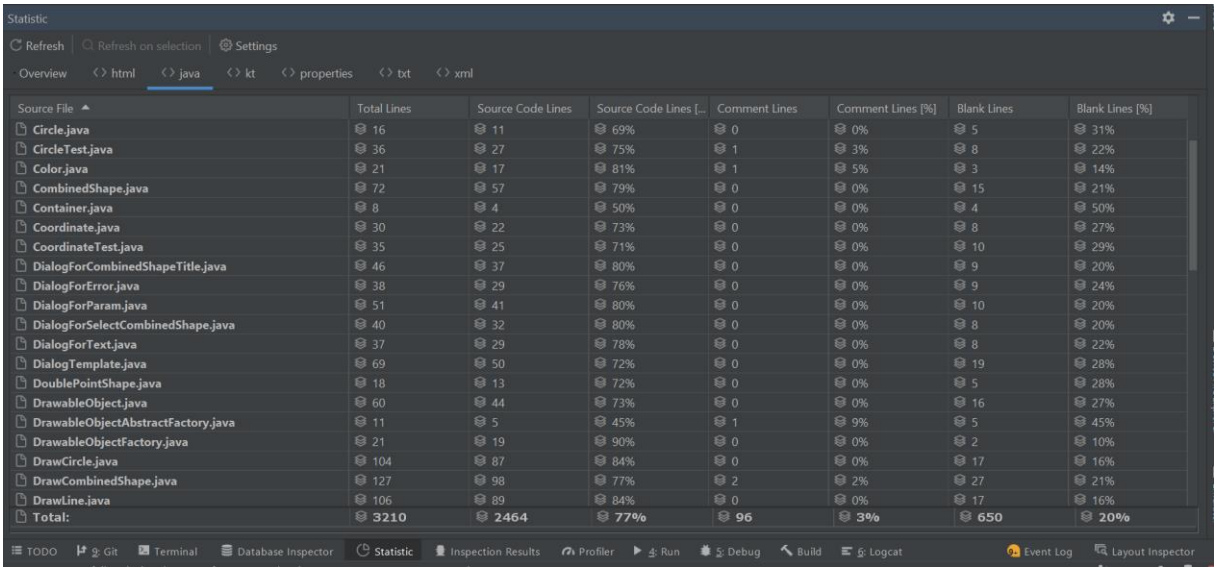
3.1 Code Metrics

On SUPD overall the project consists of 5 packages, 27 classes, 57 lines of comments and total number of codes is 1136.



Source File	Total Lines	Source Code Lin...	Source Code Lin...	Comment Li...	Comment Lines ...	Blank Lines	Blank Lines [%]
DrawTextBox.java	31	26	84%	0	0%	5	16%
DrawTriangle.java	37	30	81%	0	0%	7	19%
Line.java	25	19	76%	0	0%	6	24%
PaintView.java	58	46	79%	0	0%	12	21%
Quadrangle.java	23	19	83%	0	0%	4	17%
Shape.java	10	8	80%	0	0%	2	20%
TextBox.java	23	16	70%	0	0%	7	30%
Triangle.java	21	15	71%	0	0%	6	29%
Color.java	19	15	79%	1	5%	3	16%
DrawService.java	53	44	83%	1	2%	8	15%
Sketch.java	47	33	70%	1	2%	13	28%
MainActivityViewModel.java	47	34	72%	2	4%	11	23%
SketchEditActivity.java	210	156	74%	3	1%	51	24%
IncorrectAttributesException.java	12	6	50%	4	33%	2	17%
SketchRepository.java	56	37	66%	4	7%	15	27%
ExampleUnitTest.java	17	9	53%	5	29%	3	18%
ExampleInstrumentedTest.java	26	15	58%	6	23%	5	19%
SketchEditActivityViewModel.java	90	63	70%	6	7%	21	23%
MainActivity.java	102	52	51%	24	24%	26	25%
Total:	**1136**	**826**	**73%**	**57**	**5%**	**253**	**22%**

This time the total number of classes is 58, total number of code is 3210, 96 comment lines of code and other important metrics can be observed through the added and updated, for DEAD part, screenshot.



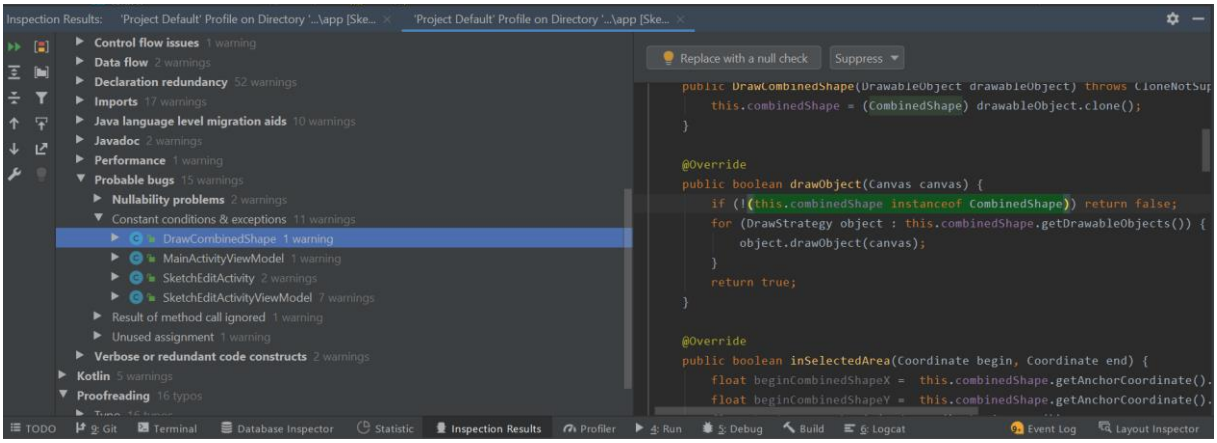
Source File	Total Lines	Source Code Lines	Source Code Lines [...	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
Circle.java	16	11	69%	0	0%	5	31%
CircleTest.java	36	27	75%	1	3%	8	22%
Color.java	21	17	81%	1	5%	3	14%
CombinedShape.java	72	57	79%	0	0%	15	21%
Container.java	8	4	50%	0	0%	4	50%
Coordinate.java	30	22	73%	0	0%	8	27%
CoordinateTest.java	35	25	71%	0	0%	10	29%
DialogForCombinedShapeTitle.java	46	37	80%	0	0%	9	20%
DialogForError.java	38	29	76%	0	0%	9	24%
DialogForParam.java	51	41	80%	0	0%	10	20%
DialogForSelectCombinedShape.java	40	32	80%	0	0%	8	20%
DialogForText.java	37	29	78%	0	0%	8	22%
DialogTemplate.java	69	50	72%	0	0%	19	28%
DoublePointShape.java	18	13	72%	0	0%	5	28%
DrawableObject.java	60	44	73%	0	0%	16	27%
DrawableObjectAbstractFactory.java	11	5	45%	1	9%	5	45%
DrawableObjectFactory.java	21	19	90%	0	0%	2	10%
DrawCircle.java	104	87	84%	0	0%	17	16%
DrawCombinedShape.java	127	98	77%	2	2%	27	21%
DrawLine.java	106	89	84%	0	0%	17	16%
Total:	**3210**	**2464**	**77%**	**96**	**3%**	**650**	**20%**

Bugs that we are aware of - could not be analysed because of missing time:

1. Selecting objects - in order to select a shape the whole object has to be selected, so if a user draws a shape which borders are not visible in the sketch itself, it will make it impossible to select the object and for example to change the attributes
2. Naming the combined shape - in this scenario the user is able to name the combined shapes identical which it makes hard to distinguish them

Probable bugs based on tool Lint:

By invoking the **lint** task and entering the command ***gradlew lint*** from the root directory of the project and getting the following results in the snippet from the Inspection Results we were able to see the recommendation concerning Performance, usability, correctness and also 4 probable bugs.



3.2 Test Cases for Functional Requirements

As a test case for functional requirements - JUnit tests were implemented for each model package in the project. The Test Classes were placed in at.ac.univie.sketchup (test) folder. Especially were used @After, @Before and @Test annotations of JUnit4 testing library. in order to mark methods as test methods. Furthermore were used the static methods as assertEquals of the specified testing library.

3.3 Quality Requirements Coverage

We tried to comment the code wherever possible to make it readable. By using Google Java Style Guide and JavaDocs comment guidelines we provided the first two quality requirements in our implementation. Also shown in chapter 2.2 of this paper.

For Q3 please visit chapter 2.2 of this paper.

For Q4 please visit chapter 2.3 of this paper.

For Q5 please visit chapter 2.2 of this paper.

For Q6 please visit chapter 3.2 of this paper

For Q7 please visit chapter 2.3 of this paper.

1 Team Contribution

1.1 Project Tasks and Schedule

Task	Begin	End	19.Oct-23.Oct Week 43	26.Oct-30.Oct Week 44	2.Nov-6.Nov Week 45	9.Nov-13.Nov Week 46	16.Nov-20.Nov Week 47	23.Nov-27.Nov Week 48	30.Nov-4.Dec Week 49	7.Dec-11.Dec Week 50	14.Dec-18.Dec Week 51	21.Dec-25.Dec Week 52	28.Dec-1.Jan Week 1	4.Jan-8.Jan Week 2	11.Jan-15.Jan Week 3
Define models / entities	20.10.20	18.12.20													
MVVM	22.10.20	30.10.20													
Drawable Objects & Attributes	20.10.20	07.01.21													
Edit Draw.Objects (Move, Delete, Attr.)	06.11.20	20.11.20													
List of Sketches	29.10.20	03.11.20													
CRUD functionality for Sketch	28.10.20	25.12.20													
Refactor project readability	12.11.20	08.01.21													
Patterns	10.11.20	06.01.21													
Testing and Exception Handling	28.10.20	17.11.20													
-Part 1	16.11.20	19.11.20													
-Part 2	17.12.20	21.12.20													
-Part 3	07.01.21	14.01.21													
Modelling	27.10.20	10.12.20													
-Part 1	21.10.20	27.10.20													
-Part 2	25.11.20	01.12.20													
Documentation	21.10.20	15.01.21													

1.2 Distribution of Work and Efforts

Team-Member	Task	Time used
Lala Mammadova	DEAD documentation, implementation of functional requirement regarding - saving, loading, deleting, editing of the Sketch, also the Implementation of the Iterator pattern, and the 9th functional requirement also wrote basic JUnit test cases, and of course worked on designing, replacement of the relevant buttons of the SketchApp	60h
Ema Dupovac-Kilincarslan	DEAD documentation, implementation of FR5, attributes,, implementation of the Abstract Pattern, Design and UI, Video	60h
Muhammed Akinci	DrawableObject, DrawStrategy, selection of DrawableObject, movement and deletion, UML diagram part 2, report, code documentation	60h
Maxim Bogoutdinov	DEAD documentation, Implementation of Facade, Template, Composite patterns. List of Sketches. Modelling. Entities. Refactor	60h