# Software Engineering 2
# DEAD REPORT
## Deadline Report

| | |
|---|---|
| **Team number:** | 0310 |

| **Team member 1** | |
|---|---|
| **Name:** | Christian Orlowski |
| **Student ID:** | 12024267 |
| **E-mail address:** | a12024267@unet.univie.ac.at |

| **Team member 2** | |
|---|---|
| **Name:** | Jonas Michael Speiser |
| **Student ID:** | 12012545 |
| **E-mail address:** | a12012545@unet.univie.ac.at |

| **Team member 3** | |
|---|---|
| **Name:** | Sandra Tomeschek |
| **Student ID:** | 01504450 |
| **E-mail address:** | a01504450@unet.univie.ac.at |

| **Team member 4** | |
|---|---|
| **Name:** | Leyla Durdyyeva |
| **Student ID:** | 01576824 |
| **E-mail address:** | a01576824@unet.univie.ac.at |

| **Team member 5** | |
|---|---|
| **Name:** | Felix Achim Hubert Lindau |
| **Student ID:** | 11948883 |
| **E-mail address:** | a11948883@unet.univie.ac.at |

# 1 Final Design

https://www.youtube.com/watch?v=6_r-Dsv6w14&feature=youtu.be

## 1.1 Design Approach and Overview

The following section describes our initial approach to the internal structure of our application:

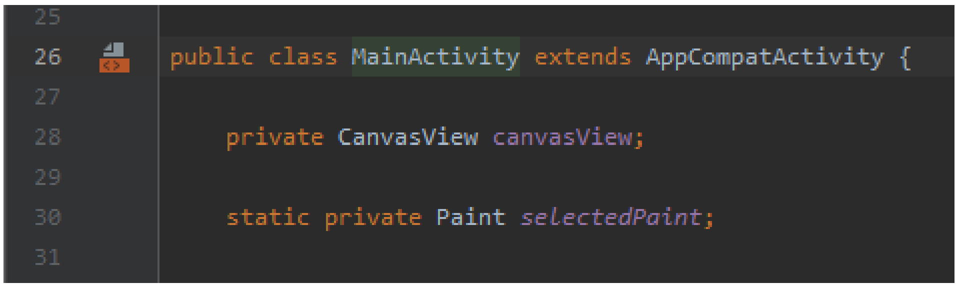
"Our internal software design is based on the Model – View – ViewModel approach. We thought, this might be a good idea, due to the model's ability to provide cohesive, externally low coupled classes and structure highly interactive systems in a maintainable manner. The "View-part" of the model is responsible for displaying our design and fetching user inputs. The "Model-part is mostly used for storing data. Both carry no logic for themselves. In contrast the "ViewModel-part" has logic, and its job is to process the user input and depending on the user actions populate the "View" with certain data from the "Model", such as drawing a circle on the screen."

From the start on, we strived to structure our code in accordance with the Model-View-ViewModel approach. As we now have realized, our implementation at the point of SUPD was not well done and we did not quite grasp the concept in its entirety. While the approach was correct, the implementation could not follow up and there was still a lot of business logic and presentation logic in our View class as well as further business logic in the ViewModel class.

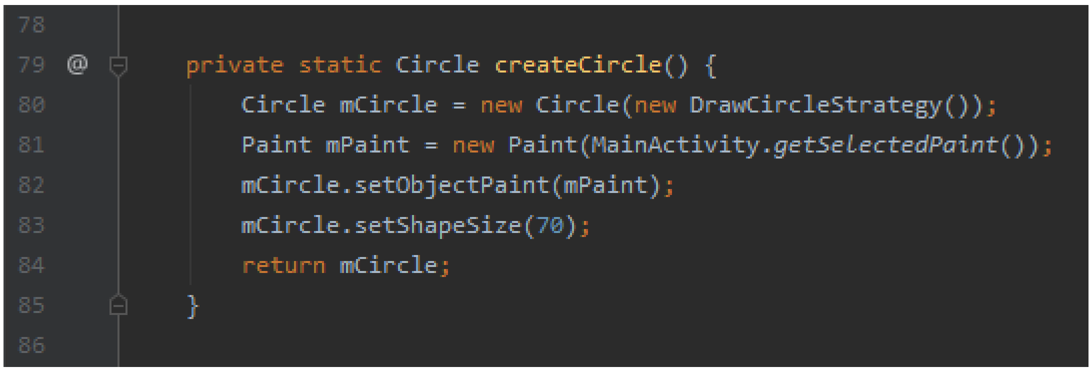
After SUPD, we reworked our code structure and now our business logic is mostly in the model classes (e.g., Sketch, Layer) and the presentation logic is placed in our two ViewModel classes "CanvasViewModel" and "MainActivityViewModel". The View classes: "MainActivity" and "CanvasView" should carry as little logic as possible.

This code clean-up and restructuring can be very well exemplified with the development of our "paint" object which is required so the canvas class can draw something on the screen. At the beginning, we passed a paint object with colour, stroke width and size from the "MainActivity" to the "selectedPaint" attribute.

*git: 36dfce0476edee184943df8547d6a644ca7ddb37 (\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\view\MainActivity.java*



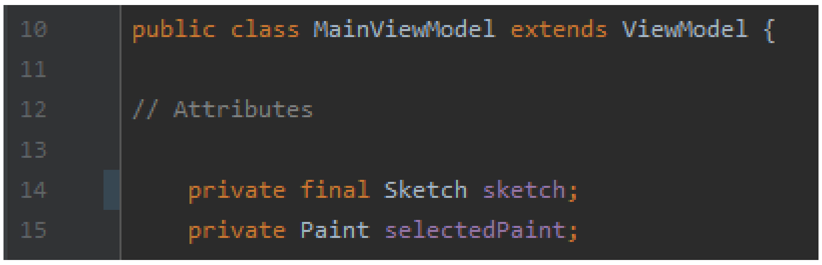
```java

public class MainActivity extends AppCompatActivity {

    private CanvasView canvasView;

    static private Paint selectedPaint;

```



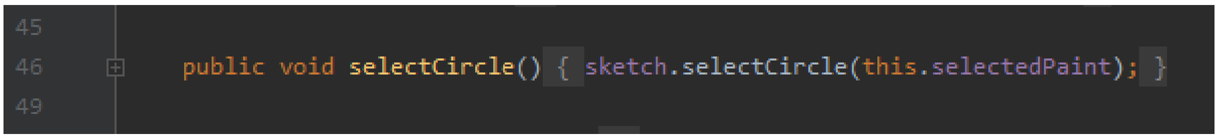
```java

    private static Circle createCircle() {
        Circle mCircle = new Circle(new DrawCircleStrategy());
        Paint mPaint = new Paint(MainActivity.getSelectedPaint());
        mCircle.setObjectPaint(mPaint);
        mCircle.setShapeSize(70);
        return mCircle;
    }

```

In the next iterative step, we move the attribute from the View class to the ViewModel class. From where it was further passed on to the GraphicalElementFactory.

*git: 6de8bc6aa4f08ae30abf8d29b0de74d312575602 (\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\viewmodel\MainViewModel.java*

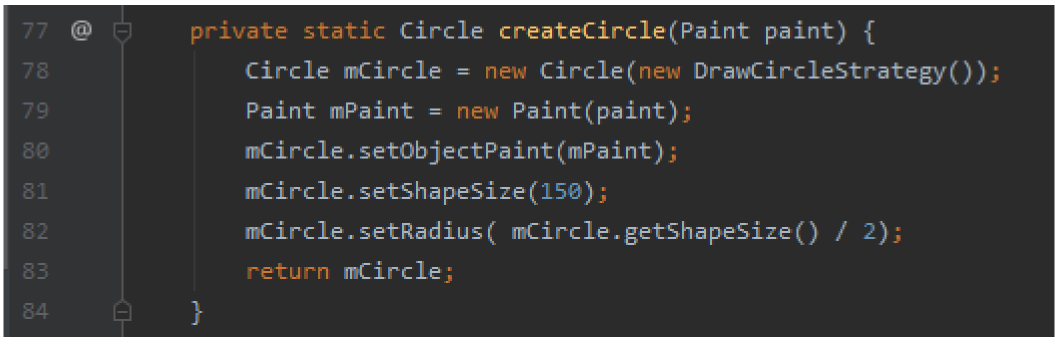


```java
public class MainViewModel extends ViewModel {

// Attributes

    private final Sketch sketch;
    private Paint selectedPaint;
```



```java

    public void selectCircle() { sketch.selectCircle(this.selectedPaint); }

```

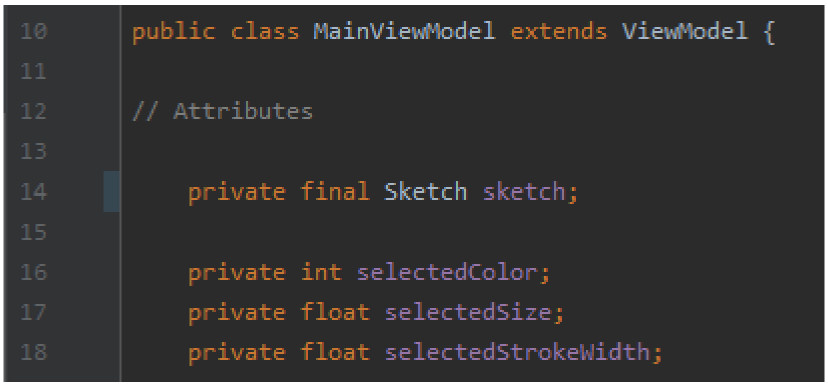
*git: 6de8bc6aa4f08ae30abf8d29b0de74d312575602 (\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\model\GraphicalElementFactory.java)*



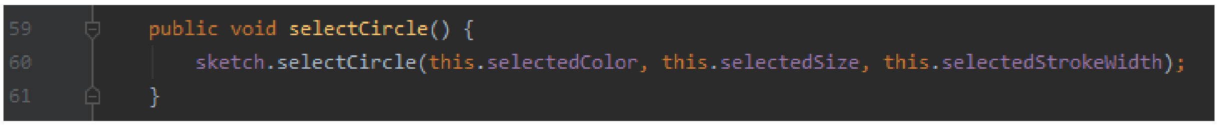
```java
    private static Circle createCircle(Paint paint) {
        Circle mCircle = new Circle(new DrawCircleStrategy());
        Paint mPaint = new Paint(paint);
        mCircle.setObjectPaint(mPaint);
        mCircle.setShapeSize(150);
        mCircle.setRadius( mCircle.getShapeSize() / 2);
        return mCircle;
    }
```

In order to adhere to the MVVM approach the creation of said paint object was moved to the model ("GraphicalElementFactory"). The ViewModel class only passed primitive values for colour, stroke width and size.

*git: b107e7f9e88e87c64d2b2a8aa7d579f4deeebab4 (\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\viewmodel\MainViewModel.java*

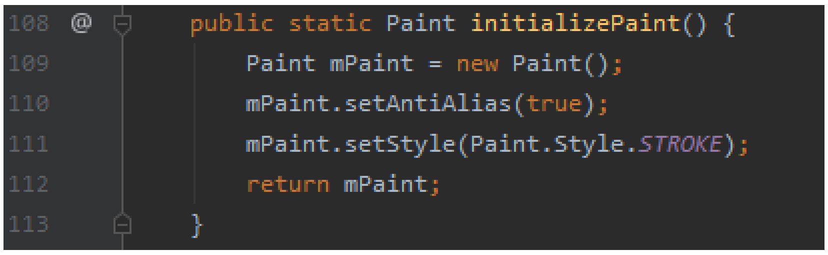


```java
public class MainViewModel extends ViewModel {

// Attributes

    private final Sketch sketch;

    private int selectedColor;
    private float selectedSize;
    private float selectedStrokeWidth;
```

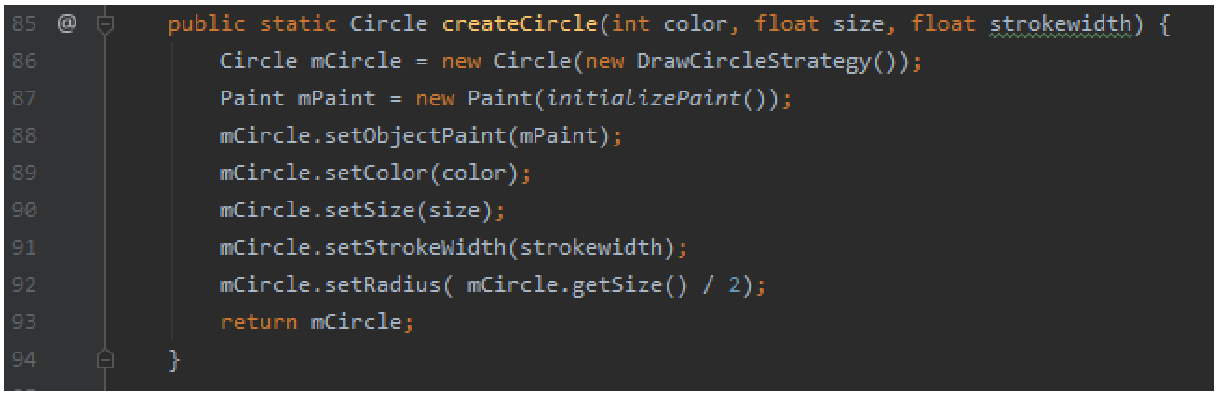


```java
    public void selectCircle() {
        sketch.selectCircle(this.selectedColor, this.selectedSize, this.selectedStrokeWidth);
    }
```

*git: b107e7f9e88e87c64d2b2a8aa7d579f4deeebab4 (\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\model\GraphicalElementFactory.java)*



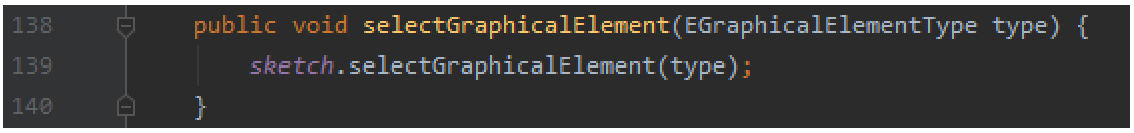
```java
    public static Paint initializePaint() {
        Paint mPaint = new Paint();
        mPaint.setAntiAlias(true);
        mPaint.setStyle(Paint.Style.STROKE);
        return mPaint;
    }
```



```java
public static Circle createCircle(int color, float size, float strokewidth) {
    Circle mCircle = new Circle(new DrawCircleStrategy());
    Paint mPaint = new Paint(initializePaint());
    mCircle.setObjectPaint(mPaint);
    mCircle.setColor(color);
    mCircle.setSize(size);
    mCircle.setStrokeWidth(strokewidth);
    mCircle.setRadius( mCircle.getSize() / 2);
    return mCircle;
}
```

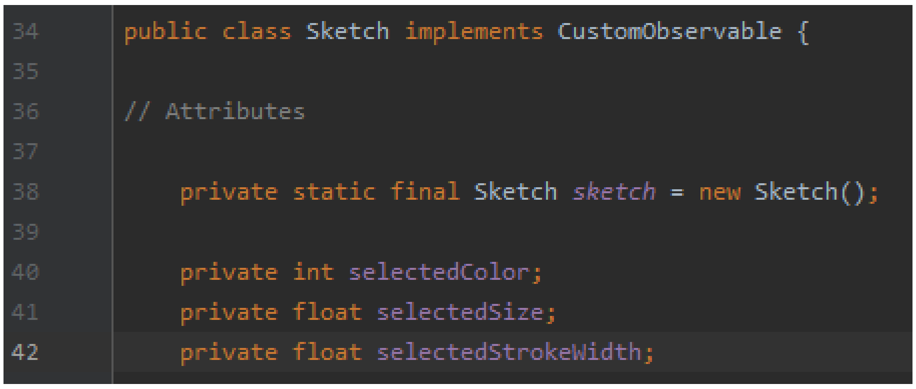
After another iteration of the paint object, we decided that we want to create the paint object just in time when it is necessary. Hence, it is created within the concrete “DrawingStrategy” classes where the canvas method “draw” relies on having a paint object. Outside of these classes, we only work with the primitive values which we receive from the “CanvasViewModel”. This leads to a better overall performance and is easier to work with, especially when we serialize objects. If we should decide to not use paint objects anymore or wanted to reuse parts of a code in another project where we do not have paint objects, there is a lot less to refactor.

*(\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\viewmodel\MainActivityViewModel.java*

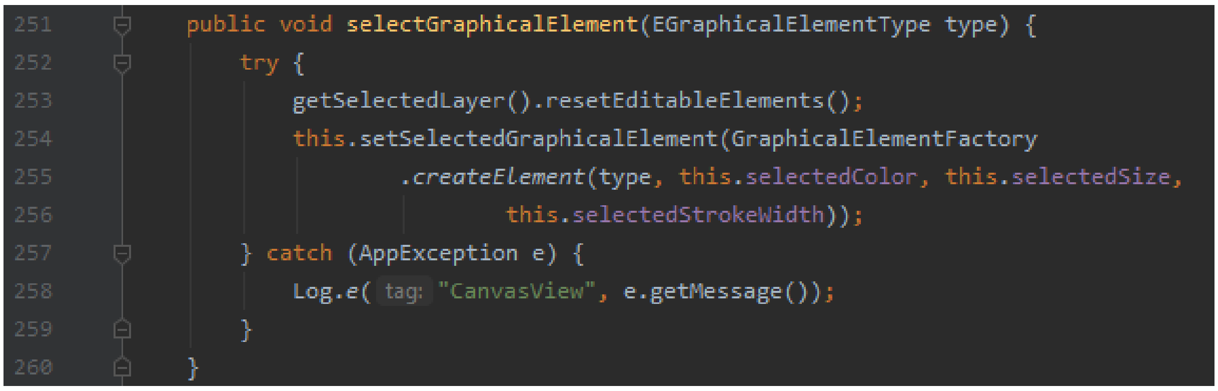


```java
public void selectGraphicalElement(EGraphicalElementType type) {
    sketch.selectGraphicalElement(type);
}
```

*(\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\model\Sketch.java)*

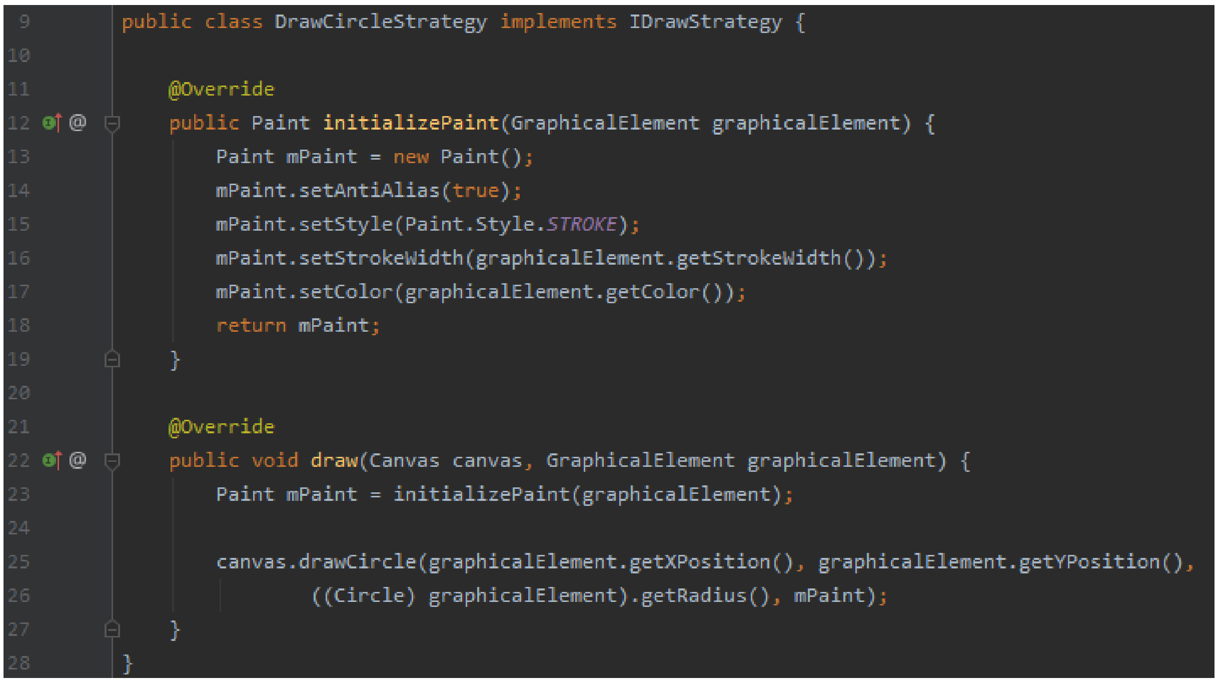


```java
public class Sketch implements CustomObservable {

// Attributes

    private static final Sketch sketch = new Sketch();

    private int selectedColor;
    private float selectedSize;
    private float selectedStrokeWidth;
```



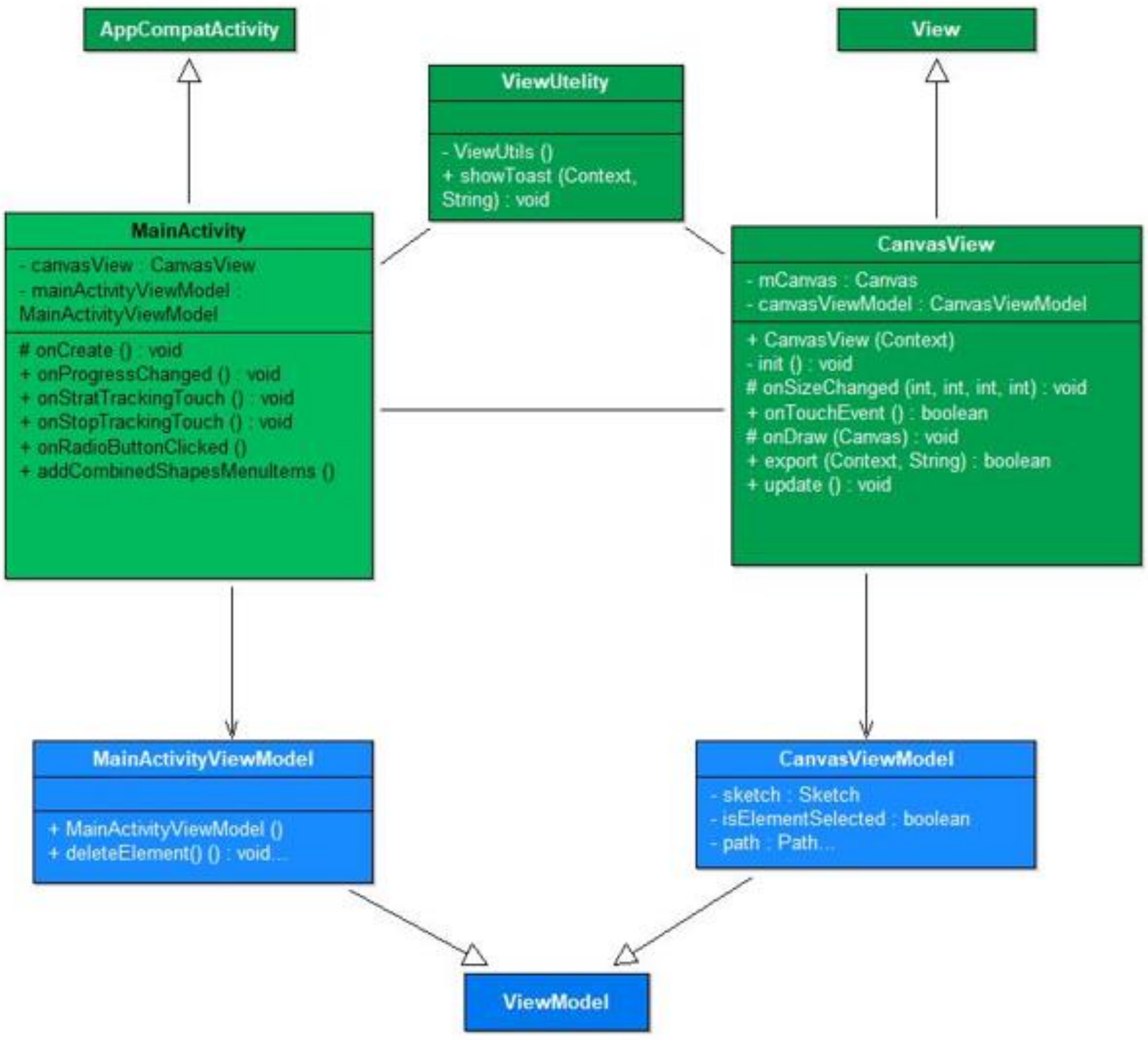
```java
    public void selectGraphicalElement(EGraphicalElementType type) {
        try {
            getSelectedLayer().resetEditableElements();
            this.setSelectedGraphicalElement(GraphicalElementFactory
                    .createElement(type, this.selectedColor, this.selectedSize,
                            this.selectedStrokeWidth));
        } catch (AppException e) {
            Log.e( tag: "CanvasView", e.getMessage());
        }
    }
```

*(\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\model\draw\DrawCircleStrategy.java*



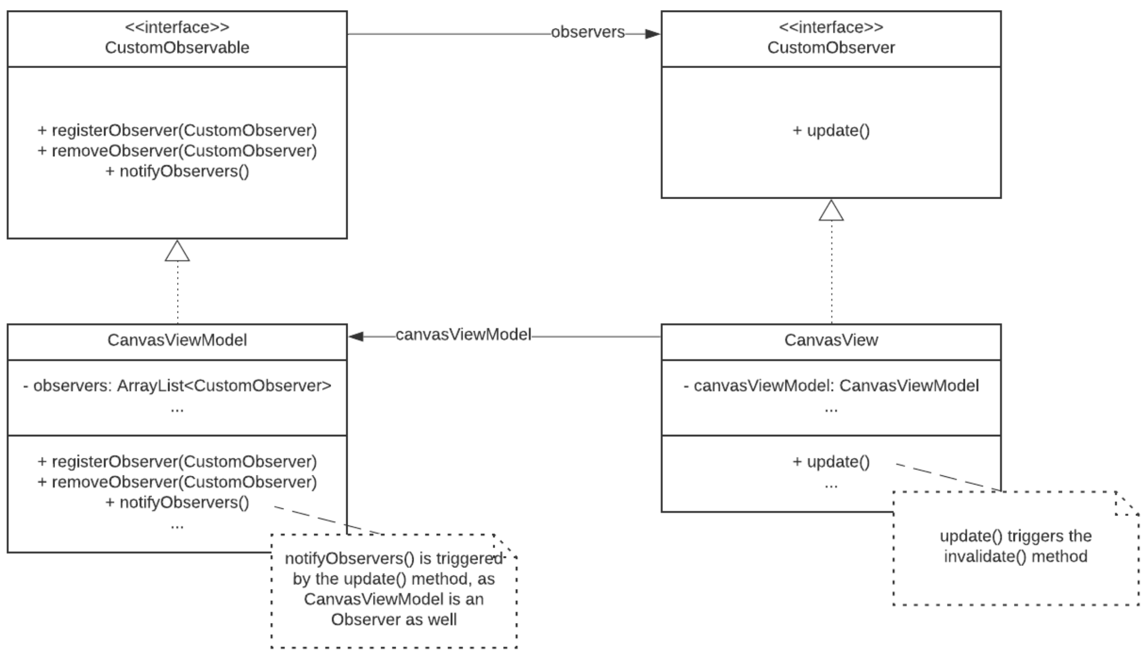
```java
public class DrawCircleStrategy implements IDrawStrategy {

    @Override
    public Paint initializePaint(GraphicalElement graphicalElement) {
        Paint mPaint = new Paint();
        mPaint.setAntiAlias(true);
        mPaint.setStyle(Paint.Style.STROKE);
        mPaint.setStrokeWidth(graphicalElement.getStrokeWidth());
        mPaint.setColor(graphicalElement.getColor());
        return mPaint;
    }

    @Override
    public void draw(Canvas canvas, GraphicalElement graphicalElement) {
        Paint mPaint = initializePaint(graphicalElement);

        canvas.drawCircle(graphicalElement.getXPosition(), graphicalElement.getYPosition(),
                ((Circle) graphicalElement).getRadius(), mPaint);
    }
}
```

This leads us to another active design decision. We felt that our ViewModel class was overloaded, so we decided to divide it into two classes, one that is related to the “Canvas” class itself and one for operations that deal with buttons and menus. Hence, the previously mentioned View and ViewModel classes were created.


AppCompatActivity
View
ViewUtelity
- ViewUtils ()
+ showToast (Context, String) : void
MainActivity
- canvasView : CanvasView
- mainActivityViewModel : MainActivityViewModel
# onCreate () : void
+ onProgressChanged () : void
+ onStratTrackingTouch () : void
+ onStopTrackingTouch () : void
+ onRadioButtonClicked ()
+ addCombinedShapesMenuItems ()
CanvasView
- mCanvas : Canvas
- canvasViewModel : CanvasViewModel
+ CanvasView (Context)
- init () : void
# onSizeChanged (int, int, int, int) : void
+ onTouchEvent () : boolean
# onDraw (Canvas) : void
+ export (Context, String) : boolean
+ update () : void
MainActivityViewModel
+ MainActivityViewModel ()
+ deleteElement() () : void...
CanvasViewModel
- sketch : Sketch
- isElementSelected : boolean
- path : Path...
ViewModel

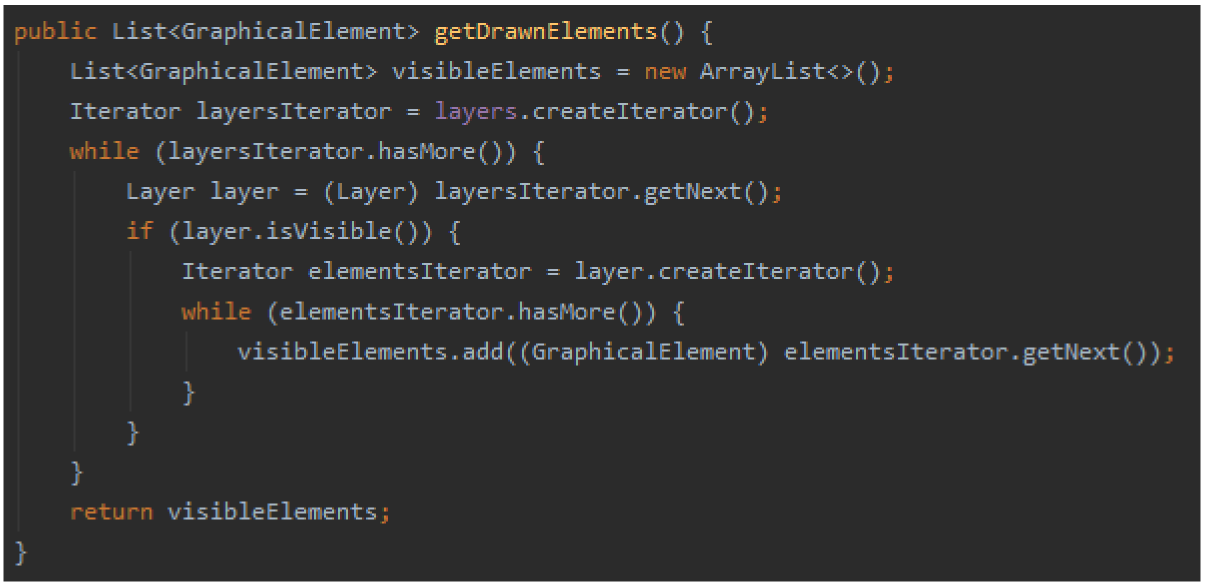
Further, with the one-by-one addition of design patterns, we had different options at our hands. One of the decisions with the most influence on our code structure, especially in regard to maintaining the MVVM approach was the implementation of the observer pattern (See 1.3).

By adding this pattern, we were able to establish a one directional flow of method calls from View classes to ViewModel classes ending at Model classes. If any information must go the other way around, the observer pattern takes care of it by updating the subscribed classes. “CanvasViewModel” reacts on changes in Model classes, while “CanvasView” reacts on changes in the ViewModel class.


<<interface>>
CustomObservable
+ registerObserver(CustomObserver)
+ removeObserver(CustomObserver)
+ notifyObservers()
observers
<<interface>>
CustomObserver
+ update()
CanvasViewModel
- observers: ArrayList<CustomObserver>
...
+ registerObserver(CustomObserver)
+ removeObserver(CustomObserver)
+ notifyObservers()
...
canvasViewModel
CanvasView
- canvasViewModel: CanvasViewModel
...
+ update()
...
notifyObservers() is triggered by the update() method, as CanvasViewModel is an Observer as well
update() triggers the invalidate() method

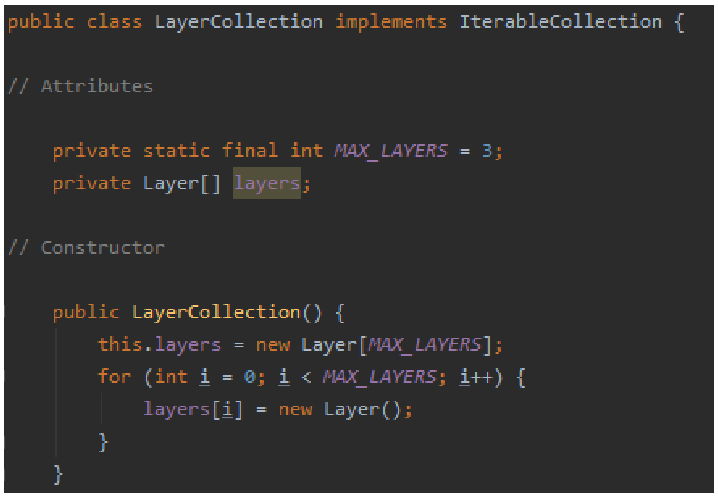
Another influential decision is connected to the implementation of the iterator pattern (see 1.3). We added three layer objects which all have an "ElementCollection" over which an "ElementCollectionIterator" traverses. Additionally, there is a "LayerCollectionIterator" which traverses over all active layers. As a result, we receive all graphical elements which should be displayed on the screen. This structural combination of collections and iterators is necessary, to properly map the requirement of multi-layered sketches and marks a milestone in the way, our graphical elements are created.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)


```java
public List<GraphicalElement> getDrawnElements() {
    List<GraphicalElement> visibleElements = new ArrayList<>();
    Iterator layersIterator = layers.createIterator();
    while (layersIterator.hasMore()) {
        Layer layer = (Layer) layersIterator.getNext();
        if (layer.isVisible()) {
            Iterator elementsIterator = layer.createIterator();
            while (elementsIterator.hasMore()) {
                visibleElements.add((GraphicalElement) elementsIterator.getNext());
            }
        }
    }
    return visibleElements;
}
```

Furthermore, we discussed the number of layers, the user can use to create his sketches. In the end, we decided to go with three layers, since it seemed sufficient for an enjoyable user experience. Therefore, an array was the data collection of our choice. We had a fixed amount of data and it made sense performance wise. However, we considered giving the user the option to create unlimited layers. In that case, an array would not have been the correct choice. Instead, we would have decided to go with a list, due to its flexibility.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\LayerCollection*.java)

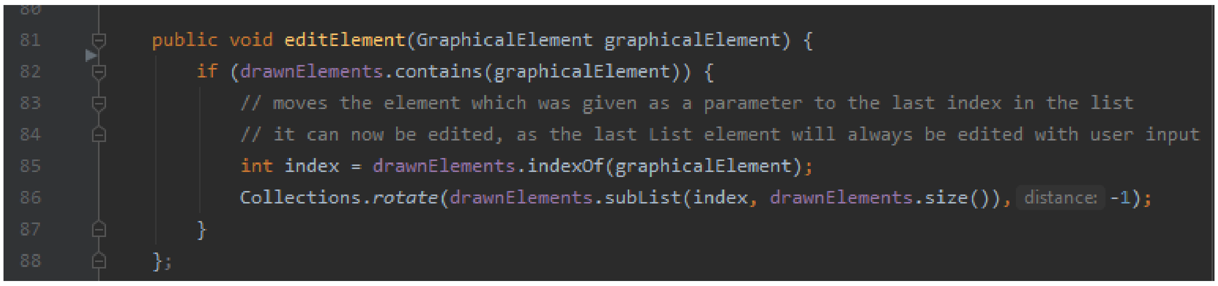


```
public class LayerCollection implements IterableCollection {

// Attributes

    private static final int MAX_LAYERS = 3;
    private Layer[] layers;

// Constructor

    public LayerCollection() {
        this.layers = new Layer[MAX_LAYERS];
        for (int i = 0; i < MAX_LAYERS; i++) {
            layers[i] = new Layer();
        }
    }
```

Continuing with the iterator pattern, we want to mention that it is also involved in the rework of how we make graphical elements editable. In our early solution only the last object in the "ElementCollection" was editable. Hence, if an object was supposed to change its colour, it had to be moved at the end of its collection. This was done by rotating the elements in the data structure. At a certain point of our implementation, this was no longer feasible, since we had to come up with a way to make multiple objects editable. Our solution was to create an additional Collection which holds the indices of all created graphical elements. Therefore, all objects have to stay at their place within the "ElementCollection" and are still distinguishable, due their unique indices.

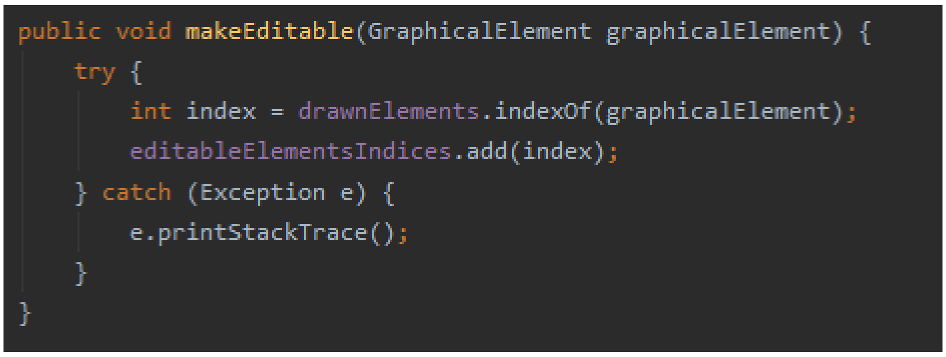
*git: 0c385d625453e85cfbe01f0e5ace59954b6d98e6*

*(\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\model\Layer.java)*



```java
    public void editElement(GraphicalElement graphicalElement) {
        if (drawnElements.contains(graphicalElement)) {
            // moves the element which was given as a parameter to the last index in the list
            // it can now be edited, as the last List element will always be edited with user input
            int index = drawnElements.indexOf(graphicalElement);
            Collections.rotate(drawnElements.subList(index, drawnElements.size()), distance: -1);
        }
    };
```

*(\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020 team0310\sketch_app\model\Layer.java)*

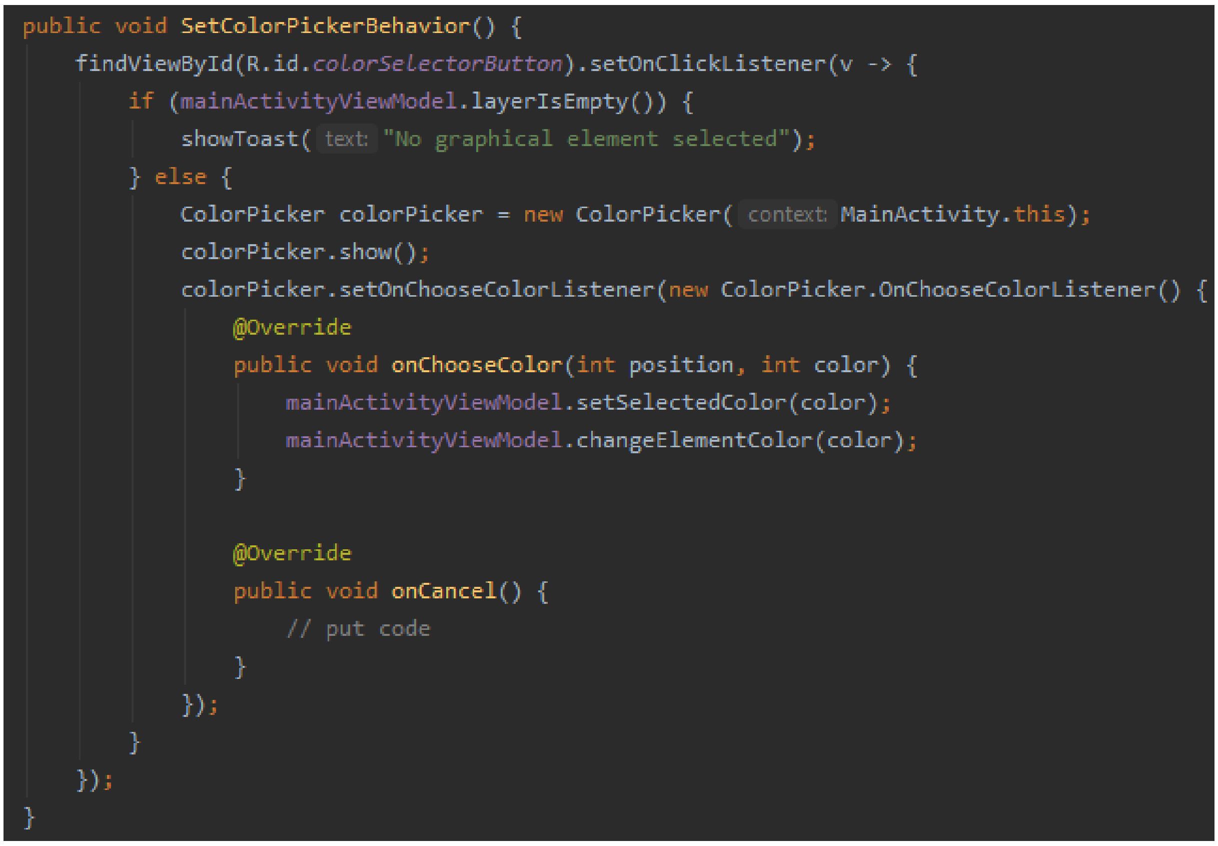


```java
public void makeEditable(GraphicalElement graphicalElement) {
    try {
        int index = drawnElements.indexOf(graphicalElement);
        editableElementsIndices.add(index);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

We also considered the implementation of other design patterns. For instance, one alternative which we considered, was the abstract factory pattern. Our approach would have been to create an abstract factory for filled objects and one for objects which only show the outline (the way our shapes are displayed right now). In the end, we decided that other design patterns would be more useful to our application, so we discarded it.

Regarding the technology stack, we only added two third party libraries before testing our implementation. The first one is the ColorPicker which was introduced early on, in order to provide a reliable source for modifying the colour of our graphical elements. Alternatives like the Holopicker and the ColorPickerView were discarded due to less extensive documentations and incompatibility with our design.

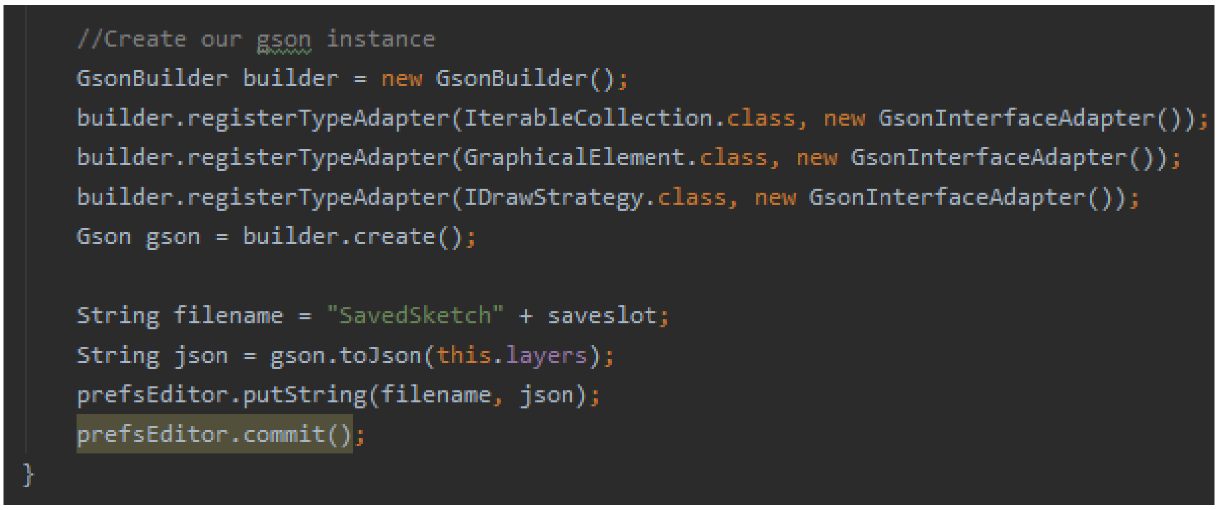
*(\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\view\MainActivity.java)*



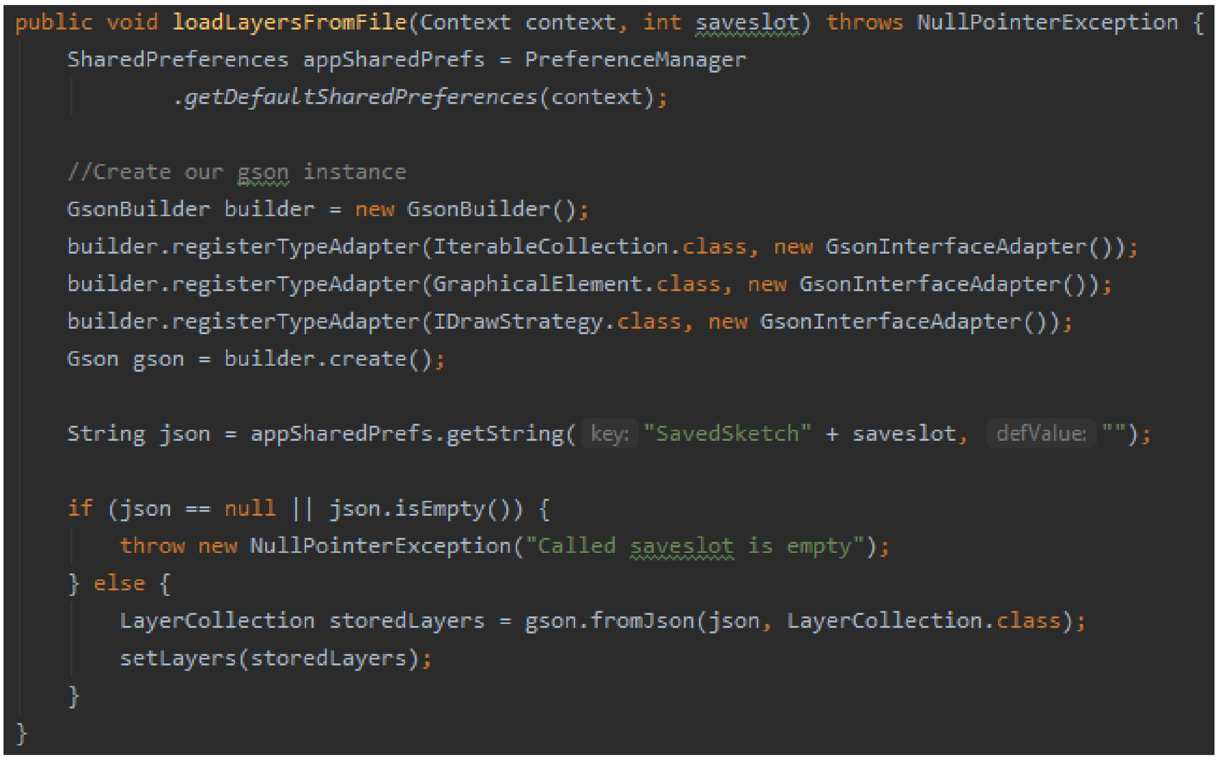
```java
public void SetColorPickerBehavior() {
    findViewById(R.id.colorSelectorButton).setOnClickListener(v -> {
        if (mainActivityViewModel.layerIsEmpty()) {
            showToast( text: "No graphical element selected");
        } else {
            ColorPicker colorPicker = new ColorPicker( context: MainActivity.this);
            colorPicker.show();
            colorPicker.setOnChooseColorListener(new ColorPicker.OnChooseColorListener() {
                @Override
                public void onChooseColor(int position, int color) {
                    mainActivityViewModel.setSelectedColor(color);
                    mainActivityViewModel.changeElementColor(color);
                }

                @Override
                public void onCancel() {
                    // put code
                }
            });
        }
    });
}
```

Close to the end of the project we also added the Gson library which is required for our approach to saving and loading sketches. We favoured this library over the counterpart from android, due to the same reasons as above: Ease of use and better documentation.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)

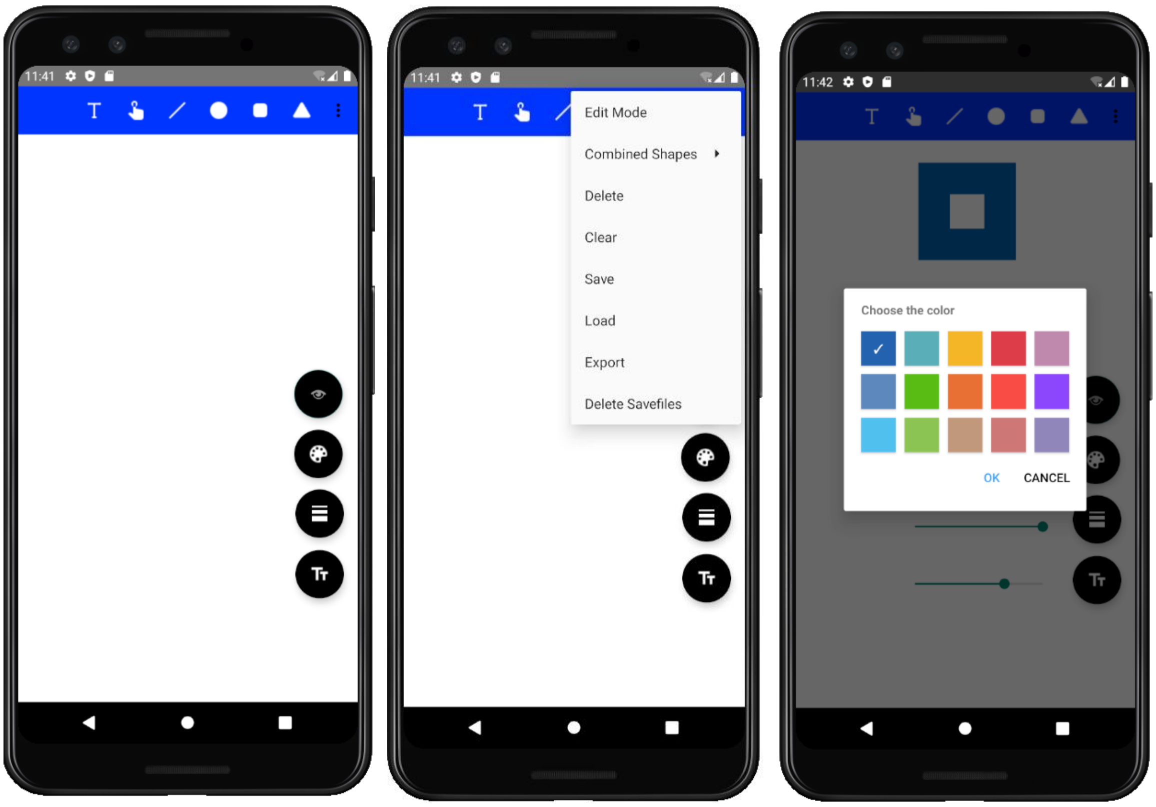


```java
    //Create our gson instance
    GsonBuilder builder = new GsonBuilder();
    builder.registerTypeAdapter(IterableCollection.class, new GsonInterfaceAdapter());
    builder.registerTypeAdapter(GraphicalElement.class, new GsonInterfaceAdapter());
    builder.registerTypeAdapter(IDrawStrategy.class, new GsonInterfaceAdapter());
    Gson gson = builder.create();

    String filename = "SavedSketch" + saveslot;
    String json = gson.toJson(this.layers);
    prefsEditor.putString(filename, json);
    prefsEditor.commit();
}
```



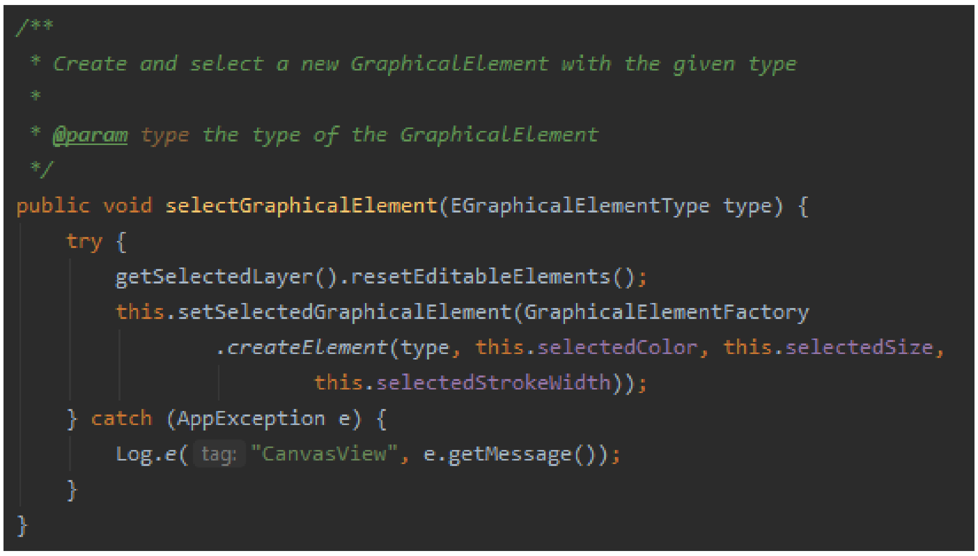
```java
public void loadLayersFromFile(Context context, int saveslot) throws NullPointerException {
    SharedPreferences appSharedPrefs = PreferenceManager
            .getDefaultSharedPreferences(context);

    //Create our gson instance
    GsonBuilder builder = new GsonBuilder();
    builder.registerTypeAdapter(IterableCollection.class, new GsonInterfaceAdapter());
    builder.registerTypeAdapter(GraphicalElement.class, new GsonInterfaceAdapter());
    builder.registerTypeAdapter(IDrawStrategy.class, new GsonInterfaceAdapter());
    Gson gson = builder.create();

    String json = appSharedPrefs.getString( key: "SavedSketch" + saveslot,  defValue: "");

    if (json == null || json.isEmpty()) {
        throw new NullPointerException("Called saveslot is empty");
    } else {
        LayerCollection storedLayers = gson.fromJson(json, LayerCollection.class);
        setLayers(storedLayers);
    }
}
```

Finally, we arrived at the current design of our app which we will discuss in the following paragraph. Let us have a quick overview of the graphical user interface which our application provides. When a user starts the application, he can choose to open the layer menu (black icon on the right representing an eye) and switch layers or make them visible as he wishes (up to three are possible). However, it is only possible to draw on one layer at a time. Afterwards, the user is able to select what type of graphical element he wants to create. By tapping on the corresponding icon and then choosing a position on the screen, the selected object will be displayed. On the top right-hand corner, additional options are provided which relate to actions that affect sketches in general and not only single objects (e.g., "Save", "Load" and "Clear" sketches). However, there are two exceptions to this generalization, namely, "Edit Mode" and "Delete" which enable the user to select and modify at least a single object. In terms of further modification, the buttons on the right-hand side allow the user to edit selected objects in terms of size, stroke widths and colour.



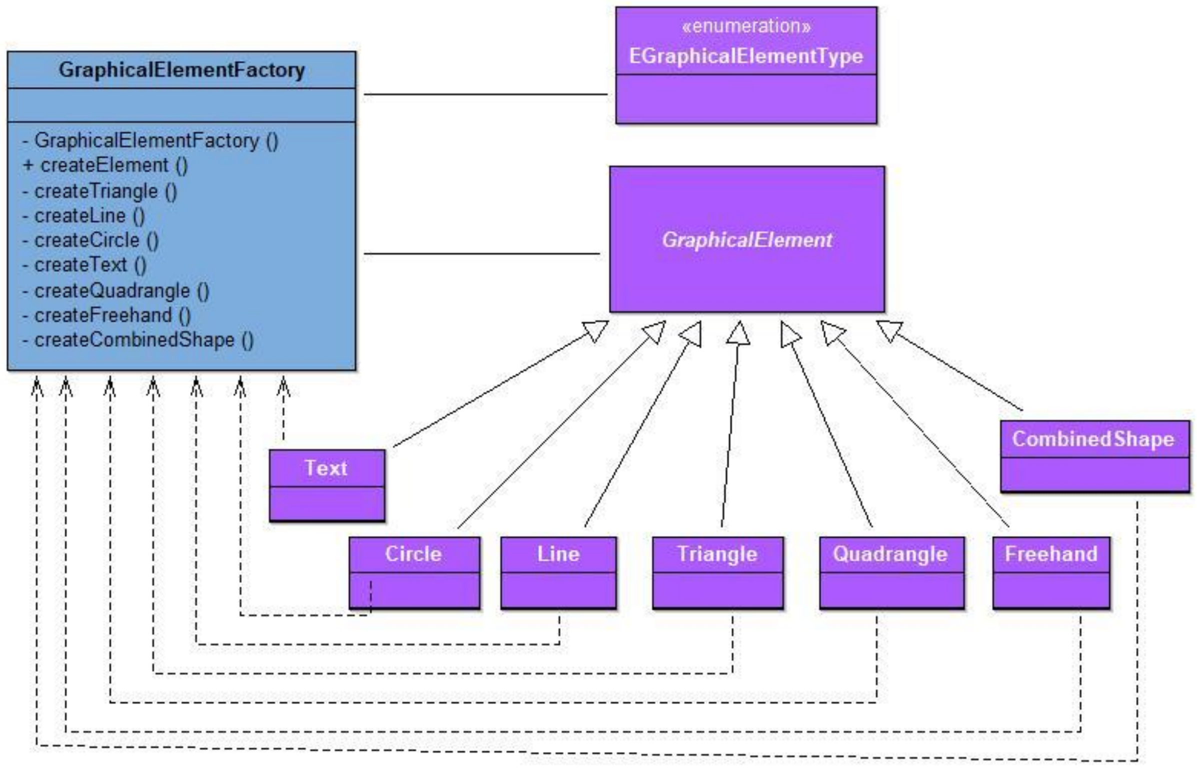
In order to explain our final design approach, we will walk through the process of a displaying a graphical element on the screen. When a user starts to interact with the application by selecting a graphical element, the "MainActivity" and the "MainActivityViewModel" transmit the prompt to the model where our Business logic resides. There, the "selectGraphicalElement" method receives the type of the chosen graphical element and adds the current values for colour, stroke widths and size which are stored as attributes in the class "Sketch".

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)



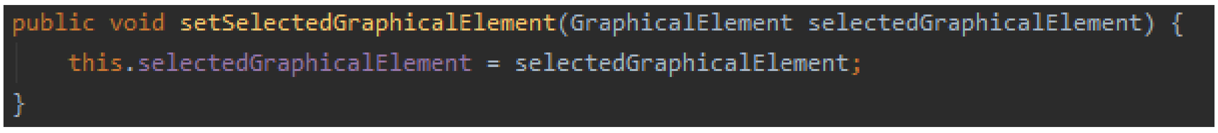
```
/**
 * Create and select a new GraphicalElement with the given type
 *
 * @param type the type of the GraphicalElement
 */
public void selectGraphicalElement(EGraphicalElementType type) {
    try {
        getSelectedLayer().resetEditableElements();
        this.setSelectedGraphicalElement(GraphicalElementFactory
                .createElement(type, this.selectedColor, this.selectedSize,
                        this.selectedStrokeWidth));
    } catch (AppException e) {
        Log.e( tag: "CanvasView", e.getMessage());
    }
}
```

This information is passed to the "GraphicalElementFactory" where, depending on the chosen type and given attributes, an object of that type is created. Furthermore, a "DrawStrategy" which correlates to the type of the graphical element, is added to the object. This process is supported by a factory pattern. The explicit creation of the object in question is delegated to a responsible class. For each type of graphical element, such a class exists which results in an internally low coupled code (as further described in 1.3).



The "setSelectedGraphicalElement" method is called which saves the freshly created graphical element object with all its attributes in the "Sketch" class.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)



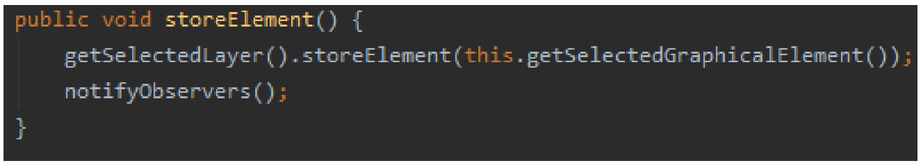
```
public void setSelectedGraphicalElement(GraphicalElement selectedGraphicalElement) {
    this.selectedGraphicalElement = selectedGraphicalElement;
}
```

So far, we have considered the business logic, which is carried out once the user chooses a graphical element. In the following, the user selects an empty spot on the display where he wants his graphical object to be drawn. This action triggers a sequence of steps:

First of all, the "TouchDownEvent" in "CanvasView" is triggered and the received information (coordinates) is passed through the

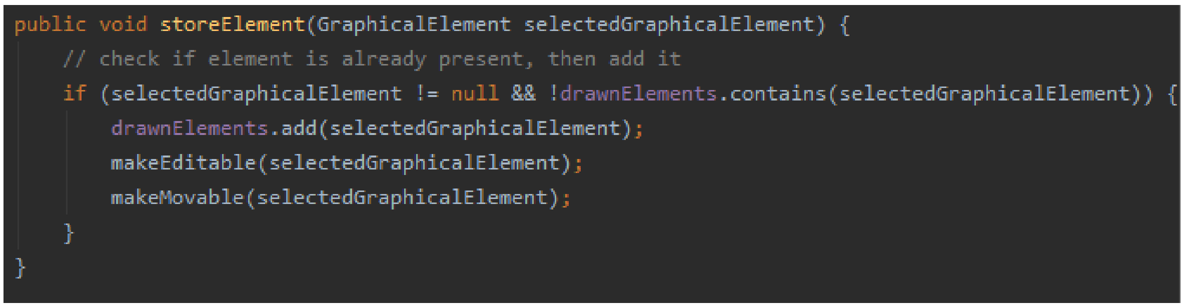
"CanvasViewModel" to the "Sketch" class. As a result, the previously created graphical element object which is currently saved in the "selectedGraphicalElement" attribute is added to the data collection on the active layer which is called "ElementCollection".

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)



```java
public void storeElement() {
    getSelectedLayer().storeElement(this.getSelectedGraphicalElement());
    notifyObservers();
}
```

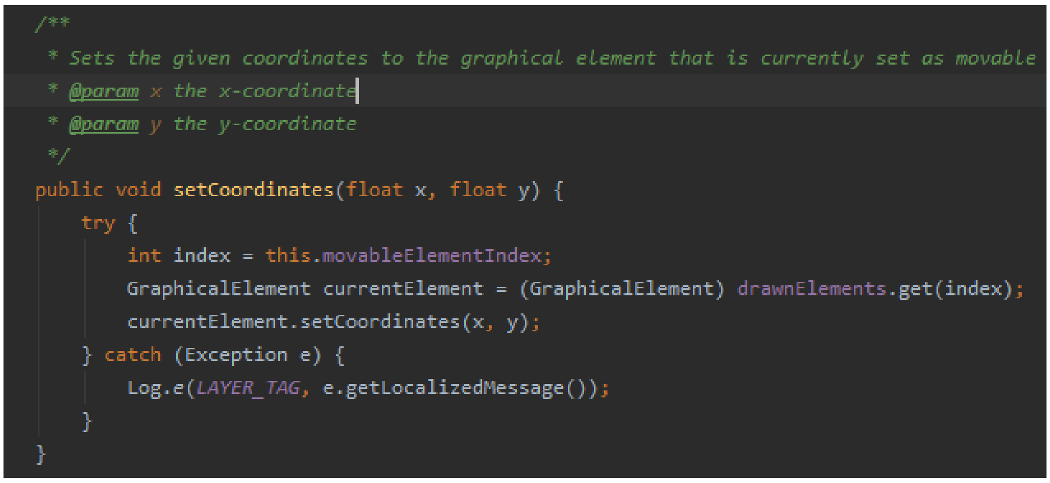
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Layer*.java)



```java
public void storeElement(GraphicalElement selectedGraphicalElement) {
    // check if element is already present, then add it
    if (selectedGraphicalElement != null && !drawnElements.contains(selectedGraphicalElement)) {
        drawnElements.add(selectedGraphicalElement);
        makeEditable(selectedGraphicalElement);
        makeMovable(selectedGraphicalElement);
    }
}
```

Also, a corresponding index is added to an additional data collection. Hence, the object is distinct and set as editable and movable. Invoking the "setCoordinates" method in the "Sketch" class adds the coordinates which we received from the "TouchDownEvent" on the CanvasView to our graphical element, as it is set to movable.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Layer*.java)

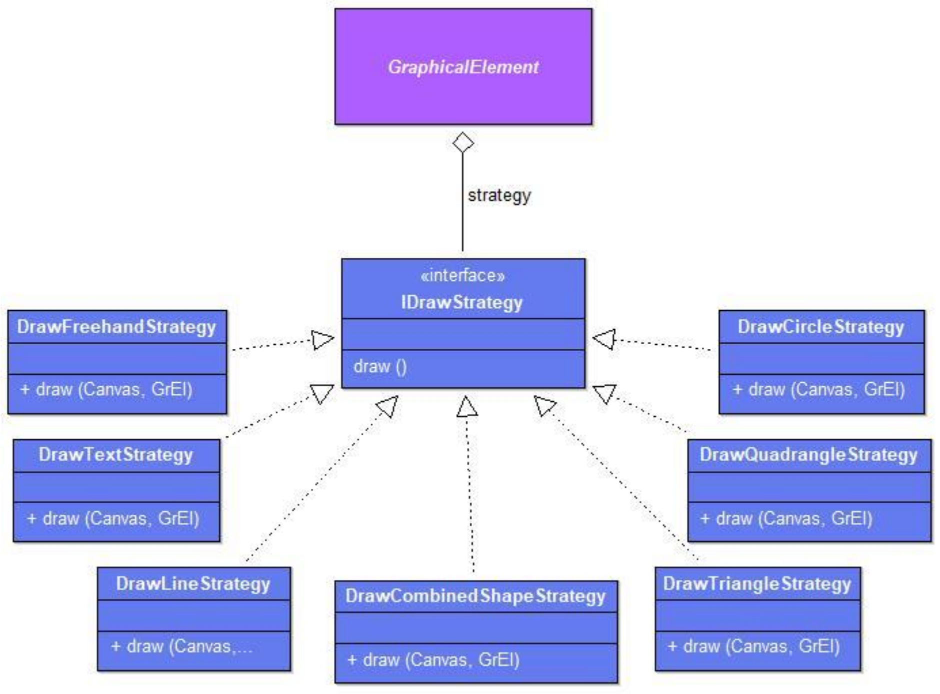


```java
/**
 * Sets the given coordinates to the graphical element that is currently set as movable
 * @param x the x-coordinate
 * @param y the y-coordinate
 */
public void setCoordinates(float x, float y) {
    try {
        int index = this.movableElementIndex;
        GraphicalElement currentElement = (GraphicalElement) drawnElements.get(index);
        currentElement.setCoordinates(x, y);
    } catch (Exception e) {
        Log.e(LAYER_TAG, e.getLocalizedMessage());
    }
}
```

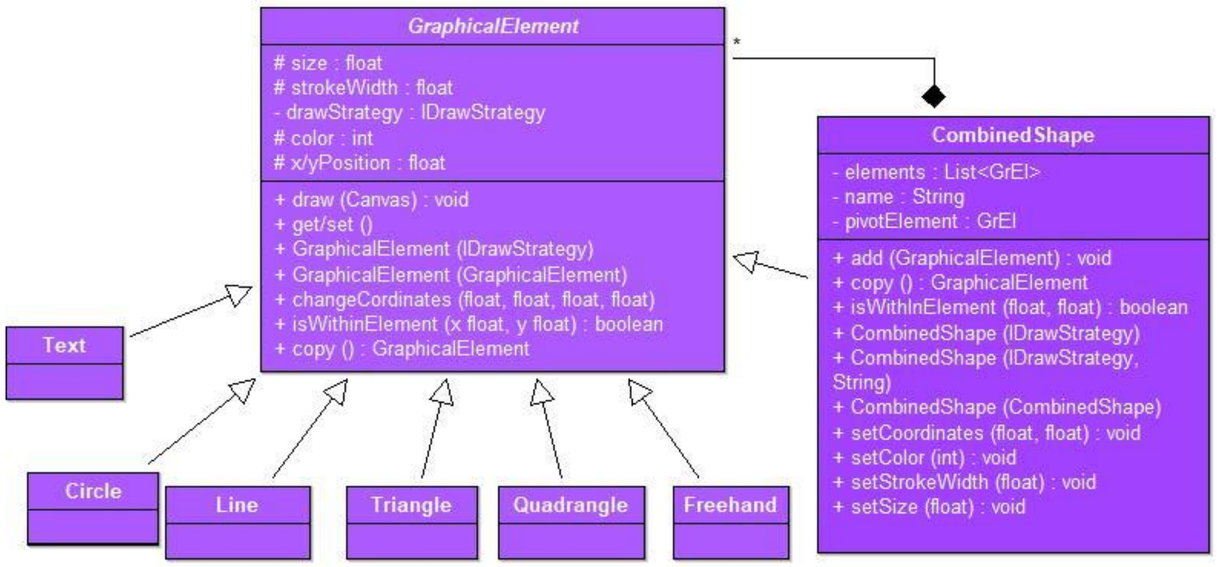
Now, our business logic for the creation of a new graphical element is complete and an additional design pattern is entrusted with the task to update the presentation logic. We implemented two observers. One on the "CanvasViewModel" class and one on the "CanvasView" class.

Once, this chain of observers is notified, based on the described change in the business logic, the "invalidate" method in the "CanvasView" is invoked which in return triggers "onDraw". Here, we use the iterator pattern (see 1.3 for more details) to traverse over the data collection of each active layer. Thereby, we receive all graphical elements that should be displayed on the screen with all their necessary information and finally, the draw method is invoked on each graphical element which is found.

The draw method is part of the third design pattern, the strategy pattern (1.3 for further reference). Each graphical element must be drawn in a different way on the canvas due to its different criteria. These criteria are specified in a unique "DrawGraphicalElementStrategy" class for each of the graphical elements. All these classes override the "draw" method in their own way when it is invoked.

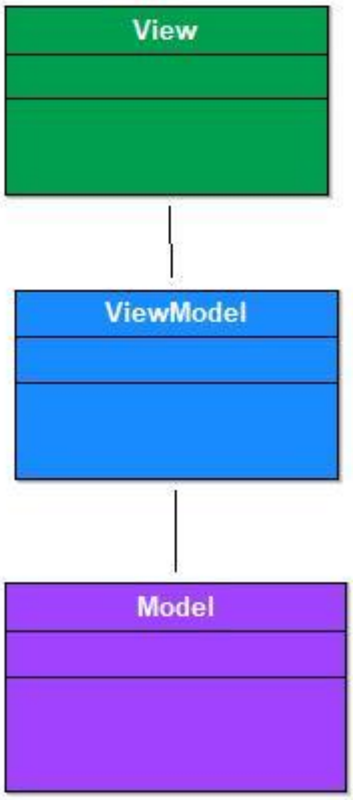


This process is the same for combined shapes which are created via the composite pattern. We make use of that pattern by adding primitive graphical elements (leaf objects) to complex combined shapes (container objects). These combined shapes can hold any number of leaf or container objects, representing a part-whole hierarchy. The "CombinedShape" object delegates requests to its leaf objects (See 1.3).

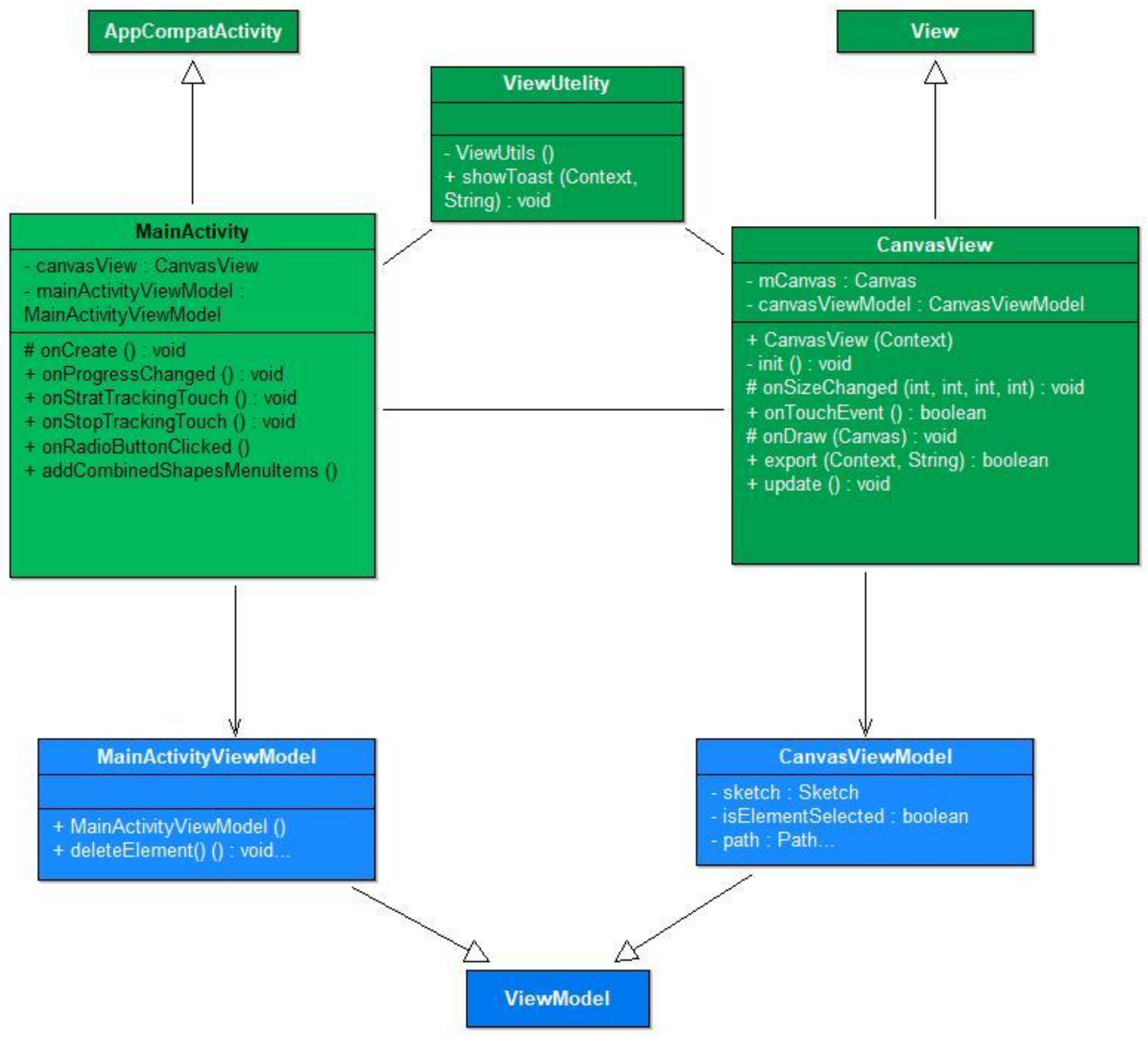

GraphicalElement
# size : float
# strokeWidth : float
- drawStrategy : IDrawStrategy
# color : int
# x/yPosition : float
+ draw (Canvas) : void
+ get/set ()
+ GraphicalElement (IDrawStrategy)
+ GraphicalElement (GraphicalElement)
+ changeCordinates (float, float, float, float)
+ isWithinElement (x float, y float) : boolean
+ copy () : GraphicalElement
*
CombinedShape
- elements : List<GrEl>
- name : String
- pivotElement : GrEl
+ add (GraphicalElement) : void
+ copy () : GraphicalElement
+ isWithInElement (float, float) : boolean
+ CombinedShape (IDrawStrategy)
+ CombinedShape (IDrawStrategy, String)
+ CombinedShape (CombinedShape)
+ setCoordinates (float, float) : void
+ setColor (int) : void
+ setStrokeWidth (float) : void
+ setSize (float) : void
Text
Circle
Line
Triangle
Quadrangle
Freehand

### 1.1.1 Class Diagrams

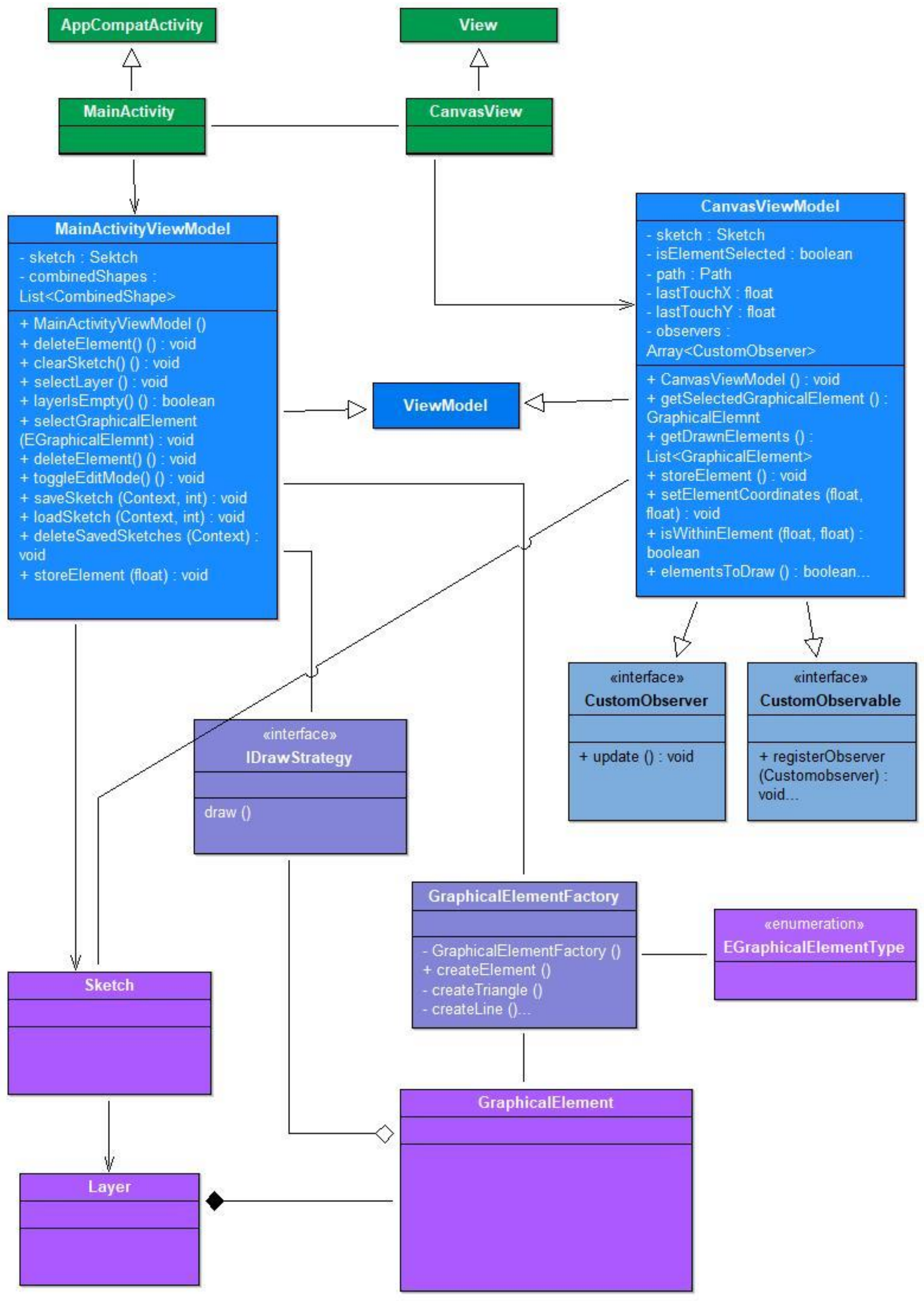
General MVVM – approach:



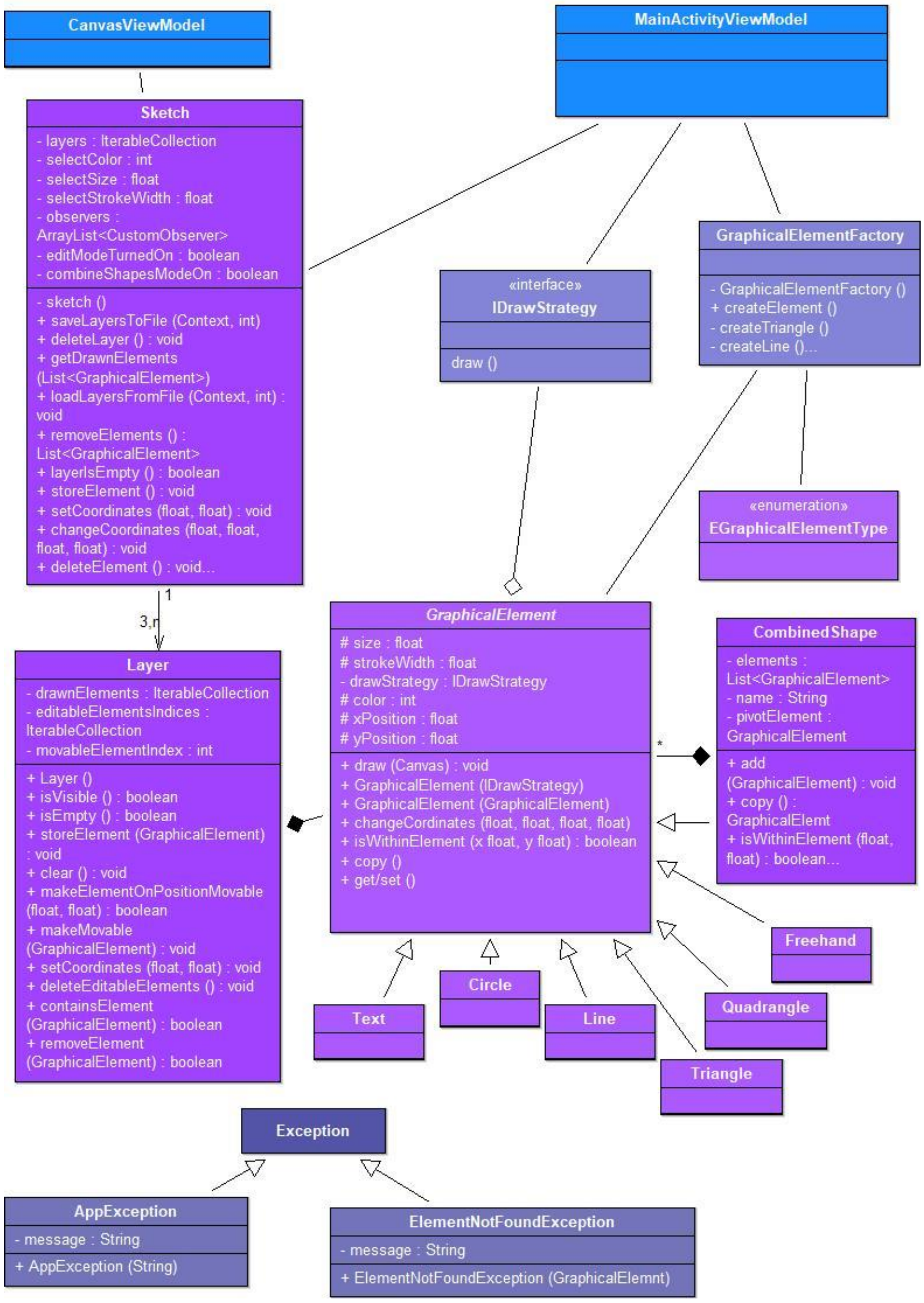
View and ViewModel:


AppCompatActivity
ViewUtelity
- ViewUtils ()
+ showToast (Context, String) : void
View
MainActivity
- canvasView : CanvasView
- mainActivityViewModel : MainActivityViewModel
# onCreate () : void
+ onProgressChanged () : void
+ onStratTrackingTouch () : void
+ onStopTrackingTouch () : void
+ onRadioButtonClicked ()
+ addCombinedShapesMenuItems ()
CanvasView
- mCanvas : Canvas
- canvasViewModel : CanvasViewModel
+ CanvasView (Context)
- init () : void
# onSizeChanged (int, int, int, int) : void
+ onTouchEvent () : boolean
# onDraw (Canvas) : void
+ export (Context, String) : boolean
+ update () : void
MainActivityViewModel
+ MainActivityViewModel ()
+ deleteElement() () : void...
CanvasViewModel
- sketch : Sketch
- isElementSelected : boolean
- path : Path...
ViewModel

View, ViewModel and Model:


AppCompatActivity
View
MainActivity
CanvasView
MainActivityViewModel
- sketch : Sektch
- combinedShapes : List<CombinedShape>
+ MainActivityViewModel ()
+ deleteElement() () : void
+ clearSketch() () : void
+ selectLayer () : void
+ layerIsEmpty() () : boolean
+ selectGraphicalElement (EGraphicalElemnt) : void
+ deleteElement() () : void
+ toggleEditMode() () : void
+ saveSketch (Context, int) : void
+ loadSketch (Context, int) : void
+ deleteSavedSketches (Context) : void
+ storeElement (float) : void
ViewModel
CanvasViewModel
- sketch : Sketch
- isElementSelected : boolean
- path : Path
- lastTouchX : float
- lastTouchY : float
- observers : Array<CustomObserver>
+ CanvasViewModel () : void
+ getSelectedGraphicalElement () : GraphicalElemnt
+ getDrawnElements () : List<GraphicalElement>
+ storeElement () : void
+ setElementCoordinates (float, float) : void
+ isWithinElement (float, float) : boolean
+ elementsToDraw () : boolean...
«interface» CustomObserver
+ update () : void
«interface» CustomObservable
+ registerObserver (Customobserver) : void...
«interface» IDrawStrategy
draw ()
GraphicalElementFactory
- GraphicalElementFactory ()
+ createElement ()
- createTriangle ()
- createLine ()...
«enumeration» EGraphicalElementType
Sketch
Layer
GraphicalElement

ViewModel and Model:


CanvasViewModel
MainActivityViewModel
Sketch
- layers : IterableCollection
- selectColor : int
- selectSize : float
- selectStrokeWidth : float
- observers : ArrayList<CustomObserver>
- editModeTurnedOn : boolean
- combineShapesModeOn : boolean
- sketch ()
+ saveLayersToFile (Context, int)
+ deleteLayer () : void
+ getDrawnElements (List<GraphicalElement>)
+ loadLayersFromFile (Context, int) : void
+ removeElements () : List<GraphicalElement>
+ layerIsEmpty () : boolean
+ storeElement () : void
+ setCoordinates (float, float) : void
+ changeCoordinates (float, float, float, float) : void
+ deleteElement () : void...
«interface»
IDrawStrategy
draw ()
GraphicalElementFactory
- GraphicalElementFactory ()
+ createElement ()
- createTriangle ()
- createLine ()...
«enumeration»
EGraphicalElementType
1
3,n
Layer
- drawnElements : IterableCollection
- editableElementsIndices : IterableCollection
- movableElementIndex : int
+ Layer ()
+ isVisible () : boolean
+ isEmpty () : boolean
+ storeElement (GraphicalElement) : void
+ clear () : void
+ makeElementOnPositionMovable (float, float) : boolean
+ makeMovable (GraphicalElement) : void
+ setCoordinates (float, float) : void
+ deleteEditableElements () : void
+ containsElement (GraphicalElement) : boolean
+ removeElement (GraphicalElement) : boolean
GraphicalElement
# size : float
# strokeWidth : float
- drawStrategy : IDrawStrategy
# color : int
# xPosition : float
# yPosition : float
+ draw (Canvas) : void
+ GraphicalElement (IDrawStrategy)
+ GraphicalElement (GraphicalElement)
+ changeCordinates (float, float, float, float)
+ isWithinElement (x float, y float) : boolean
+ copy ()
+ get/set ()
*
CombinedShape
- elements : List<GraphicalElement>
- name : String
- pivotElement : GraphicalElement
+ add (GraphicalElement) : void
+ copy () : GraphicalElemt
+ isWithinElement (float, float) : boolean...
Freehand
Quadrangle
Triangle
Line
Circle
Text
Exception
AppException
- message : String
+ AppException (String)
ElementNotFoundException
- message : String
+ ElementNotFoundException (GraphicalElemnt)

### 1.1.2 Technology Stack

Our Technology Stack is based on the programming language Java. We went with this language, due to its ubiquity and broad range of functionalities. Moreover, we did not have any prior experience with Kotlin which simplified the decision. Our Software Development Kit (SDK) is Android Studio, since it is Google's official development platform for Android applications and offers great aid for the creation of user interfaces. Therefore, it was an obvious decision. In the following table, we present all third-party libraries which we used in our implementation:

| Name | Website link | Version | Purpose and Reasoning |
|---|---|---|---|
| ColorPicker | https://github.com/kristiyanP/colorpicker | 1.1.10 | Is implementing the color palette for the color change of objects, which is linked to the floating button in the bottom left corner. Considered alternatives where the Holopicker (by M. Schweiz) and ColorPickerView (by skydoves). Holopicker was not fitting to our design so well, and the documentation of ColorPickerView was not as extensive as the ColorPicker library. |
| Gson | https://github.com/google/gson/blob/master/UserGuide.md | 2.8.6 | Used to serialize and deserialize objects for the save and load functionality for sketches. It was chosen over the android SDK Serializable-Interface because there is no need for Classes to implement a certain interface. Furthermore, its ease of use and excellent documentation were factored into that decision. |
| Mockito | https://site.mockito.org/ | 1.10.19 | Use of JUnit and Mockito<br><br>For the test cases, we used the Frameworks of JUnit and Mockito. |

| | | | |
|---|---|---|---|
| Junit | https://junit.org/junit4/ | 4.13.1 | JUnit offered us the necessary functionalities for creating the test classes (with the @test annotation functionalities for the methods), whereas Mockito allowed us to mock dependencies of methods on other classes or interfaces than the test class itself. The latter was especially helpful with regard to the design of our app classes according to the Model-View-ViewModel scheme. Alternatives that we considered included the Appium testing framework, Robolectric and the mockk-framework, but the amount of documentation material and issue resolving information led us to the decisions towards JUnit and Mockito. |

We tested our application with the "Pixel_3_API_30" Emulator. The corresponding configurations are attached at the end of the report.

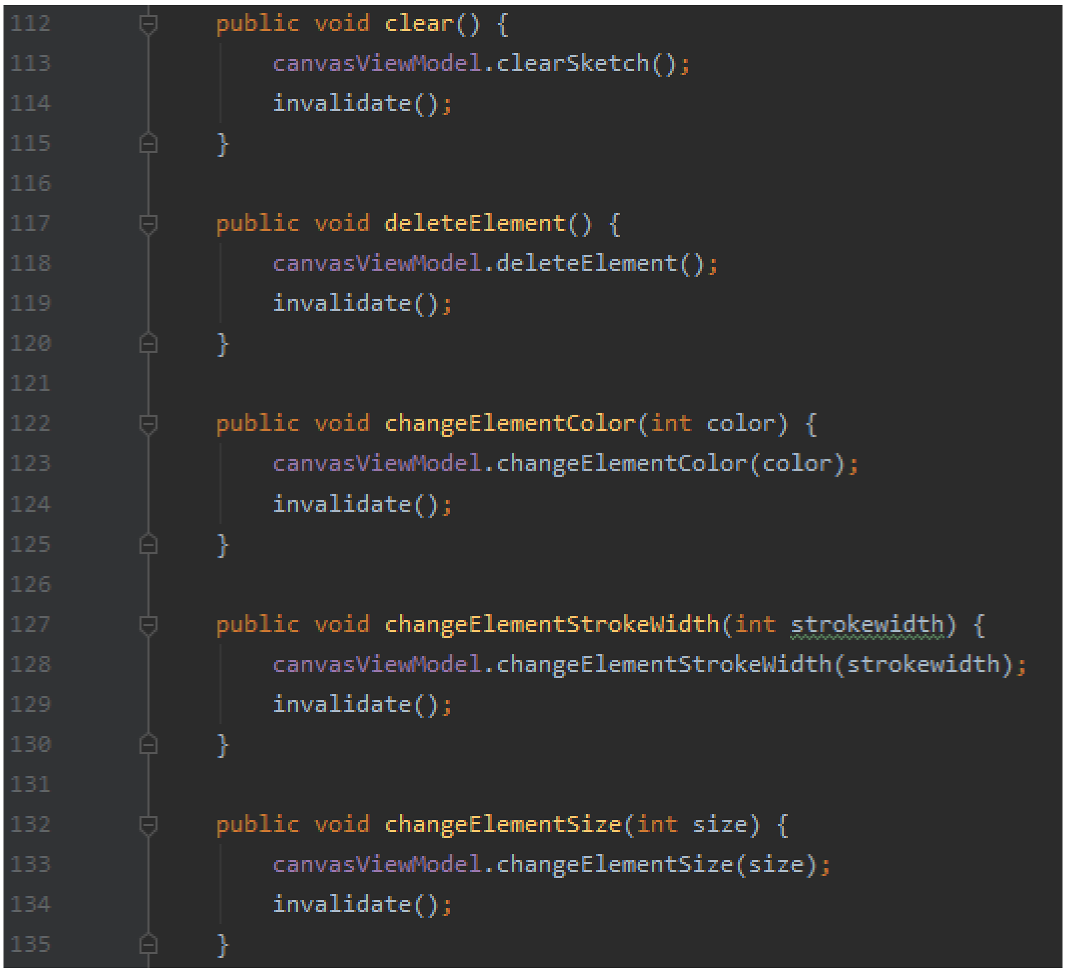
## 1.2 Major Changes Compared to SUPD

The most noticeable differences between our final solution and the one which we handed in for SUPD would be the addition of required functionalities and further implementation of design patterns. However, since these changes are expectable and discussed in section 1.3, we will, at this point, focus on structural changes, design improvements and the incorporation of the SUPD feedback. Most of these aspects, however, have been already mentioned in the design overview section 1.1, describing the process of how we arrived at our final application. In the following, we want to briefly emphasize the key differences, once again.

Our structural rework focused on the correct implementation of the MVVM Model. Method calls are only passed from View to ViewModel to model and not in the other direction. If a change in the logic occurs, the ViewModel is updated via the observer pattern which itself is the base for updates in the view classes, where the actual changes are displayed for the user. Additionally, we paid much more attention to the difference of business logic, presentation logic and code that belongs in View classes. An example for this change would be the development of the "invalidate" method. Earlier we had many different "invalidate" methods in the ViewModel for each of the cases when the canvas had to be updated. This also meant that we had to make calls to the "CanvasView" class directly from the "MainActivity" class. Now, each of these cases is processed in the Model and invokes the "notifyObserver" method which in return calls a single "invalidate method" in the ViewModel.

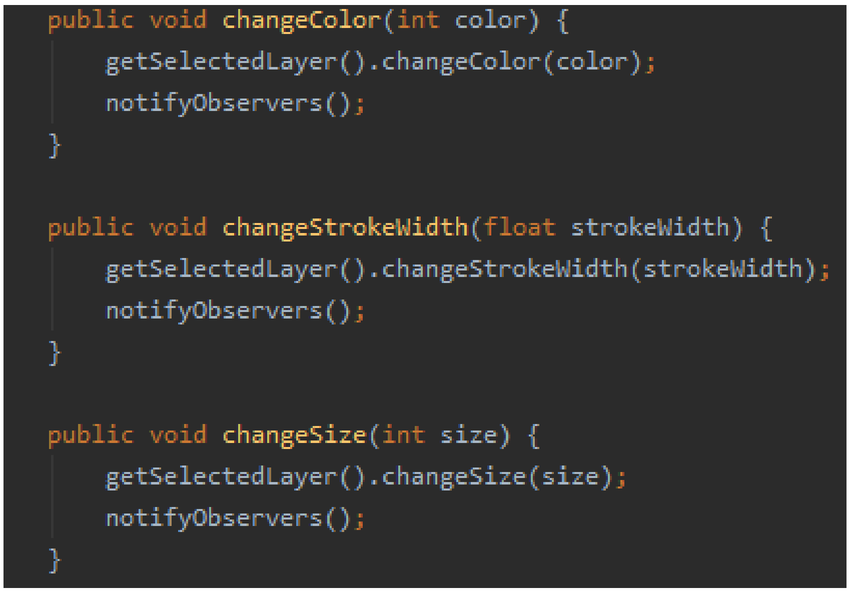
*git: 70a16b8ae5bcd3cb51a4b8801cbb977e84cf1dbf*

*\implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\view\CanvasView.java*



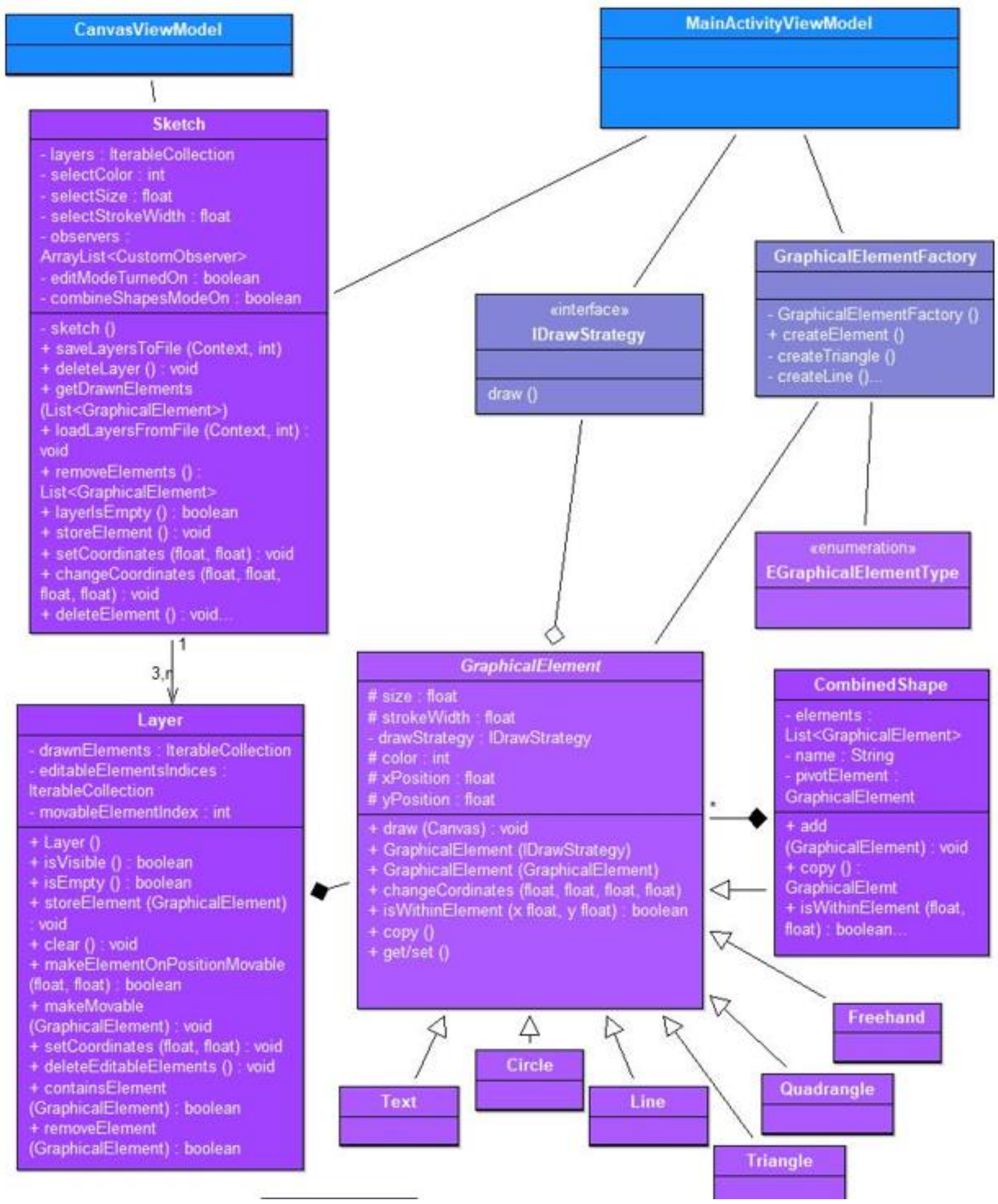
```java
    public void clear() {
        canvasViewModel.clearSketch();
        invalidate();
    }

    public void deleteElement() {
        canvasViewModel.deleteElement();
        invalidate();
    }

    public void changeElementColor(int color) {
        canvasViewModel.changeElementColor(color);
        invalidate();
    }

    public void changeElementStrokeWidth(int strokewidth) {
        canvasViewModel.changeElementStrokeWidth(strokewidth);
        invalidate();
    }

    public void changeElementSize(int size) {
        canvasViewModel.changeElementSize(size);
        invalidate();
    }
```

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)



```java
public void changeColor(int color) {
    getSelectedLayer().changeColor(color);
    notifyObservers();
}

public void changeStrokeWidth(float strokeWidth) {
    getSelectedLayer().changeStrokeWidth(strokeWidth);
    notifyObservers();
}

public void changeSize(int size) {
    getSelectedLayer().changeSize(size);
    notifyObservers();
}
```

Addressing the implementation of layer objects, as stated in section 1.1, this is a major difference to our initial approach. In our SUPD report, we stated: "In addition to this major design decision, we also had to come up with an internal structure for our graphical elements. Our design gravitates around the abstract class "GraphicalElement"." No doubt, the abstract class "GraphicalElement" still is necessary for many operations, its overall relevance in the structure, however, has decreased significantly. Instead, The Model classes "Layer" and "Sketch" (which also is a singleton) are in the centre of most of the important method calls.


CanvasViewModel
MainActivityViewModel
Sketch
- layers : IterableCollection
- selectColor : int
- selectSize : float
- selectStrokeWidth : float
- observers : ArrayList<CustomObserver>
- editModeTurnedOn : boolean
- combineShapesModeOn : boolean
- sketch ()
+ saveLayersToFile (Context, int)
+ deleteLayer () : void
+ getDrawnElements (List<GraphicalElement>)
+ loadLayersFromFile (Context, int) : void
+ removeElements () : List<GraphicalElement>
+ layerIsEmpty () : boolean
+ storeElement () : void
+ setCoordinates (float, float) : void
+ changeCoordinates (float, float, float, float) : void
+ deleteElement () : void...
«interface»
IDrawStrategy
draw ()
GraphicalElementFactory
- GraphicalElementFactory ()
+ createElement ()
- createTriangle ()
- createLine ()...
«enumeration»
EGraphicalElementType
1
3,n
Layer
- drawnElements : IterableCollection
- editableElementsIndices : IterableCollection
- movableElementIndex : int
+ Layer ()
+ isVisible () : boolean
+ isEmpty () : boolean
+ storeElement (GraphicalElement) : void
+ clear () : void
+ makeElementOnPositionMovable (float, float) : boolean
+ makeMovable (GraphicalElement) : void
+ setCoordinates (float, float) : void
+ deleteEditableElements () : void
+ containsElement (GraphicalElement) : boolean
+ removeElement (GraphicalElement) : boolean
GraphicalElement
# size : float
# strokeWidth : float
- drawStrategy : IDrawStrategy
# color : int
# xPosition : float
# yPosition : float
+ draw (Canvas) : void
+ GraphicalElement (IDrawStrategy)
+ GraphicalElement (GraphicalElement)
+ changeCordinates (float, float, float, float)
+ isWithinElement (x float, y float) : boolean
+ copy ()
+ get/set ()
*
CombinedShape
- elements : List<GraphicalElement>
- name : String
- pivotElement : GraphicalElement
+ add (GraphicalElement) : void
+ copy () : GraphicalElemt
+ isWithinElement (float, float) : boolean...
Freehand
Circle
Text
Line
Quadrangle
Triangle

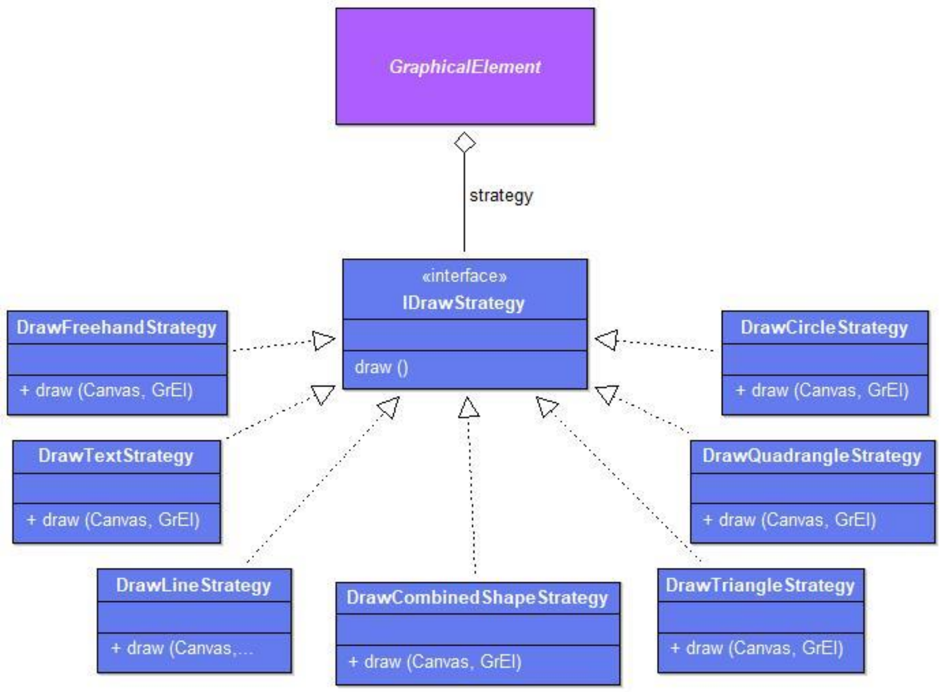
Another difference to SUPD is the movement behavior of graphical elements. In our previous solution when an object is moved, it immediately jumps to the new position. In order to improve the user experience, we decided to add "drag behavior" which offers enhanced controllability of the object's new position. This is achieved by first checking, whether a user touch on the screen falls within a drawn element and, if true, setting this element as movable until the user stops touching the screen.

Lastly, we also incorporated the feedback which was given at SUPD. As described in detail above, from our point of view, the code style in general and the structure of the app has improved. Furthermore, this report should be more elaborate than the one which we handed in for SUPD since we now have a broader understanding of the topic and invested more time in the creation of the report, recognizing its importance. Additionally, the Code snippets for the strategy pattern have been updated.

## 1.3 Design Patterns

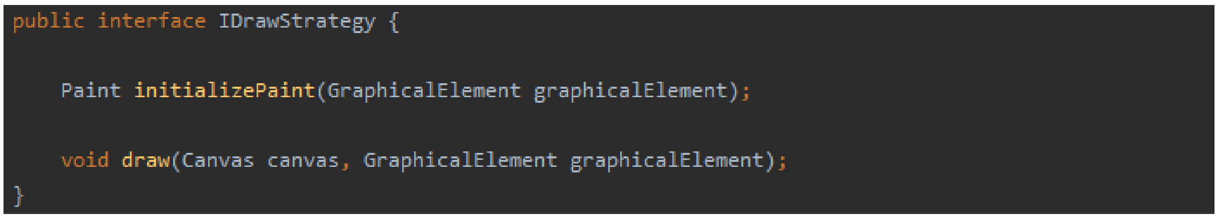
### 1.3.1 Strategy Pattern

The Strategy Pattern is useful when there are multiple algorithms which solve a problem, and they can be used interchangeably, according to the concrete context. It is an alternative to implementing behavioural logic in subclasses of a Context, which has the benefit of separating business logic from Context state and thus making the code easier to read and maintain. Moreover, conditional statements for selecting the behaviour are also avoided thanks to this pattern.



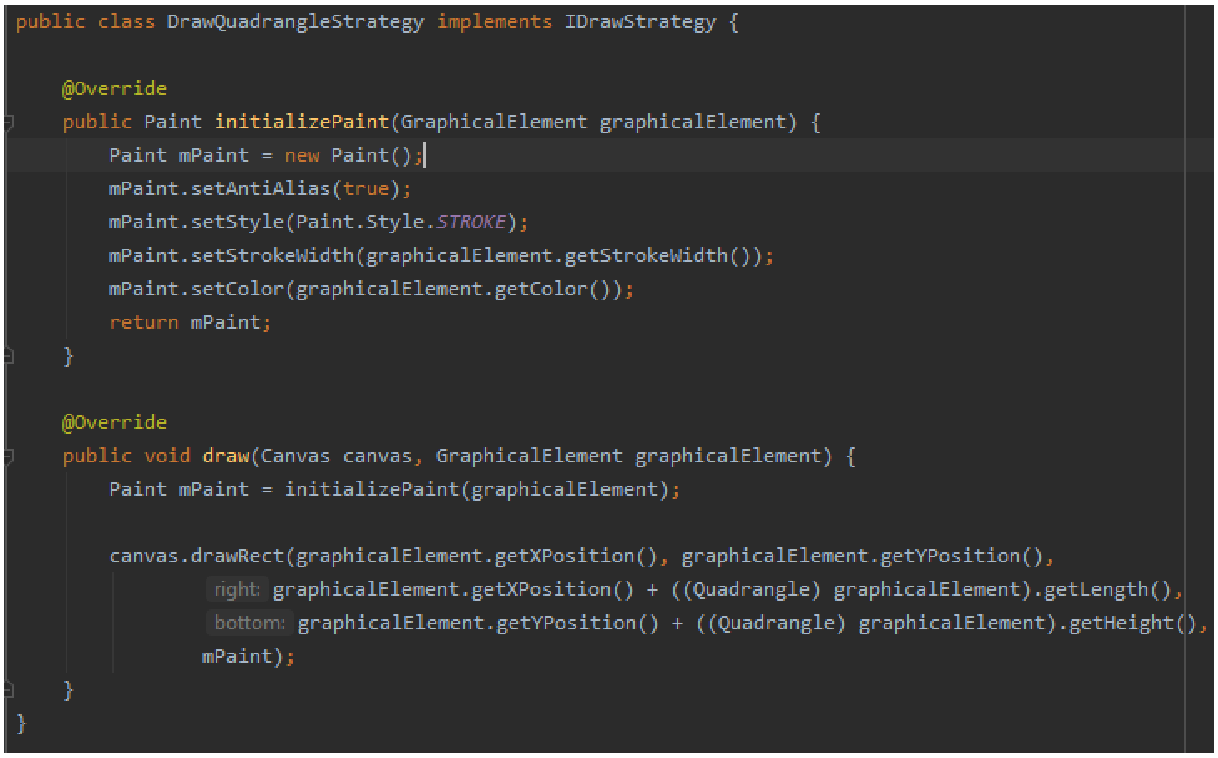
In our project, the Strategy Pattern turned out useful to avoid adding logic for drawing on the Canvas directly in our concrete subclasses of "GraphicalElement". The logic for drawing is different for each graphical element, different Strategies have been implemented.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\draw\IDrawStrategy.java)*



```java
public interface IDrawStrategy {

    Paint initializePaint(GraphicalElement graphicalElement);

    void draw(Canvas canvas, GraphicalElement graphicalElement);
}
```

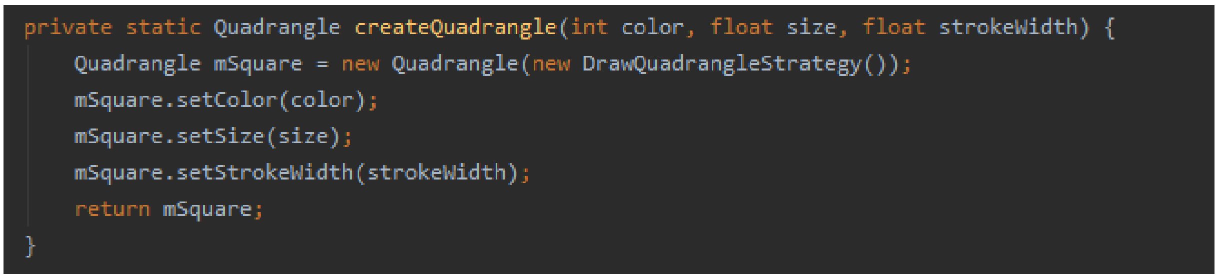
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\draw\DrawQuadrangleStrategy.java)*



```java
public class DrawQuadrangleStrategy implements IDrawStrategy {

    @Override
    public Paint initializePaint(GraphicalElement graphicalElement) {
        Paint mPaint = new Paint();
        mPaint.setAntiAlias(true);
        mPaint.setStyle(Paint.Style.STROKE);
        mPaint.setStrokeWidth(graphicalElement.getStrokeWidth());
        mPaint.setColor(graphicalElement.getColor());
        return mPaint;
    }

    @Override
    public void draw(Canvas canvas, GraphicalElement graphicalElement) {
        Paint mPaint = initializePaint(graphicalElement);

        canvas.drawRect(graphicalElement.getXPosition(), graphicalElement.getYPosition(),
                right: graphicalElement.getXPosition() + ((Quadrangle) graphicalElement).getLength(),
                bottom: graphicalElement.getYPosition() + ((Quadrangle) graphicalElement).getHeight(),
                mPaint);
    }
}
```

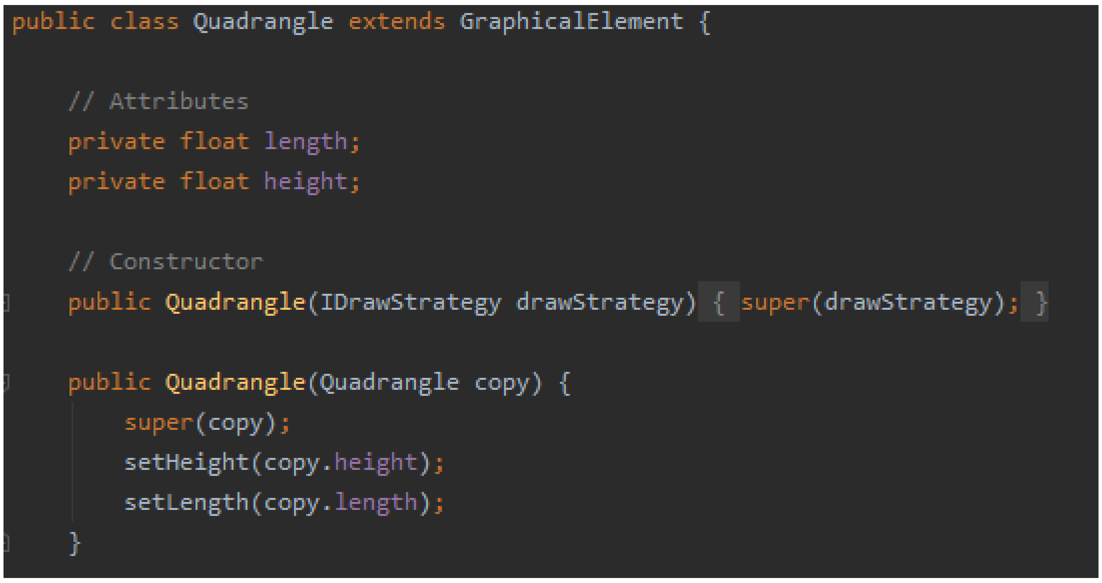
This design pattern relates to the Functional Requirements FR1-FR4, requiring new Graphical Elements to be created and displayed (drawn) on the View. The Strategy pattern is executed when the whole View is being drawn again, by going through each graphical element and invoking the corresponding Strategy object. The drawing is performed by the ViewModel and used to represent the data on the View.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model*\GraphicalElementFactory.java)



```java
private static Quadrangle createQuadrangle(int color, float size, float strokeWidth) {
    Quadrangle mSquare = new Quadrangle(new DrawQuadrangleStrategy());
    mSquare.setColor(color);
    mSquare.setSize(size);
    mSquare.setStrokeWidth(strokeWidth);
    return mSquare;
}
```

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\Quadrangle.java)*



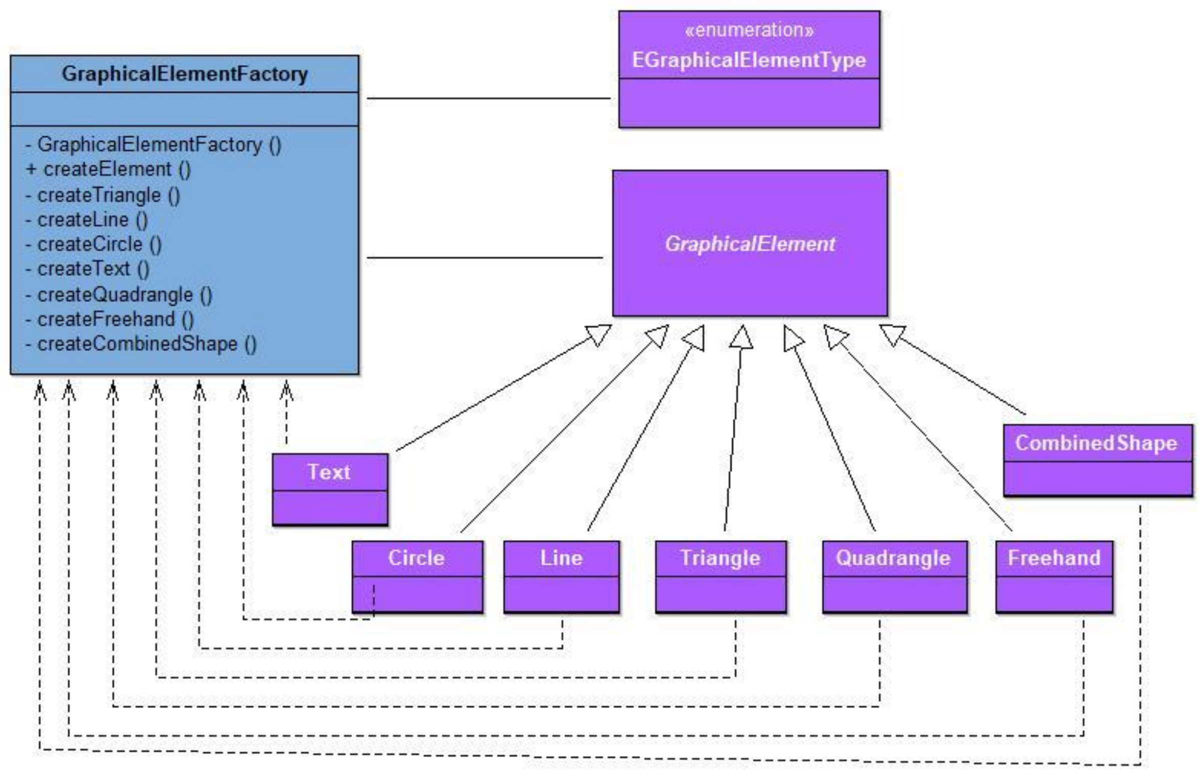
```java
public class Quadrangle extends GraphicalElement {

    // Attributes
    private float length;
    private float height;

    // Constructor
    public Quadrangle(IDrawStrategy drawStrategy) { super(drawStrategy); }

    public Quadrangle(Quadrangle copy) {
        super(copy);
        setHeight(copy.height);
        setLength(copy.length);
    }
```

### 1.3.2 Factory Pattern

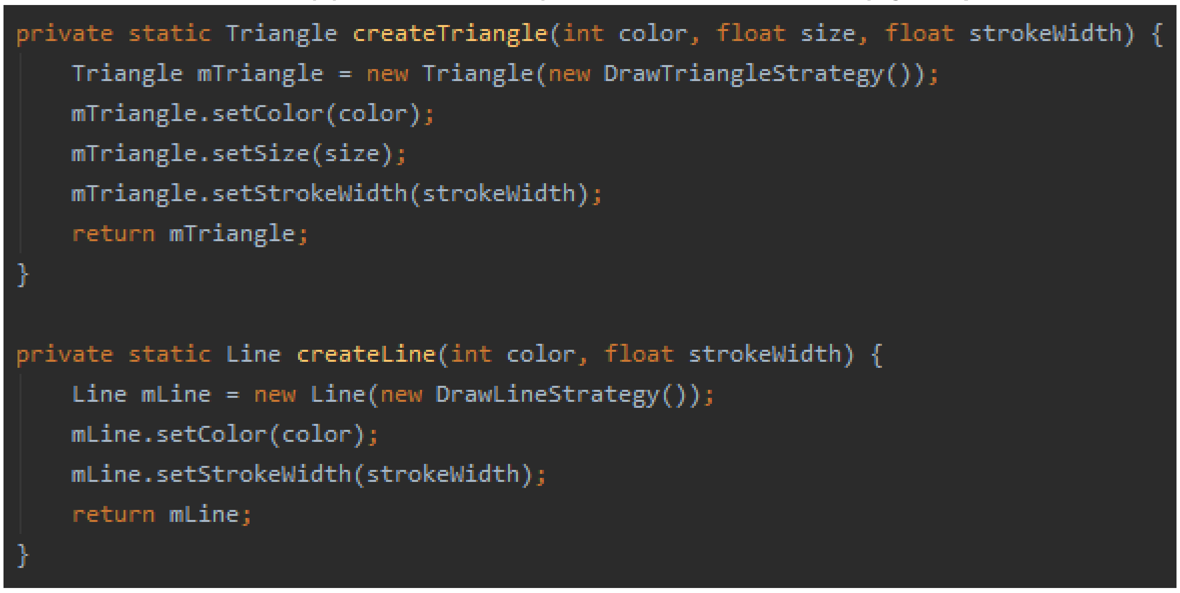
The Factory Method Pattern is useful for delegating the creation of new objects to a dedicated class. This is useful in order to better split the code and avoid bugs. The required attributes and constructor of a concrete object might also change over time and having the creation logic in one determined class makes it easier to address the changes.

In our project we needed to implement the Factory Method pattern to create objects of the “GraphicalElement” type from our Model. These objects are being later displayed on the View.

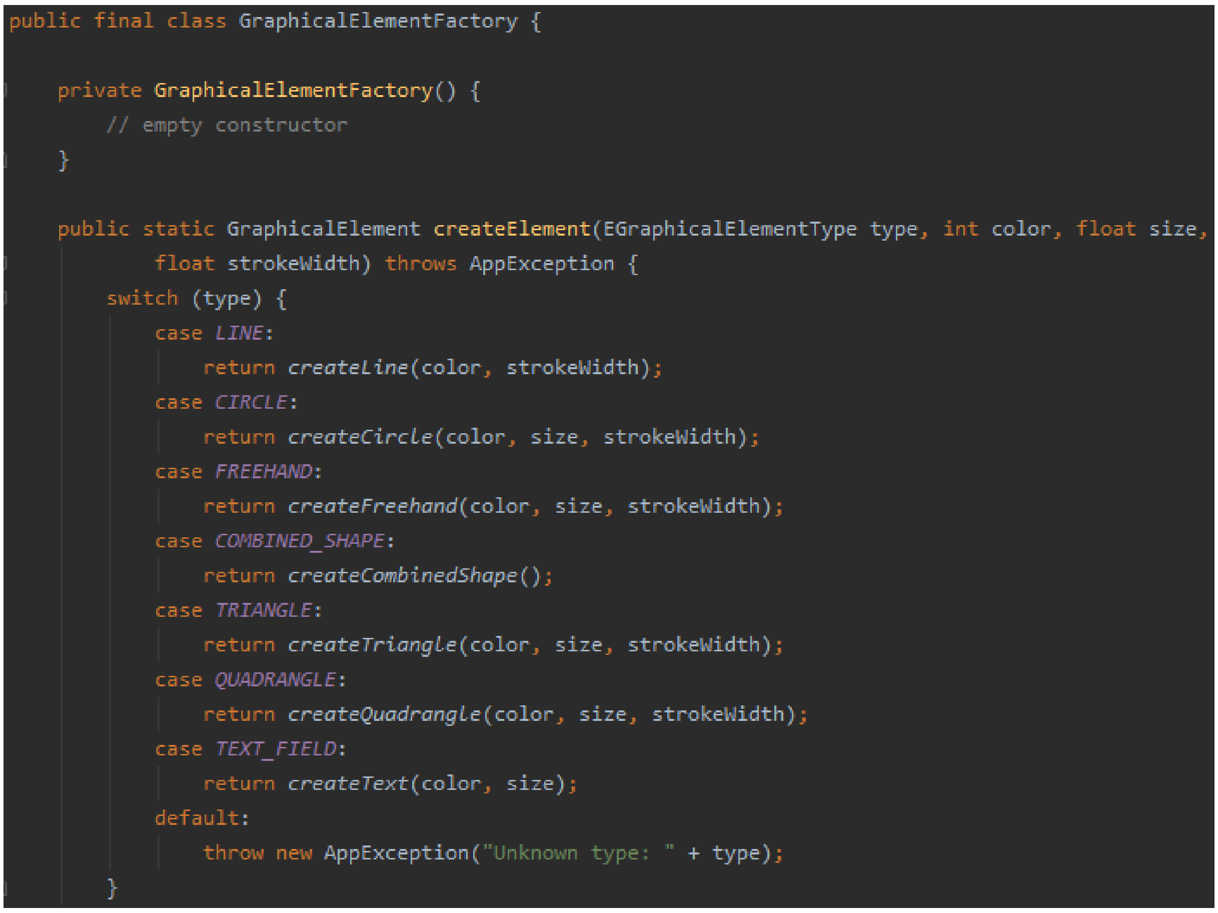
This design pattern relates to the Functional Requirements FR1-FR4, requiring new Graphical Elements to be created and displayed on the View. The creation happens after the user selects a graphical element. The creation is an intermediary step between the data saved in the Model and the representation of the data in the View.



*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model*\GraphicalElementFactory.java)



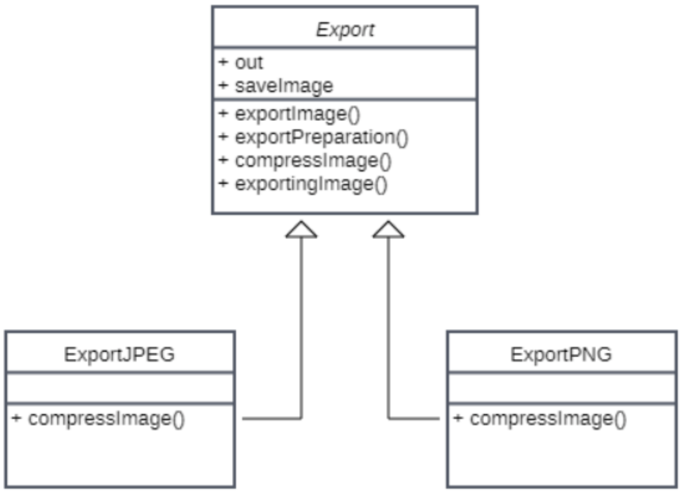
```
private static Triangle createTriangle(int color, float size, float strokeWidth) {
    Triangle mTriangle = new Triangle(new DrawTriangleStrategy());
    mTriangle.setColor(color);
    mTriangle.setSize(size);
    mTriangle.setStrokeWidth(strokeWidth);
    return mTriangle;
}

private static Line createLine(int color, float strokeWidth) {
    Line mLine = new Line(new DrawLineStrategy());
    mLine.setColor(color);
    mLine.setStrokeWidth(strokeWidth);
    return mLine;
}
```



```java
public final class GraphicalElementFactory {

    private GraphicalElementFactory() {
        // empty constructor
    }

    public static GraphicalElement createElement(EGraphicalElementType type, int color, float size,
            float strokeWidth) throws AppException {
        switch (type) {
            case LINE:
                return createLine(color, strokeWidth);
            case CIRCLE:
                return createCircle(color, size, strokeWidth);
            case FREEHAND:
                return createFreehand(color, size, strokeWidth);
            case COMBINED_SHAPE:
                return createCombinedShape();
            case TRIANGLE:
                return createTriangle(color, size, strokeWidth);
            case QUADRANGLE:
                return createQuadrangle(color, size, strokeWidth);
            case TEXT_FIELD:
                return createText(color, size);
            default:
                throw new AppException("Unknown type: " + type);
        }
```

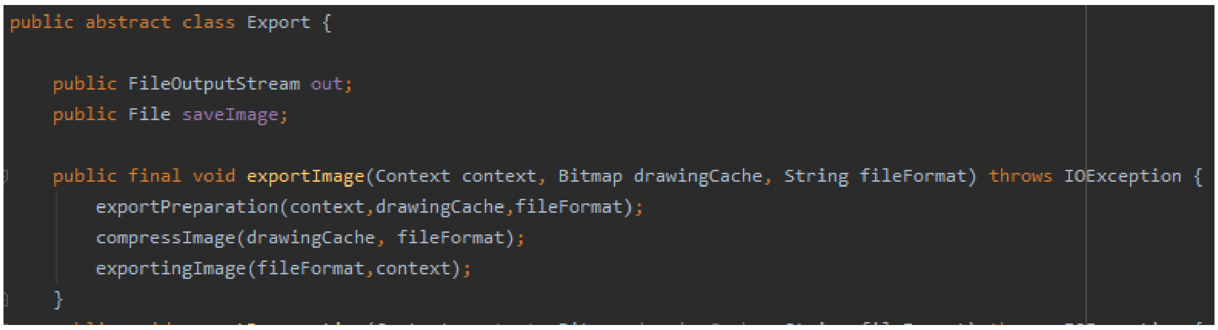
### 1.3.3 Template Methode Pattern

The template method pattern defines a guide how a certain set of methods should be carried out and distributes the individual tasks to itself and more specific subclasses. It consists of an abstract class which describes a template for a process, by declaring a final method with predefined functional steps. These steps can either be ordinary functions in the abstract parent class or they can be abstract functions which are filled with a specific body in the subclasses. Which path is chosen, depends on the nature of the function. If the step is the same for each subclass, only the parent class fills the function with logic. If the process is different for each subclass, the abstract function is overridden and specifically implemented in the subclass differently. Furthermore, the template method employs the "Hollywood principle". Concrete subclasses or only called by the parent class when they are needed for an implementation of a function. Hence, subclasses are only utilized to provide implementation details.



In our implementation the template method is used to define a general approach for exporting files. This approach slightly differs for the concrete export of JPEG-files and PNG-files.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020t eam0310\sketch_app\model\export\Export*.java)



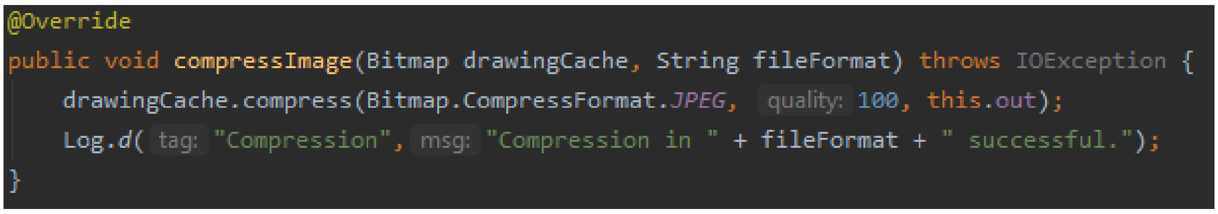
```
public abstract class Export {

    public FileOutputStream out;
    public File saveImage;

    public final void exportImage(Context context, Bitmap drawingCache, String fileFormat) throws IOException {
        exportPreparation(context,drawingCache,fileFormat);
        compressImage(drawingCache, fileFormat);
        exportingImage(fileFormat,context);
    }
```

We created the abstract class “Export” which inherits its properties to its children classes: “ExportJPEG” and “ExportPNG”. Furthermore, we divided the export process in three steps: “exportImage”, “compressImage” and “exportingImage”.

Only the “compressImage” function varies which is why it is declared abstract in the “Export” class and then implemented in the children classes specifically. The other two functions are not implemented in the children classes.

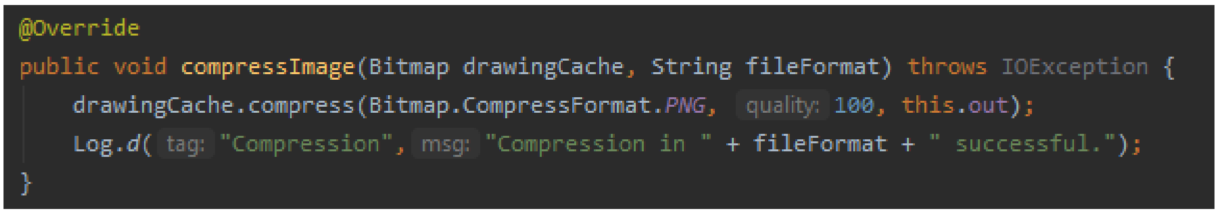
In our solution, the use of this design pattern is related to FR 9 and simplifies the export of files, for which the approach is similar, but there are still small differences in the implementation.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\export\ExportJPEG*.java)



```java
@Override
public void compressImage(Bitmap drawingCache, String fileFormat) throws IOException {
    drawingCache.compress(Bitmap.CompressFormat.JPEG, quality: 100, this.out);
    Log.d( tag: "Compression", msg: "Compression in " + fileFormat + " successful.");
}
```

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\export\ExportPNG*.java)



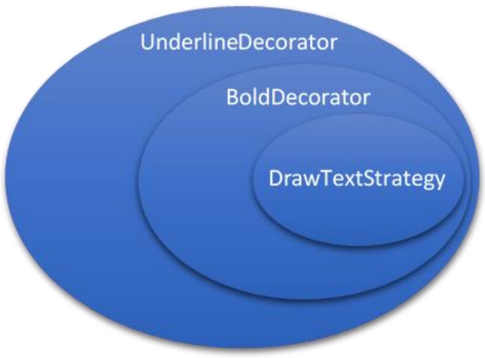
```java
@Override
public void compressImage(Bitmap drawingCache, String fileFormat) throws IOException {
    drawingCache.compress(Bitmap.CompressFormat.PNG, quality: 100, this.out);
    Log.d( tag: "Compression", msg: "Compression in " + fileFormat + " successful.");
}
```

Thus, there is no duplicate code and the general approach to the export of files remains unchanged. Furthermore, the client has no contact to the subclasses, since they are only called by the abstract class itself, resulting in well encapsulated algorithms.

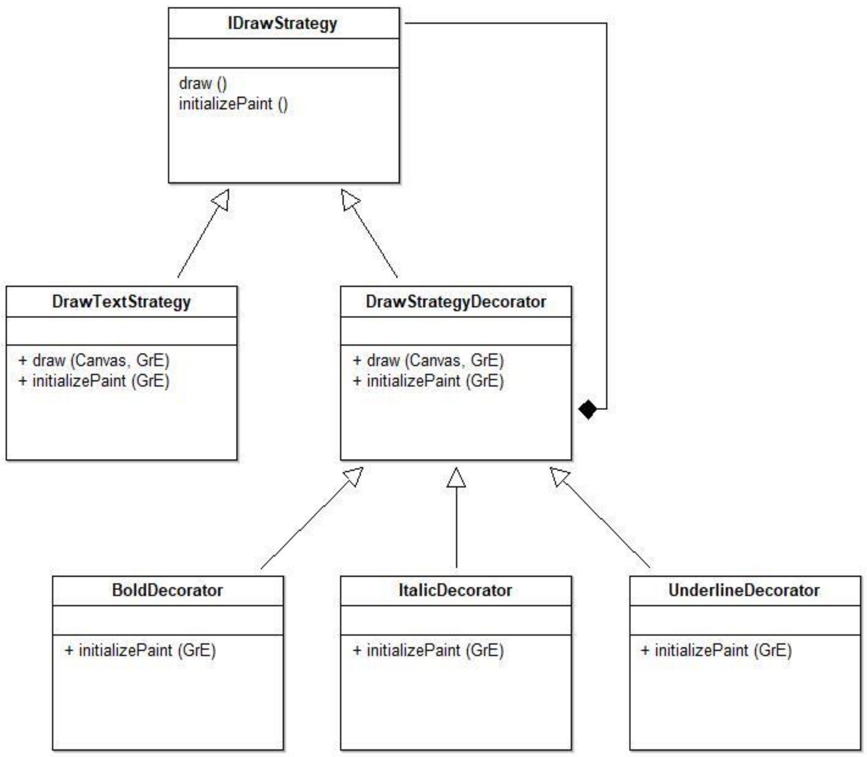
### 1.3.4 Decorator Pattern

With the Decorator pattern, new functionalities can be added dynamically during runtime. Classes can be extended by "wrapping" them in any number of decorator classes. For this purpose, the class to be decorated and the decorator classes inherit from the same supertype.

We decided to use the decorator pattern to decorate text objects with different styles. For example, a DrawTextStrategy can be "wrapped" with the BoldDecorator and then the result can be “wrapped” with the UnderlineDecorator.

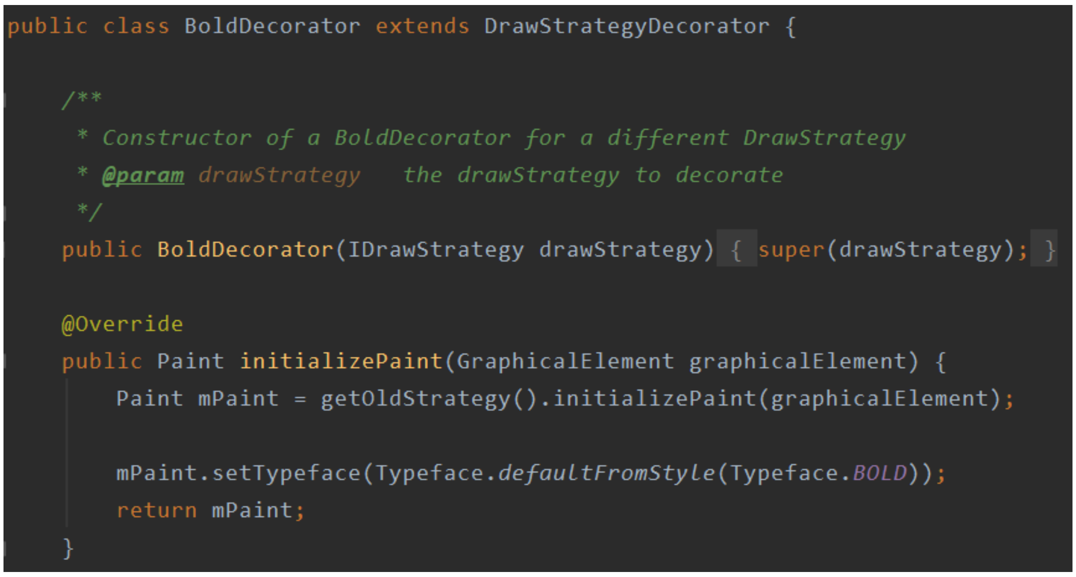


The "DrawTextStrategy" class, i.e., the class to create text, and the abstract "DrawStrategyDecorator" class both have the interface "IDrawStrategy" as a supertype. The concrete decorator classes in our project are "BoldDecorator", "ItalicDecorator" and the "UnderlineDecorator".



The "DrawStrategyDecorator" holds an attribute oldstrategy, which stores the drawStrategy that is currently in use. The concrete decorator classes, for example "BoldDecorator" then call the initializePaint method of the drawStrategy and add the desired functionality, here for example bold text.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\decorators\BoldDecorator.java):*

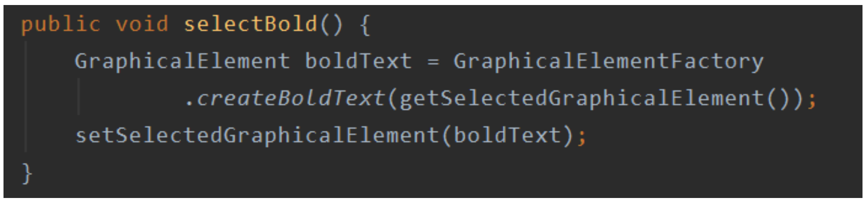


```
public class BoldDecorator extends DrawStrategyDecorator {

    /**
     * Constructor of a BoldDecorator for a different DrawStrategy
     * @param drawStrategy   the drawStrategy to decorate
     */
    public BoldDecorator(IDrawStrategy drawStrategy) { super(drawStrategy); }

    @Override
    public Paint initializePaint(GraphicalElement graphicalElement) {
        Paint mPaint = getOldStrategy().initializePaint(graphicalElement);

        mPaint.setTypeface(Typeface.defaultFromStyle(Typeface.BOLD));
        return mPaint;
    }
```

This explains the underlying logic.

The Decorator classes in our app are called up via buttons. If the user selects the text element, three additional buttons are displayed with which you can select the text styles "italic", "bold" and "underlined". In the code, this happens in the MainActivity. From here it is forwarded to the MainActivityViewModel. The methods "onClickItalicButton", "onClickBoldButton" and "onClickUnderlineButton" call methods in the class Sketch. For example, the selectBold method.

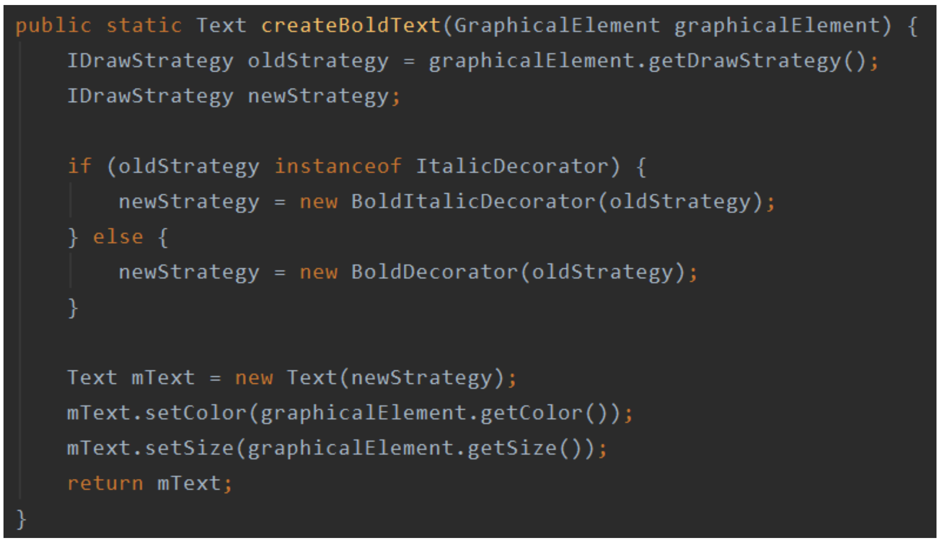
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch.java):*



```
public void selectBold() {
    GraphicalElement boldText = GraphicalElementFactory
            .createBoldText(getSelectedGraphicalElement());
    setSelectedGraphicalElement(boldText);
}
```

These methods then call "create" methods in the "GraphicalElementFactory", such as the "createBoldText" method here.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\GraphicalElementFactory.java):*

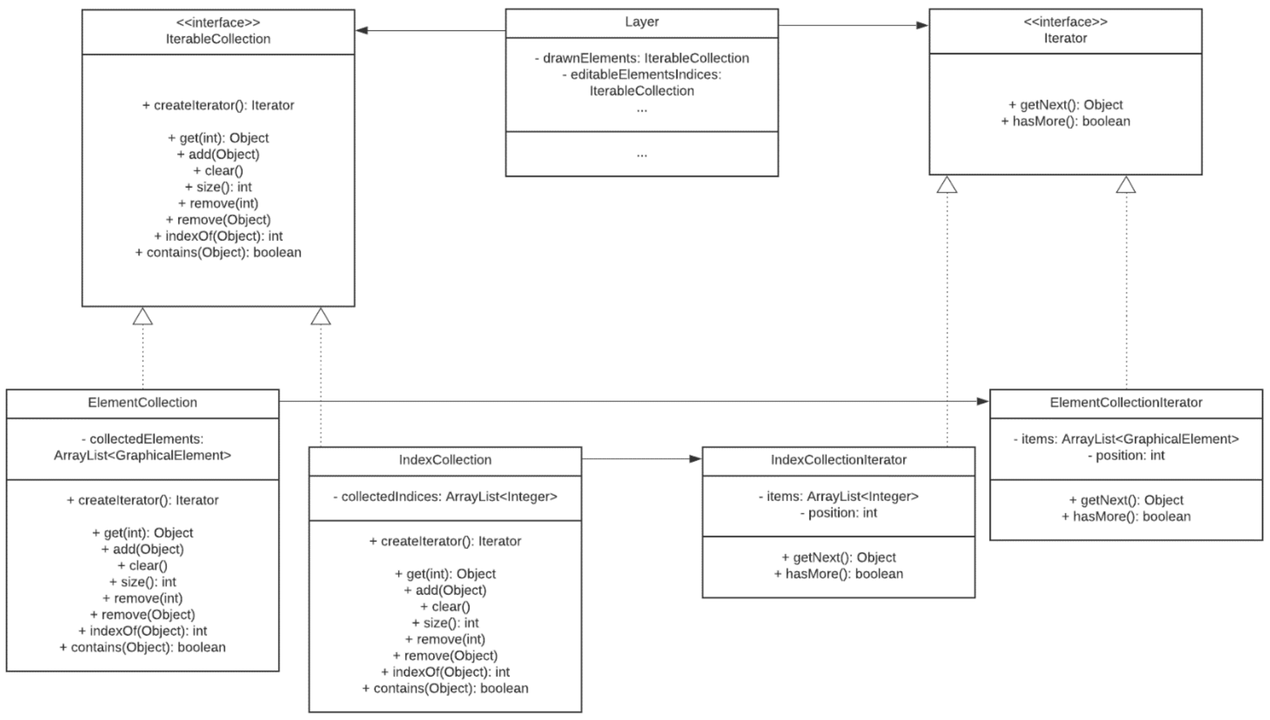


```
public static Text createBoldText(GraphicalElement graphicalElement) {
    IDrawStrategy oldStrategy = graphicalElement.getDrawStrategy();
    IDrawStrategy newStrategy;

    if (oldStrategy instanceof ItalicDecorator) {
        newStrategy = new BoldItalicDecorator(oldStrategy);
    } else {
        newStrategy = new BoldDecorator(oldStrategy);
    }

    Text mText = new Text(newStrategy);
    mText.setColor(graphicalElement.getColor());
    mText.setSize(graphicalElement.getSize());
    return mText;
}
```

And here the respective decorator is finally constructed to create a decorated text element.

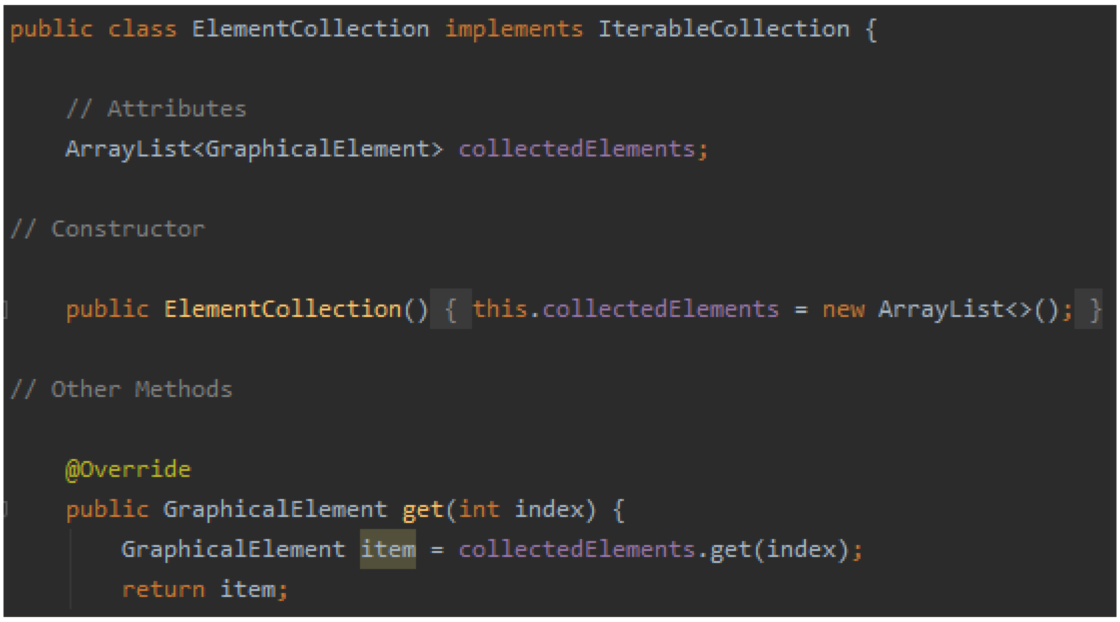
### 1.3.5 Iterator Pattern

The iterator pattern is used to iterate over any kind of data collection without exposing what kind of collection it is currently traversing. The iterator pattern relies on two interfaces. Firstly, the iterator interface is created which has all the required methods for traversing over a data structure. This interface must be implemented by all concrete iterator classes, so they can traverse over a given data structure. Secondly, the iterableCollection interface has to be created as well. This interface describes the methods which are necessary to call a concrete iterator for the data collection. The concrete subclasses that implement the iterableCollection interface, oversee the instantiation of a fitting iterator for the data collection at hand.



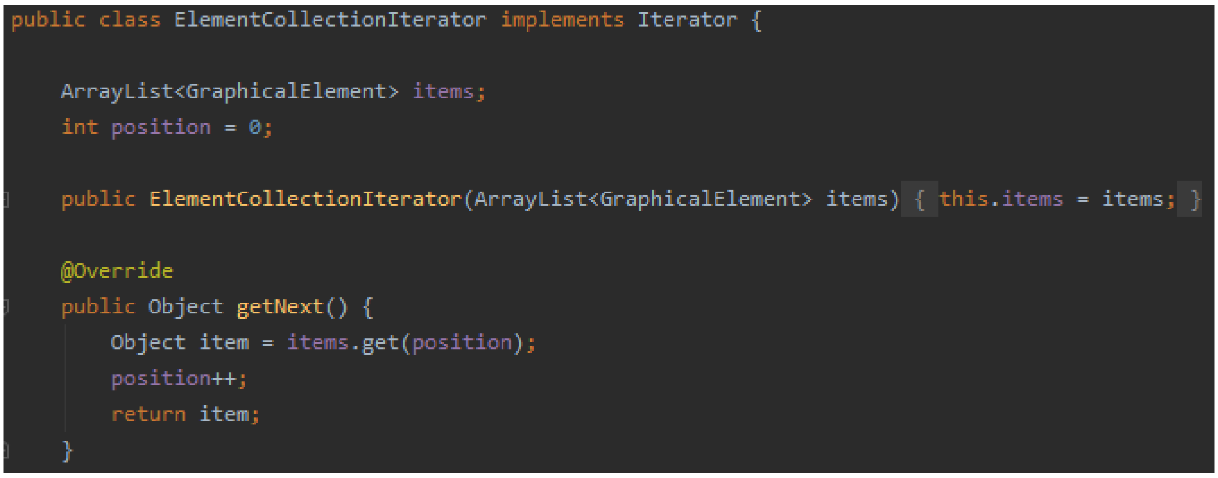
Our sketching app has several implementations of the iterator pattern. Most importantly, it is used to traverse over an array list which holds all graphical elements that are shown on a layer object. As one can see in the above UML-diagram this iteration process is closely connected to an iteration of an index collection which is used to make one or more selected graphical elements editable (FR 4 & FR7). However, since we are only supposed to focus on two instances of the pattern, we would like to disregard this for now.

(*implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\ElementCollection*.java)



```java
public class ElementCollection implements IterableCollection {

    // Attributes
    ArrayList<GraphicalElement> collectedElements;

// Constructor

    public ElementCollection() { this.collectedElements = new ArrayList<>(); }

// Other Methods

    @Override
    public GraphicalElement get(int index) {
        GraphicalElement item = collectedElements.get(index);
        return item;
```

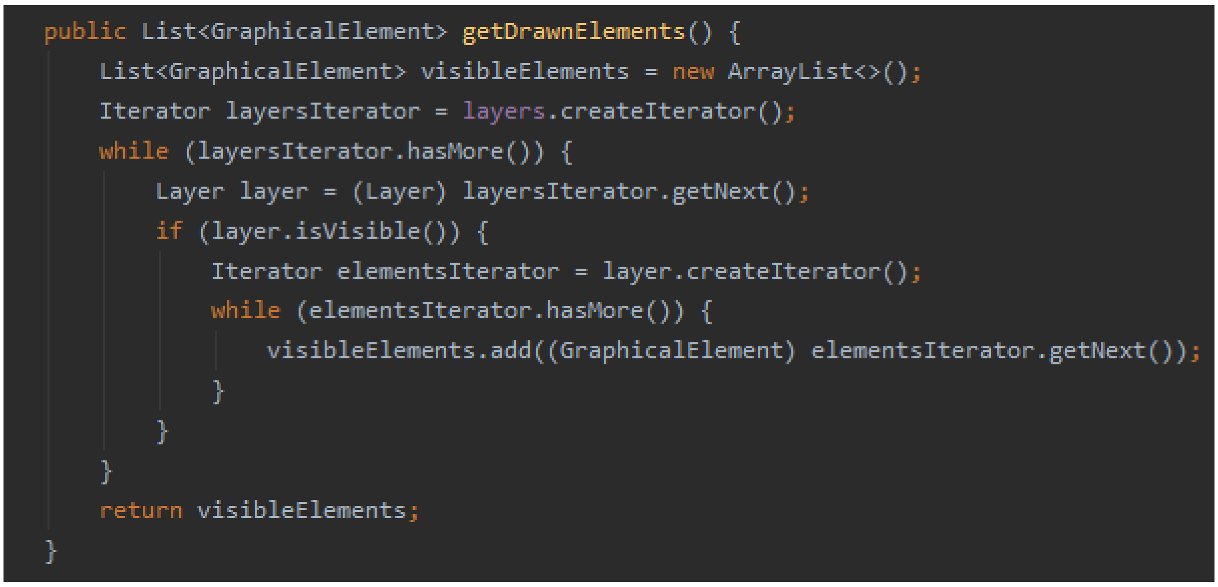
(*implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\ElementCollectionIterator*.java)



```java
public class ElementCollectionIterator implements Iterator {

    ArrayList<GraphicalElement> items;
    int position = 0;

    public ElementCollectionIterator(ArrayList<GraphicalElement> items) { this.items = items; }

    @Override
    public Object getNext() {
        Object item = items.get(position);
        position++;
        return item;
    }
```

An iterable collection “ElementCollection is created, extending the interface “IterableCollection”. This class calls the concrete iterator “ElementCollectionIterator” which extends the interface “Iterator” and is able to traverse over array lists, as long as the array list contains a next object.

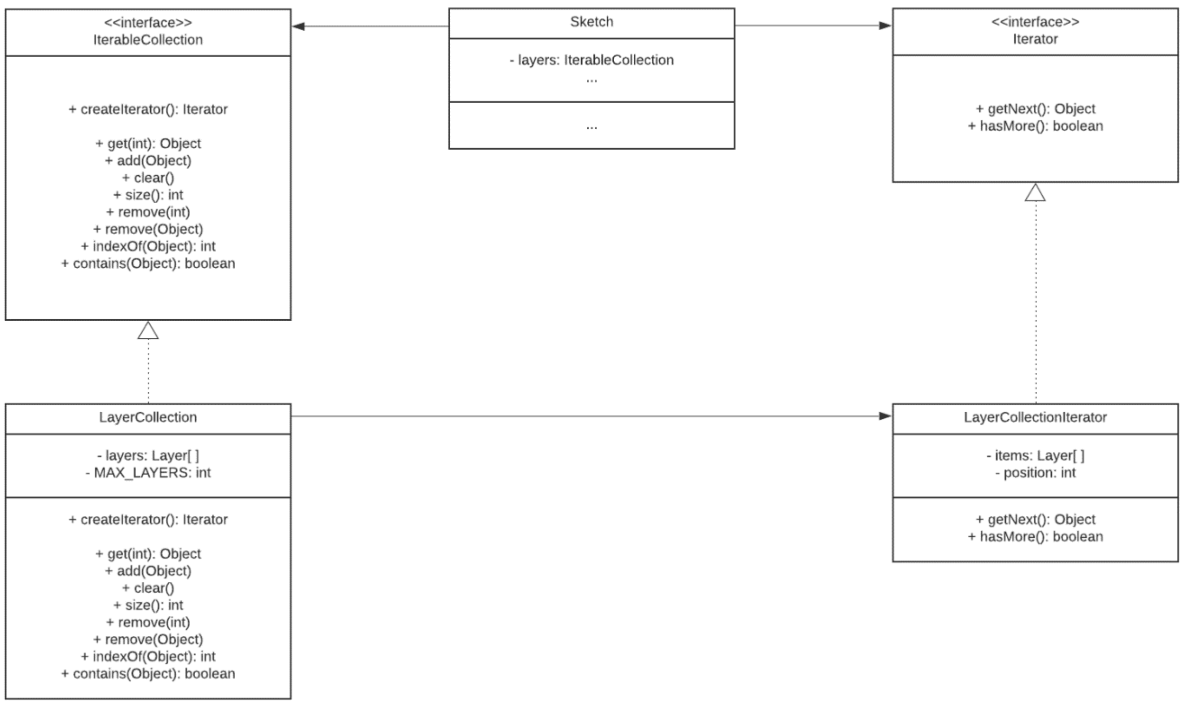
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)



```java
public List<GraphicalElement> getDrawnElements() {
    List<GraphicalElement> visibleElements = new ArrayList<>();
    Iterator layersIterator = layers.createIterator();
    while (layersIterator.hasMore()) {
        Layer layer = (Layer) layersIterator.getNext();
        if (layer.isVisible()) {
            Iterator elementsIterator = layer.createIterator();
            while (elementsIterator.hasMore()) {
                visibleElements.add((GraphicalElement) elementsIterator.getNext());
            }
        }
    }
    return visibleElements;
}
```

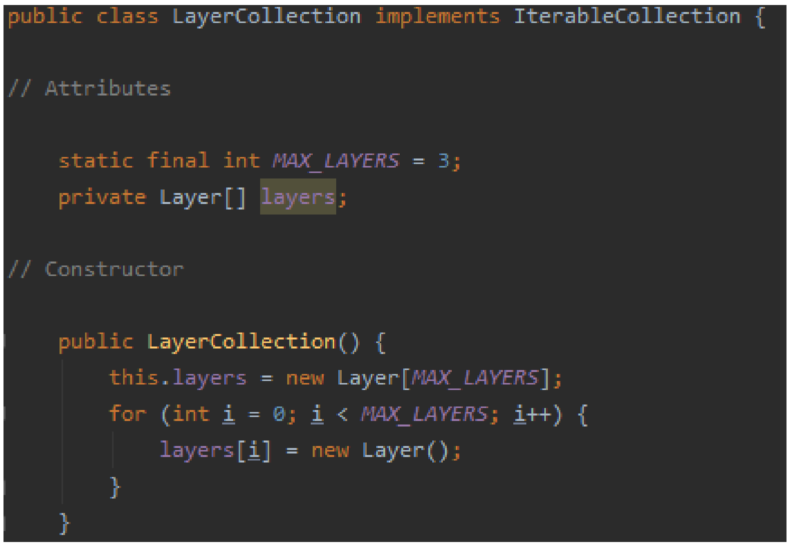
The application of iterator pattern relates to FR1 - FR3, since it enables us to easily display all of our graphical elements (including freehand drawing and text) on the screen.

the iterator pattern lets us traverse over our different data collections, without having to treat them differently. This is especially useful, since there are several kinds of different data collections that are being traversed. The iteration process always looks the same and complex traversal methods are encapsulated in concrete classes. Also, the iterator pattern provides us with the option to easily exchange our data collections in the future if it should be necessary. For example, if we would come to the conclusion that the user only should be able to create a limited number of graphical elements on a layer object, we would change the data structure in the "ElementCollection" to an array of a fixed size for better performance. Similarly, we would update the responsible iterator in the class "ElementCollectionIterator", so it would be able to travers an array, instead of an array list. The logic in the classes which rely on the output of the iterator could remain unchanged, which is a major benefit.



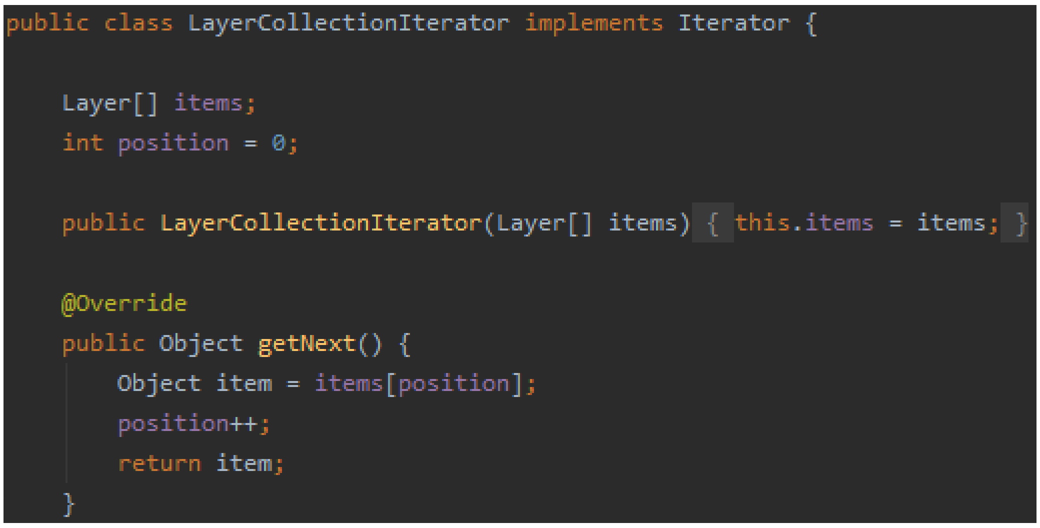
AS previously mentioned, we want to present the "LayerCollectionIterator" as our second instance of this pattern. The iterator pattern is implemented to travers over an array which stores our three layers. We implement the pattern by creating the "LayerCollection" and the "LayerCollectionIterator" analogous to our previous approach for concrete graphical elements. The Main difference is that the class "LayerCollection holds a fixed array of the size three (one for each layer). Hence, the "LayerCollectionIterator" is programmed to travers over an array instead of an array list.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\LayerCollection*.java)



```
public class LayerCollection implements IterableCollection {

// Attributes

    static final int MAX_LAYERS = 3;
    private Layer[] layers;

// Constructor

    public LayerCollection() {
        this.layers = new Layer[MAX_LAYERS];
        for (int i = 0; i < MAX_LAYERS; i++) {
            layers[i] = new Layer();
        }
    }
```

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\LayerCollectionIterator*.java)



```java
public class LayerCollectionIterator implements Iterator {

    Layer[] items;
    int position = 0;

    public LayerCollectionIterator(Layer[] items) { this.items = items; }

    @Override
    public Object getNext() {
        Object item = items[position];
        position++;
        return item;
    }
```

This instance of the iterator pattern is related to FR 5. Depending on how many of layer objects are activated, they are used to construct our sketch object which holds all objects that the user eventually sees on the screen. In other words, the iterator is responsible for displaying all graphical elements on the screen, depending on how many layers are selected.

The improvements which the iterator pattern offers are on a conceptual level identical as we described it for the "ElementCollectionIterator".

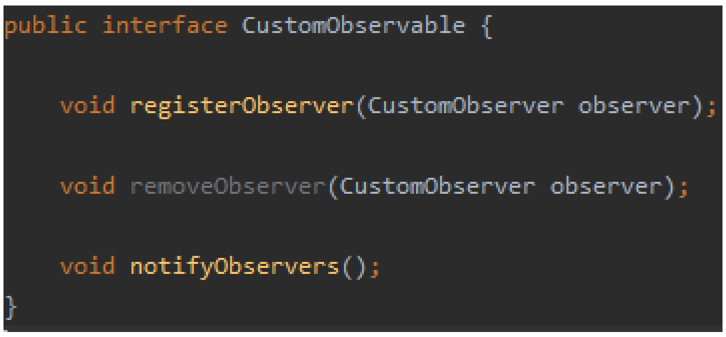
Lasty, we also make use of the iterator pattern, in order to create combined shapes (FR 6).

### 1.3.6 Observer Pattern

The observer pattern describes a one-to-many relationship between at least two classes. The relationship is characterized by class A-X (the observers) being completely dependent on class Y (the subject). For the concept to work properly, it is necessary to implement two different interfaces. One which is implemented by a concrete subject with the required methods of registering, removing and notifying observers. The other interface is later realized by classes that observe the concrete subject. Once this relationship is set up and the state of the concrete subject class changes, the concrete observer classes are updated.

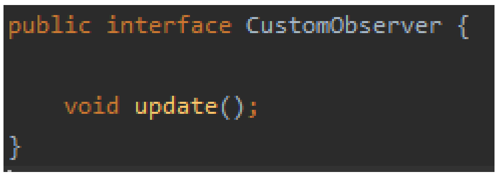
We utilize this design pattern by creating the two required interfaces and later implementing them into the concrete classes which pose as subject and observer.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\observerPatternInterfaces\CustomObservable*.java)



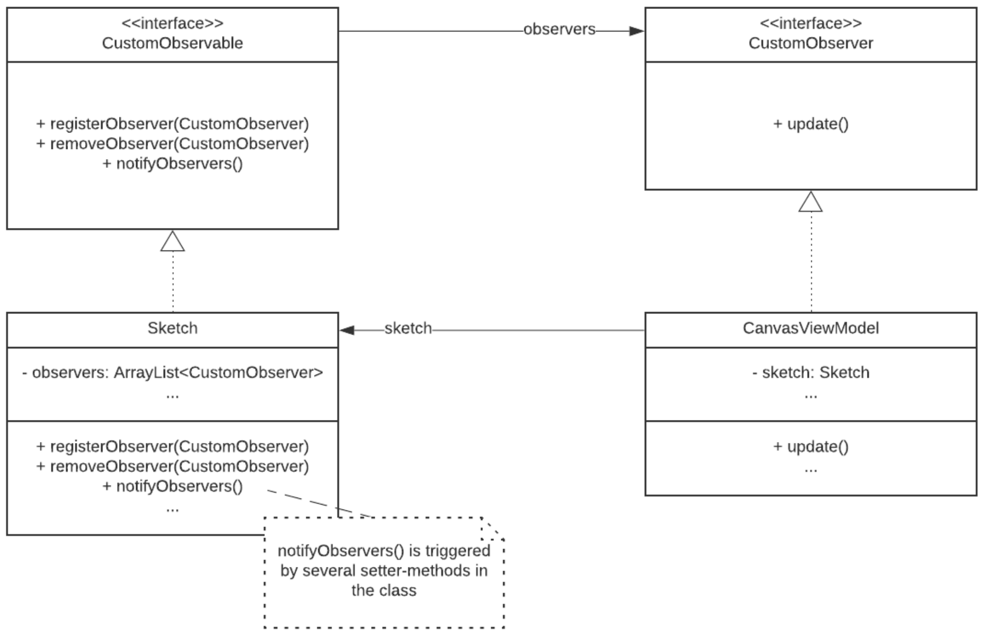
```
public interface CustomObservable {

    void registerObserver(CustomObserver observer);

    void removeObserver(CustomObserver observer);

    void notifyObservers();
}
```

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\iteratorsAndCollections\observerPatternInterfaces\CustomObserver*.java)



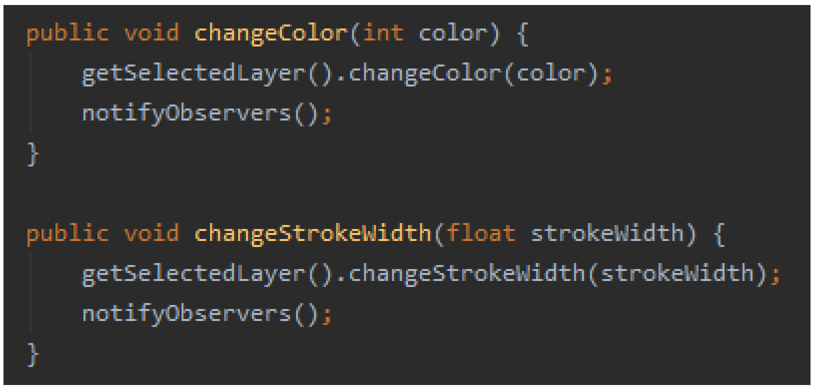
```
public interface CustomObserver {

    void update();
}
```

The interface "CustomObservable" acts as subject-interface and the interface "CustomObserver" as observer-interface. Both contain the methods that were discussed in the previous section. We have two occurrences of the observer pattern which are closely connected and are therefore, described as one. Both play an important role in realizing the overall structure of our application – The Model-View-ViewModel approach. Foremost, let us have a look at the relationship from Model to ViewModel.


<<interface>>
CustomObservable
+ registerObserver(CustomObserver)
+ removeObserver(CustomObserver)
+ notifyObservers()
observers
<<interface>>
CustomObserver
+ update()
Sketch
- observers: ArrayList<CustomObserver>
...
+ registerObserver(CustomObserver)
+ removeObserver(CustomObserver)
+ notifyObservers()
...
sketch
CanvasViewModel
- sketch: Sketch
...
+ update()
...
notifyObservers() is triggered by several setter-methods in the class

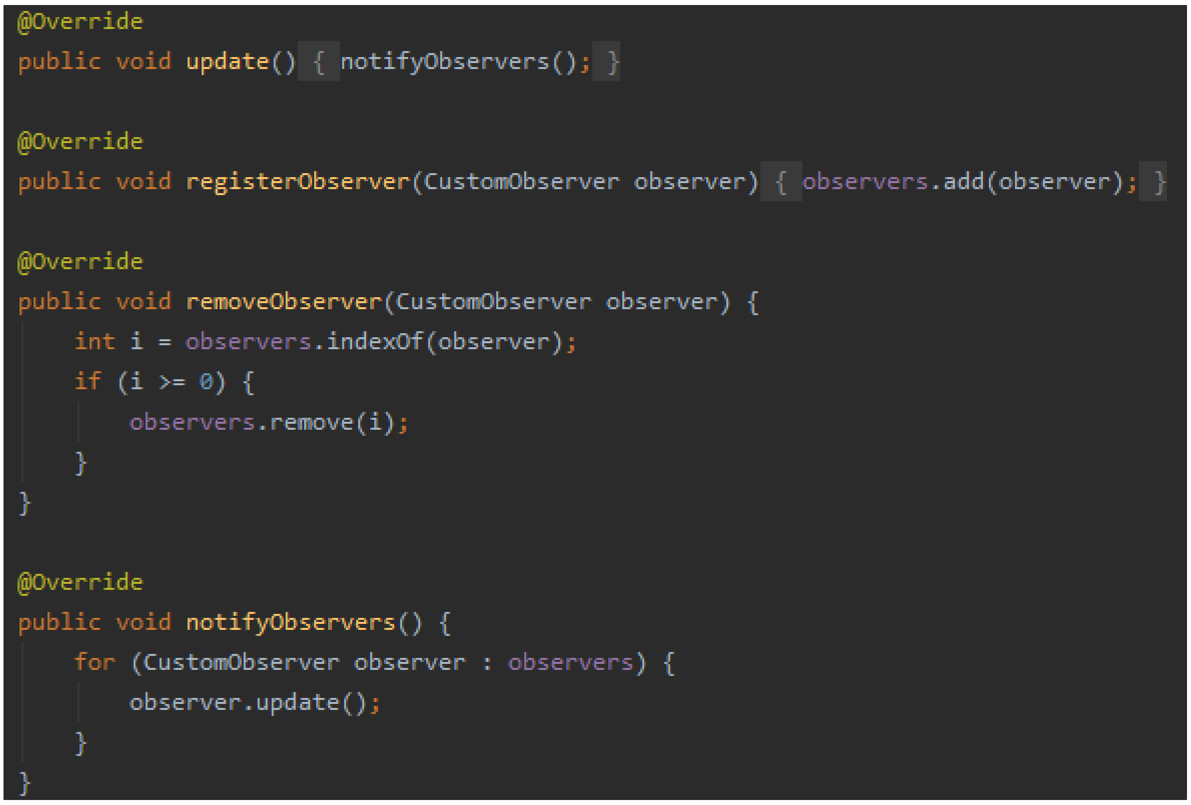
The "CustomObservable" interface is implemented in the "Sketch" class which enables other classes to register to the subject and receive updates, if the "notifyObserver" method is invoked. This opportunity is seized by the "CanvasViewModel" class. Whenever the "notifyObserver" method is called, the "update" method in "CanvasViewModel" is run.

(*implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)

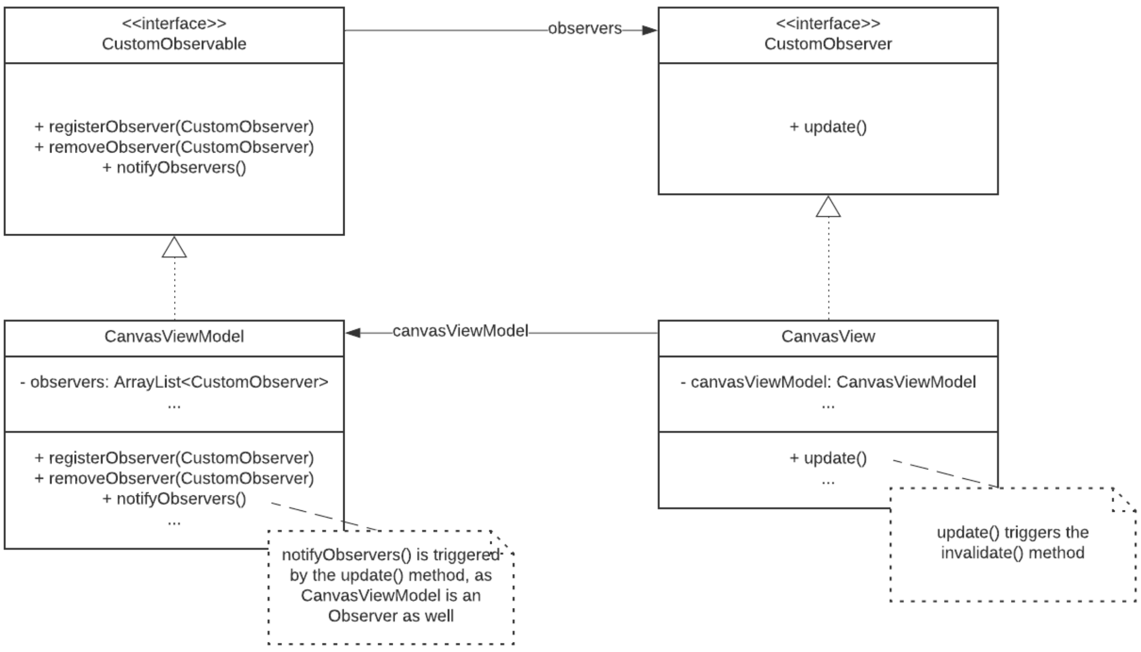


```java
public void changeColor(int color) {
    getSelectedLayer().changeColor(color);
    notifyObservers();
}

public void changeStrokeWidth(float strokeWidth) {
    getSelectedLayer().changeStrokeWidth(strokeWidth);
    notifyObservers();
}
```

(*implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\viewModel\CanvasViewModel*.java)

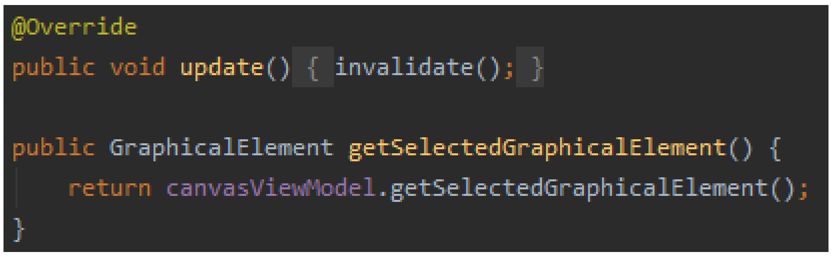


```java
@Override
public void update() { notifyObservers(); }

@Override
public void registerObserver(CustomObserver observer) { observers.add(observer); }

@Override
public void removeObserver(CustomObserver observer) {
    int i = observers.indexOf(observer);
    if (i >= 0) {
        observers.remove(i);
    }
}

@Override
public void notifyObservers() {
    for (CustomObserver observer : observers) {
        observer.update();
    }
}
```


<<interface>>
CustomObservable
+ registerObserver(CustomObserver)
+ removeObserver(CustomObserver)
+ notifyObservers()
observers
<<interface>>
CustomObserver
+ update()
CanvasViewModel
- observers: ArrayList<CustomObserver>
...
+ registerObserver(CustomObserver)
+ removeObserver(CustomObserver)
+ notifyObservers()
...
canvasViewModel
CanvasView
- canvasViewModel: CanvasViewModel
...
+ update()
...
notifyObservers() is triggered by the update() method, as CanvasViewModel is an Observer as well
update() triggers the invalidate() method

This is where the second application of the observer pattern comes into play. Our "CanvasViewModel" class does not only realize the "CustomObserver" interface and observes the "Sketch" class. It also realizes the "CustomObservable" interface, implements all its methods and therefore, has a double function as subject and observer. Since the "CanvasViewModel" class now also is observable, the "CanvasView" class realizes the "CustomObserver" class and subscribes to the ViewModel. Whenever there are changes in the ViewModel the "CanvasView" class is notified.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\view\CanvasView*.java)


```
@Override
public void update() { invalidate(); }

public GraphicalElement getSelectedGraphicalElement() {
    return canvasViewModel.getSelectedGraphicalElement();
}
```

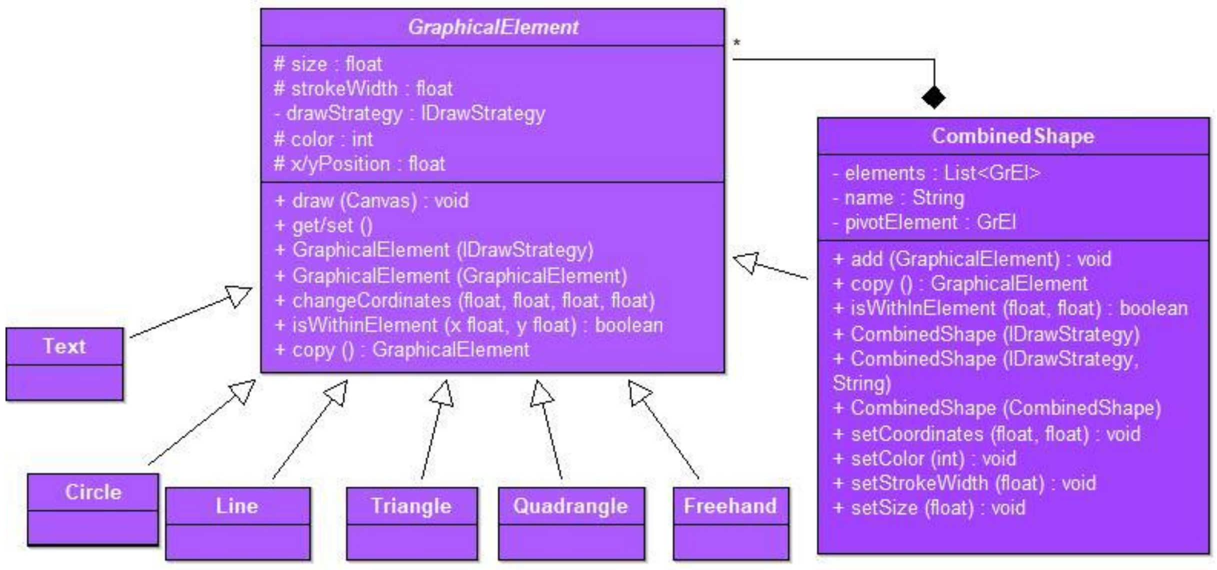
The observer pattern relates to FR 1 – FR 9, since the pattern is employed in any kind of action that involves an update of the display (e.g., modification, addition of objects).

As already mentioned in section 1.2 the observer pattern is essential for a proper implementation of the Model-View-ViewModel pattern. Since the flow of information should occur only from View to ViewModel to Model, we require the observer pattern to update "CanvasView" and "CanvasViewModel", when something happens in the subject structure below (e.g., a new graphical element is created in the "Sketch" class and needs to be displayed to the user). Also, by implementing the pattern and following the MVVM-approach, we were able to drastically reduce code duplication in the "CanvasViewModel" class.

Furthermore, our coupling between classes is reduced, since the subject and the observer are not directly connected, but only communicate via the implemented interfaces. Therefore, we would easily be able to add additional concrete subject or observer classes, if necessary.

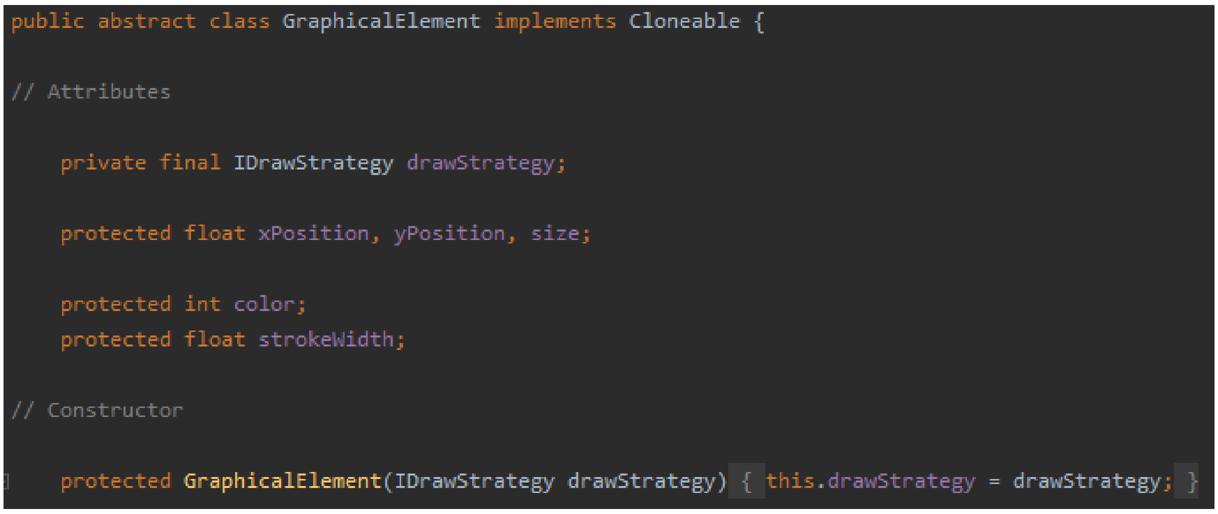
### 1.3.7 Composite Pattern

The composite pattern enables programmers to treat objects from a part-whole hierarchy, uniformly. This is achieved by connecting primitive "leaf classes" and more complex "container classes" to a shared interface. Containers can hold any number of primitive objects and each request is passed down to the respective subclass. The previously mentioned interface hides all information about the complex and primitive classes beneath. Therefore, the client does not know, with what kind of class it is working, leading to a well decoupled setup.


GraphicalElement
# size : float
# strokeWidth : float
- drawStrategy : IDrawStrategy
# color : int
# x/yPosition : float
+ draw (Canvas) : void
+ get/set ()
+ GraphicalElement (IDrawStrategy)
+ GraphicalElement (GraphicalElement)
+ changeCordinates (float, float, float, float)
+ isWithinElement (x float, y float) : boolean
+ copy () : GraphicalElement
*
CombinedShape
- elements : List<GrEl>
- name : String
- pivotElement : GrEl
+ add (GraphicalElement) : void
+ copy () : GraphicalElement
+ isWithInElement (float, float) : boolean
+ CombinedShape (IDrawStrategy)
+ CombinedShape (IDrawStrategy, String)
+ CombinedShape (CombinedShape)
+ setCoordinates (float, float) : void
+ setColor (int) : void
+ setStrokeWidth (float) : void
+ setSize (float) : void
Text
Circle
Line
Triangle
Quadrangle
Freehand

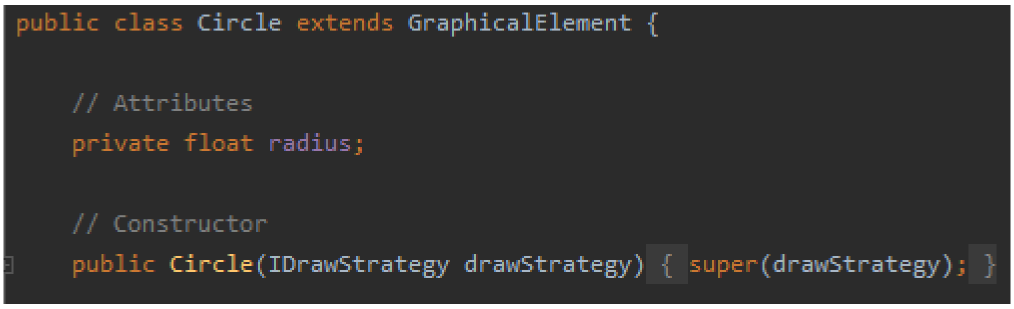
In this specific case, the composite pattern is applied to structure simple graphical elements (circles, lines, quadrangles, triangles, Text, Freehand) and combined shapes which consist of these simple shapes or even other selected combined shapes. In that way the composite pattern helps us to realize FR6 FR 7. The core element of this structure is the abstract class "GraphicalElement" which inherits most of the relevant attributes and methods to its children: Simple shape classes like "Circle" and the complex container class "CombinedShape".

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\GraphicalElement*.java)



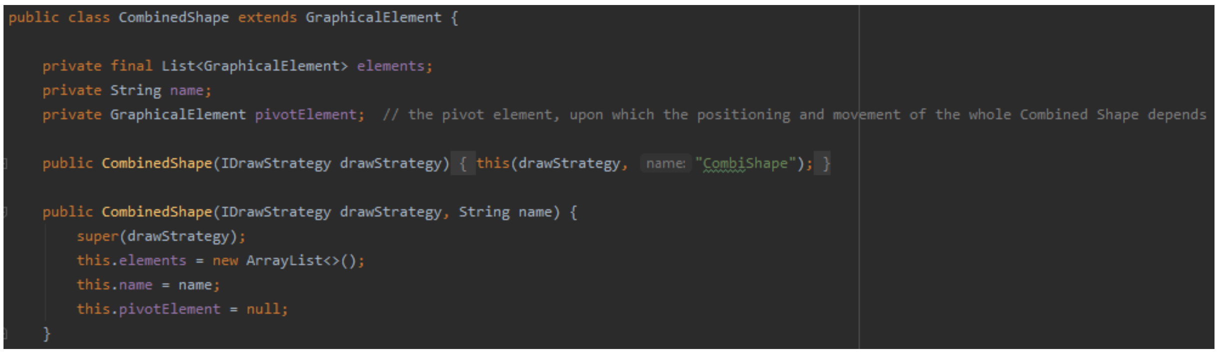
```java
public abstract class GraphicalElement implements Cloneable {

// Attributes

    private final IDrawStrategy drawStrategy;

    protected float xPosition, yPosition, size;

    protected int color;
    protected float strokeWidth;

// Constructor

    protected GraphicalElement(IDrawStrategy drawStrategy) { this.drawStrategy = drawStrategy; }
```

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\Circle*.java)



```java
public class Circle extends GraphicalElement {

    // Attributes
    private float radius;

    // Constructor
    public Circle(IDrawStrategy drawStrategy) { super(drawStrategy); }
```

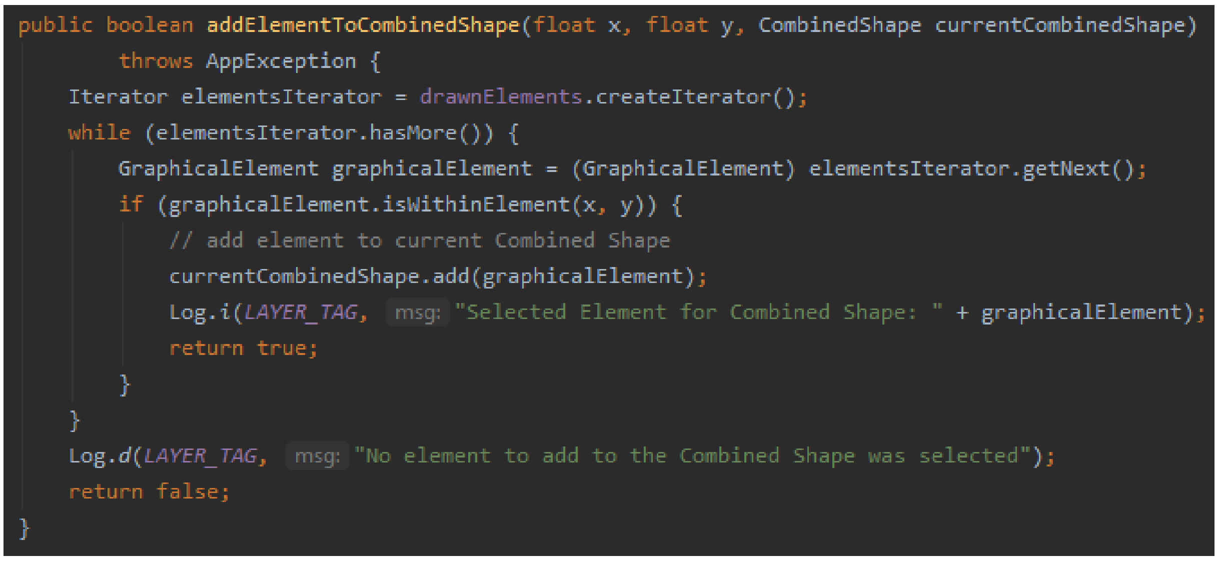
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\CombinedShape*.java)



```java
public class CombinedShape extends GraphicalElement {

    private final List<GraphicalElement> elements;
    private String name;
    private GraphicalElement pivotElement;  // the pivot element, upon which the positioning and movement of the whole Combined Shape depends

    public CombinedShape(IDrawStrategy drawStrategy) { this(drawStrategy,  name: "CombiShape"); }

    public CombinedShape(IDrawStrategy drawStrategy, String name) {
        super(drawStrategy);
        this.elements = new ArrayList<>();
        this.name = name;
        this.pivotElement = null;
    }
```

The "CombinedShape" class has an array list which can hold simple and complex graphical elements. This implementation towards the abstract class "graphical element" enables us to treat primitive and compound objects alike, when traversing over the array list on a layer object. Hence, we do not have to worry about the type of the graphical elements. This setup leaves us with a very flexible structure, so we could easily add new classes of graphical elements graphical elements in the future.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Layer*.java)



```
public boolean addElementToCombinedShape(float x, float y, CombinedShape currentCombinedShape)
        throws AppException {
    Iterator elementsIterator = drawnElements.createIterator();
    while (elementsIterator.hasMore()) {
        GraphicalElement graphicalElement = (GraphicalElement) elementsIterator.getNext();
        if (graphicalElement.isWithinElement(x, y)) {
            // add element to current Combined Shape
            currentCombinedShape.add(graphicalElement);
            Log.i(LAYER_TAG,  msg: "Selected Element for Combined Shape: " + graphicalElement);
            return true;
        }
    }
    Log.d(LAYER_TAG,  msg: "No element to add to the Combined Shape was selected");
    return false;
}
```

# 2 Implementation

## 2.1 Overview of Main Modules and Components

We followed the good principle of Software Architecture, that the quality attributes should be the determining factors for the physical and logical structure of Software – the non-functional requirements such as Maintainability, Readability and Reusability of code. As the general Model View-View Model Architecture already strives for these goals, our application also achieves these significantly – by letting the View classes (MainActivity and CanvasView) communicating only through the Model View classes with the actual model classes. Therefore, these subsystems form Components that themselves consist out of subcomponents. The Sketch classes implement the required interface provided by the Graphical Element as well as the Behavior in terms of the DrawStrategies (in both cases Socket -> Ball in this notation)

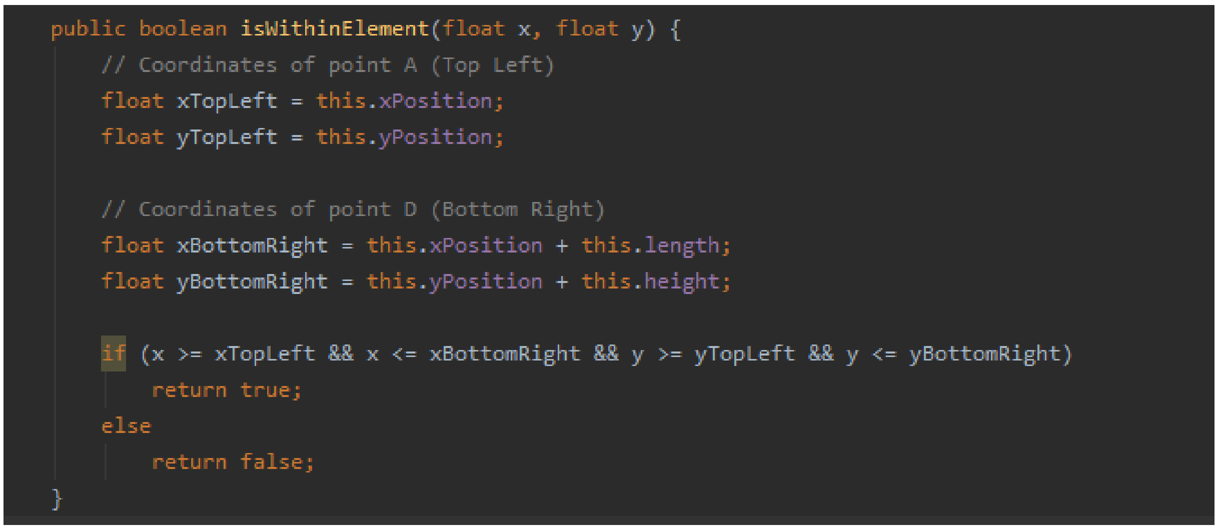
## 2.2 Coding Practices

In our implementation, we tried to consider four main coding practices: Naming, Commenting, Form of code and appropriate creation of methods.

Regarding the naming of variables, methods and classes, we decided upon the following principles:

Names should be intention revealing and avoid disinformation. We chose names that describe as close as possible what is inside a variable or what the task of a specific methods is. Thereby, we reduced the necessity of commenting and improved readability as well as maintainability.

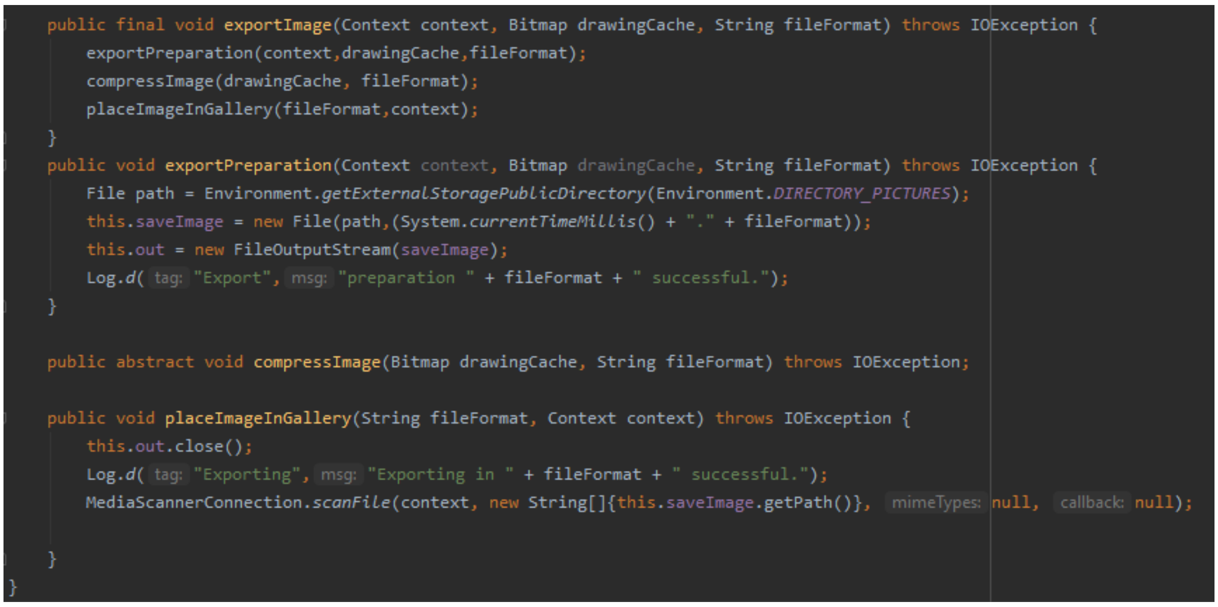
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\Circle*.java)



```java
public boolean isWithinElement(float x, float y) {
    // Coordinates of point A (Top Left)
    float xTopLeft = this.xPosition;
    float yTopLeft = this.yPosition;

    // Coordinates of point D (Bottom Right)
    float xBottomRight = this.xPosition + this.length;
    float yBottomRight = this.yPosition + this.height;

    if (x >= xTopLeft && x <= xBottomRight && y >= yTopLeft && y <= yBottomRight)
        return true;
    else
        return false;
}
```

Furthermore, our names are making meaningful distinctions between comparable or connected methods where it is feasible. Therefore, a reader does not get confused, due to similar naming schemes.

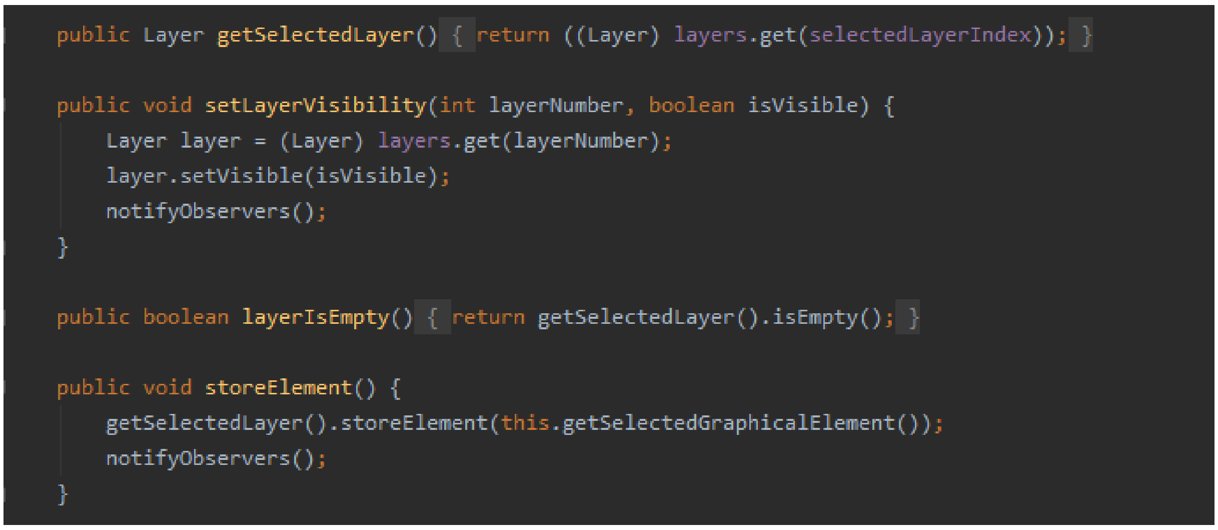
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\export\Export*.java)



```java
    public final void exportImage(Context context, Bitmap drawingCache, String fileFormat) throws IOException {
        exportPreparation(context,drawingCache,fileFormat);
        compressImage(drawingCache, fileFormat);
        placeImageInGallery(fileFormat,context);
    }
    public void exportPreparation(Context context, Bitmap drawingCache, String fileFormat) throws IOException {
        File path = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
        this.saveImage = new File(path,(System.currentTimeMillis() + "." + fileFormat));
        this.out = new FileOutputStream(saveImage);
        Log.d( tag: "Export", msg: "preparation " + fileFormat + " successful.");
    }

    public abstract void compressImage(Bitmap drawingCache, String fileFormat) throws IOException;

    public void placeImageInGallery(String fileFormat, Context context) throws IOException {
        this.out.close();
        Log.d( tag: "Exporting", msg: "Exporting in " + fileFormat + " successful.");
        MediaScannerConnection.scanFile(context, new String[]{this.saveImage.getPath()}, mimeTypes: null, callback: null);

    }
}
```

Lastly, our names are pronounceable and searchable which also enhances readability and maintainability of the code. While creating the app, this circumstance was very helpful, since we collaborated mostly virtually and being able to search for currently discussed methods or pronounce them without difficulties was beneficial, all the time.

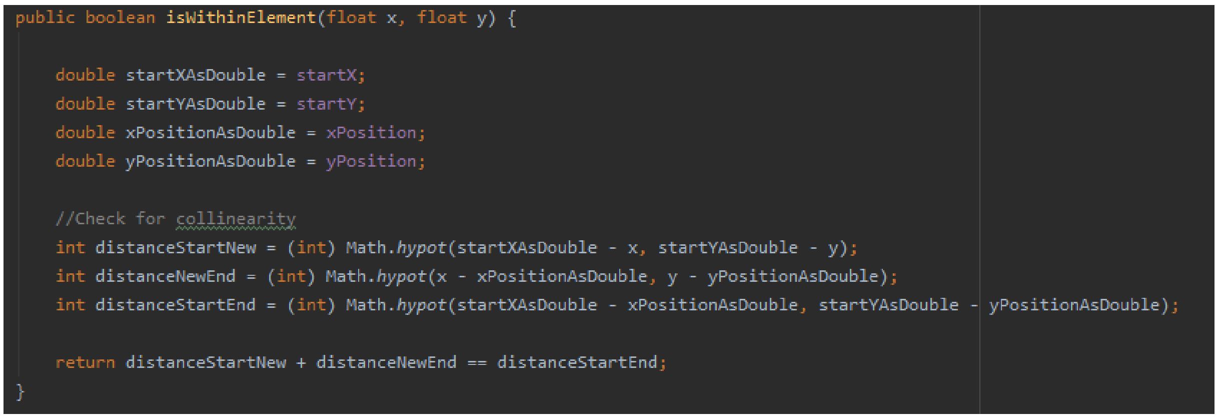
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\export\Export*.java)



```java
    public Layer getSelectedLayer() { return ((Layer) layers.get(selectedLayerIndex)); }

    public void setLayerVisibility(int layerNumber, boolean isVisible) {
        Layer layer = (Layer) layers.get(layerNumber);
        layer.setVisible(isVisible);
        notifyObservers();
    }

    public boolean layerIsEmpty() { return getSelectedLayer().isEmpty(); }

    public void storeElement() {
        getSelectedLayer().storeElement(this.getSelectedGraphicalElement());
        notifyObservers();
    }
```

Another major player in our code is commenting. One the one hand, there are ordinary comments which explain situations and details which are not obvious or exceed the explanatory capabilities of names. On the other hand, we documented with JavaDocs which we will cover in section 3.3.

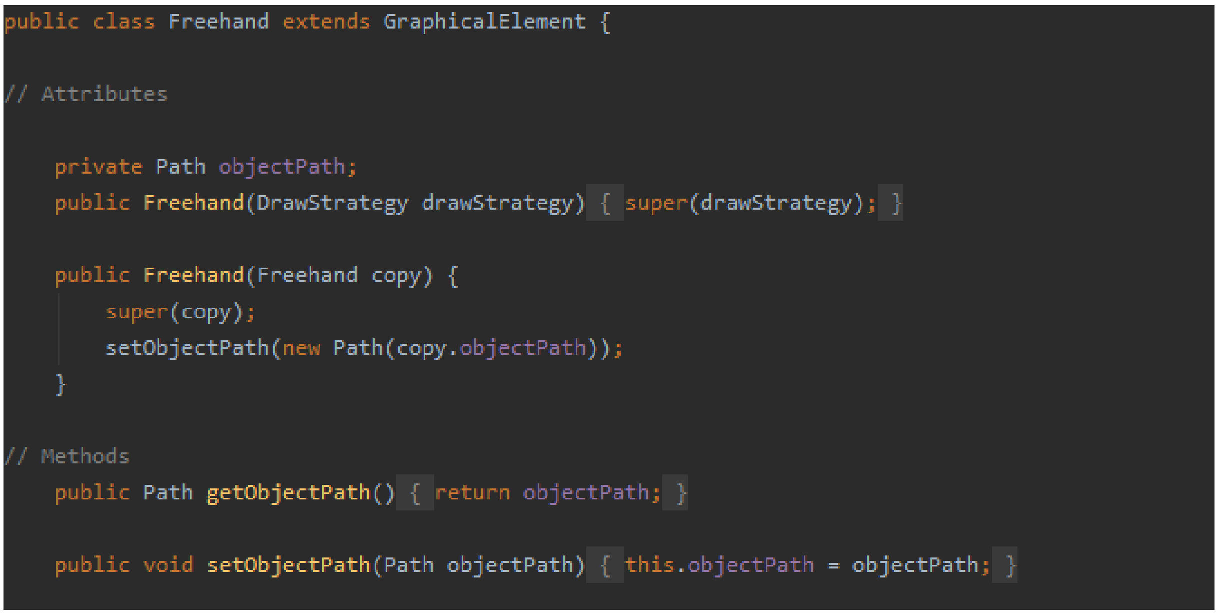
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\Line*.java)



```java
public boolean isWithinElement(float x, float y) {

    double startXAsDouble = startX;
    double startYAsDouble = startY;
    double xPositionAsDouble = xPosition;
    double yPositionAsDouble = yPosition;

    //Check for collinearity
    int distanceStartNew = (int) Math.hypot(startXAsDouble - x, startYAsDouble - y);
    int distanceNewEnd = (int) Math.hypot(x - xPositionAsDouble, y - yPositionAsDouble);
    int distanceStartEnd = (int) Math.hypot(startXAsDouble - xPositionAsDouble, startYAsDouble - yPositionAsDouble);

    return distanceStartNew + distanceNewEnd == distanceStartEnd;
}
```

In general, we tried to employ a low number of additional comments. However, we also added comments for the general structure of our code which might not be a proven approach, but in our opinion, it had a positive effect on the readability of our code.

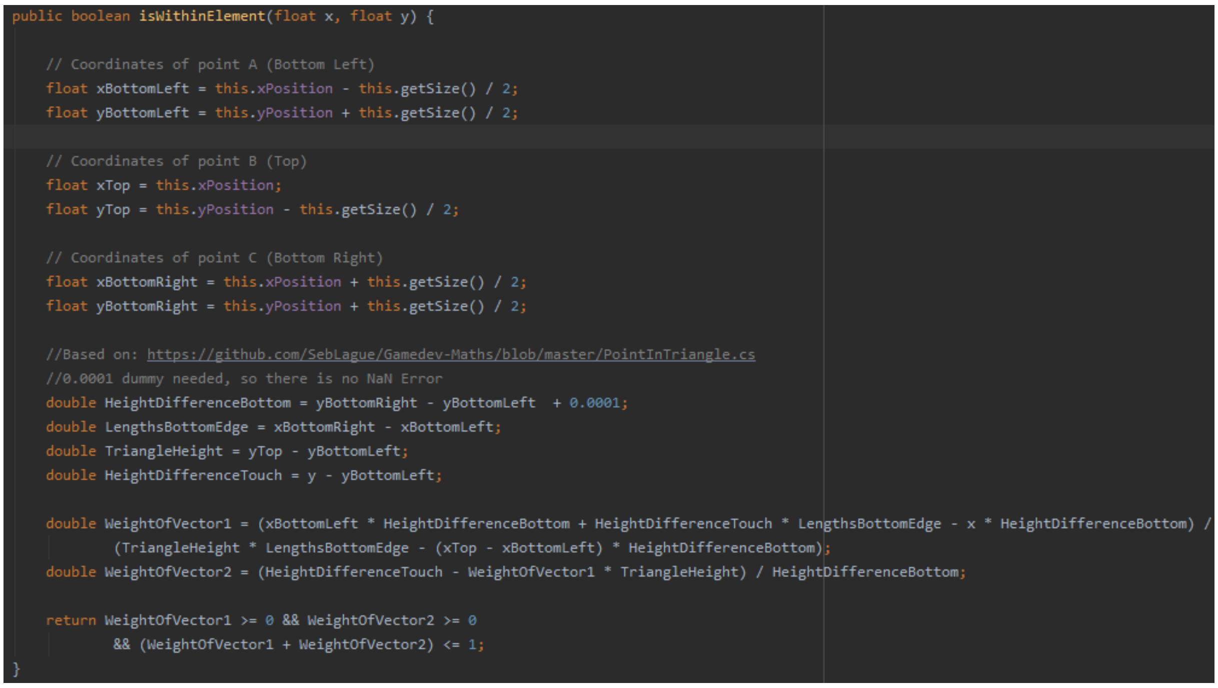
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\Freehand*.java)



```
public class Freehand extends GraphicalElement {

// Attributes

    private Path objectPath;
    public Freehand(DrawStrategy drawStrategy) { super(drawStrategy); }

    public Freehand(Freehand copy) {
        super(copy);
        setObjectPath(new Path(copy.objectPath));
    }

// Methods
    public Path getObjectPath() { return objectPath; }

    public void setObjectPath(Path objectPath) { this.objectPath = objectPath; }
```

Moreover, comments are used to mark code sections for which we our solution is heavily influenced by existing logic as described in other sources.

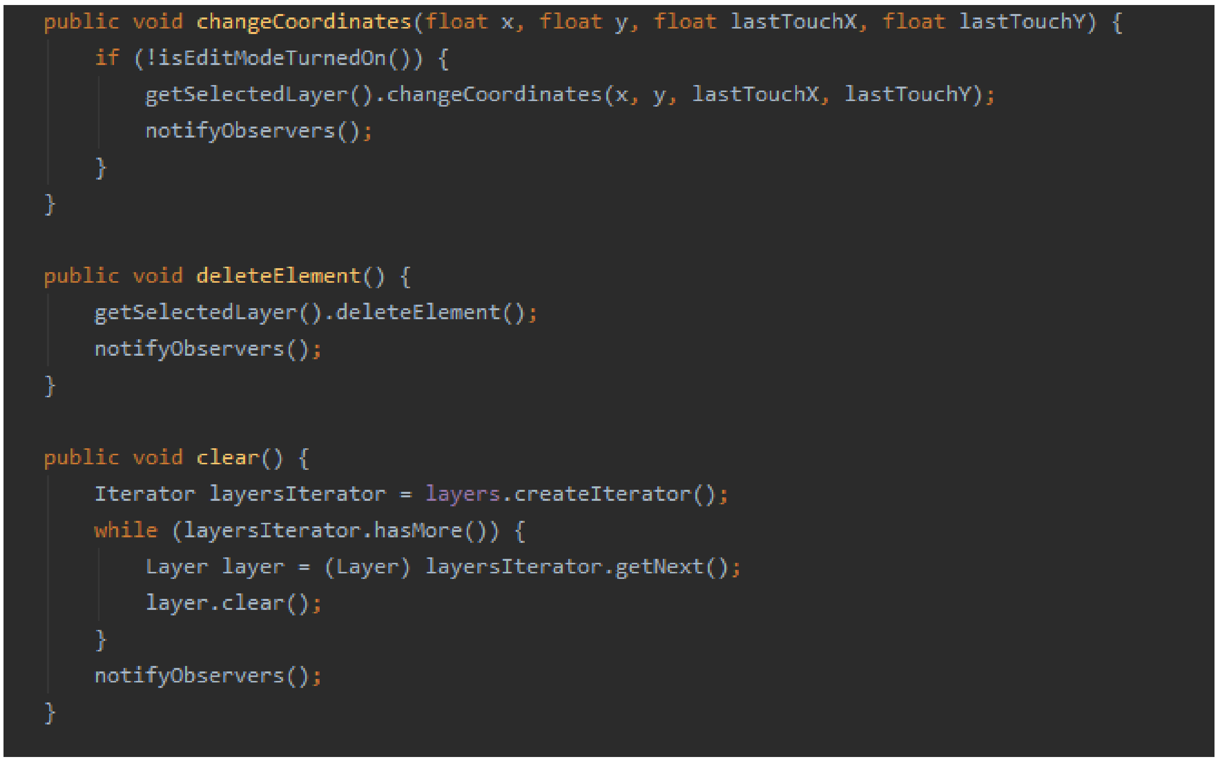
*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\Triangle*.java)



```
public boolean isWithinElement(float x, float y) {

    // Coordinates of point A (Bottom Left)
    float xBottomLeft = this.xPosition - this.getSize() / 2;
    float yBottomLeft = this.yPosition + this.getSize() / 2;

    // Coordinates of point B (Top)
    float xTop = this.xPosition;
    float yTop = this.yPosition - this.getSize() / 2;

    // Coordinates of point C (Bottom Right)
    float xBottomRight = this.xPosition + this.getSize() / 2;
    float yBottomRight = this.yPosition + this.getSize() / 2;

    //Based on: https://github.com/SebLague/Gamedev-Maths/blob/master/PointInTriangle.cs
    //0.0001 dummy needed, so there is no NaN Error
    double HeightDifferenceBottom = yBottomRight - yBottomLeft  + 0.0001;
    double LengthsBottomEdge = xBottomRight - xBottomLeft;
    double TriangleHeight = yTop - yBottomLeft;
    double HeightDifferenceTouch = y - yBottomLeft;

    double WeightOfVector1 = (xBottomLeft * HeightDifferenceBottom + HeightDifferenceTouch * LengthsBottomEdge - x * HeightDifferenceBottom) /
            (TriangleHeight * LengthsBottomEdge - (xTop - xBottomLeft) * HeightDifferenceBottom);
    double WeightOfVector2 = (HeightDifferenceTouch - WeightOfVector1 * TriangleHeight) / HeightDifferenceBottom;

    return WeightOfVector1 >= 0 && WeightOfVector2 >= 0
            && (WeightOfVector1 + WeightOfVector2) <= 1;
}
```

The appearance of our code in terms of form and style is oriented towards the Google Java Style Guide. This is also covered in greater detail in section 3.3. We consistently use an indent size of 4 and utilize indents to logically structure our code. Furthermore, we paid attention to the alignment of braces.

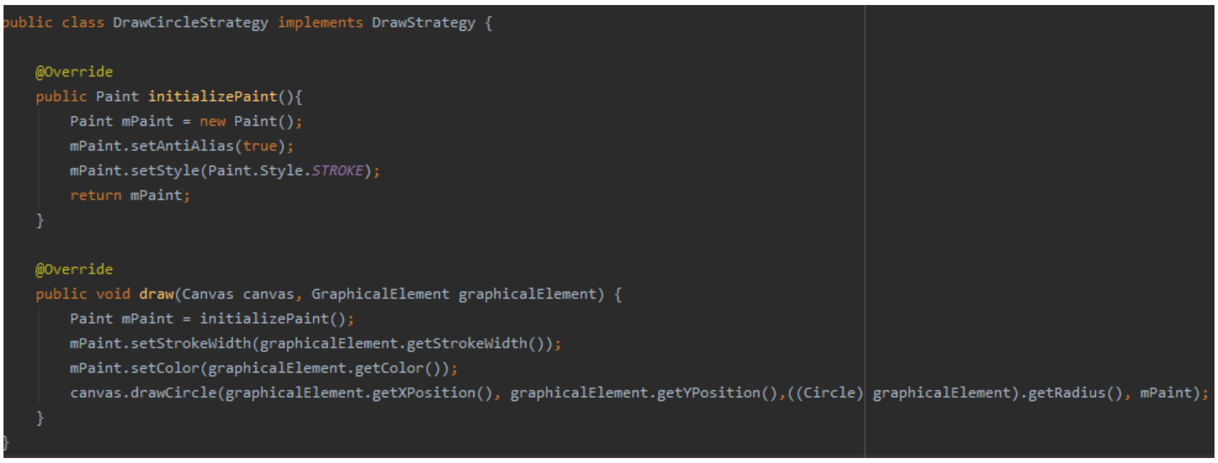
(*implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch*.java)



```java
public void changeCoordinates(float x, float y, float lastTouchX, float lastTouchY) {
    if (!isEditModeTurnedOn()) {
        getSelectedLayer().changeCoordinates(x, y, lastTouchX, lastTouchY);
        notifyObservers();
    }
}

public void deleteElement() {
    getSelectedLayer().deleteElement();
    notifyObservers();
}

public void clear() {
    Iterator layersIterator = layers.createIterator();
    while (layersIterator.hasMore()) {
        Layer layer = (Layer) layersIterator.getNext();
        layer.clear();
    }
    notifyObservers();
}
```

With respect to the creation of methods, our highest priority is to create functions that solve one specific problem in its entirety, so we no longer have to worry about it. Once this situation is achieved, we can simply call the function when a certain action is required [e.g., create a circle object] and don't have to trouble ourselves with the inner logic. Thereby we are able to hide information and reduce the complexity of our application.

(*implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\draw\DrawCircleStrategy*.java)



```java
ublic class DrawCircleStrategy implements DrawStrategy {

    @Override
    public Paint initializePaint(){
        Paint mPaint = new Paint();
        mPaint.setAntiAlias(true);
        mPaint.setStyle(Paint.Style.STROKE);
        return mPaint;
    }

    @Override
    public void draw(Canvas canvas, GraphicalElement graphicalElement) {
        Paint mPaint = initializePaint();
        mPaint.setStrokeWidth(graphicalElement.getStrokeWidth());
        mPaint.setColor(graphicalElement.getColor());
        canvas.drawCircle(graphicalElement.getXPosition(), graphicalElement.getYPosition(),((Circle) graphicalElement).getRadius(), mPaint);
    }
}
```

## 2.3 Defensive Programming

*Error-handling techniques*

- Return a neutral value / Return the same answer as the previous time: if the "CombinedShape" has no elements, return initial values set during construction or previous set values

*(implementation/Sketch_App/app/src/main/java/at/ac/univie/se2ws2020team0310/sketch_app/model/graphicalElements/CombinedShape.java)*

```java
/**
 * The Combined Shape will use the y coordinate of the pivot or first element in
the list, if
 * applicable
 *
 * @return the y coordinate
 */
@Override
public float getYPosition() {
    if (this.elements.isEmpty()) {
        return this.yPosition;
    }
    return this.pivotElement != null ? this.pivotElement.getYPosition()
            : this.elements.get(0).getYPosition();
}
```

*Exceptions:* errors that can be handled internally by the application

*(implementation/Sketch_App/app/src/main/java/at/ac/univie/se2ws2020team0310/sketch_app/model/graphicalElements/CombinedShape.java)*

```java
/**
 * Add a new GraphicalElement to this CombinedShape
 *
 * @param element the element to add, if not selected already
 * @throws AppException in case the element is already present in the list
 */
public void add(GraphicalElement element) throws AppException {
    if (this.elements.contains(element)) {
        throw new AppException(element + " is already selected");
    }
    this.elements.add(element);
}
```

- Display an error message wherever the error is encountered

*(implementation/Sketch_App/app/src/main/java/at/ac/univie/se2ws2020team0310/sketch_app/view/CanvasView.java)*

```
} catch (AppException ex) {
    // Error Handling: any AppException caused by user Touch actions will be
caught and handled here
    ViewUtils.showToast(getContext(), ex.getLocalizedMessage());
    Log.w("CanvasView", ex.getLocalizedMessage());
}
return false;
```

# 3 Software Quality

## 3.1 Code Metrics

Number of packages: 18

Number of dependencies in gradle-File: 10

Java - Lines of code: 4940

Java - Comment lines of code: 542

Java - Number of classes: 64

XML - Lines of code: 763

XML - Number of files: 18

There are three known functionality bugs in the code:

1) The saving functionality of free-hand drawings does not work as expected. The export is working, because the methods are not coupled with each other except for the Bitmap being used.
2) In the use case of users combining multiple graphical objects together, saving them as a combined shape and then trying to load this shape again, added line elements do not behave as expected. Our end-to-end-test revealed that they disappear shortly after being inserted onto the screen.
3) The export of files only works on the Emulator device. There it appears at the proper location. And we have tested multiple locations on the internal as well as external memory, but could not change the app's behaviour.

Additionally, we have made use of the SonarLint static code analysis tool – which gave useful feedback on applying defensive programming techniques such as implementing default cases for the menu click listeners.

Taking these code metrics and comparing them to SUPD (Number of packages: 6; Lines of code: 1748; Comment lines of code: 70; Number of classes: 21), back then, we only had around 35% of lines of code, classes and packages. From our point of view it is quite interesting that the number of packages, lines of code and number of classes developed at the exact same scale. This either speaks of a good distribution at the beginning which we consistently developed or gives evidence that there has not been any major improvement in our structure. On the other hand, the comment lines have increased drastically. At the time of SUPD 4% of all lines were comments. Now, this number has grown to nearly 11%. Even though, there is no general rule of thumb for the use of comments, such a growth might indicate an overuse of comments. However, it might be argued that the amount of commented lines increased with the developing complexity of the application.

## 3.2 Testcases for Functional Requirements

## 1.1 Testcases for Functional Requirements

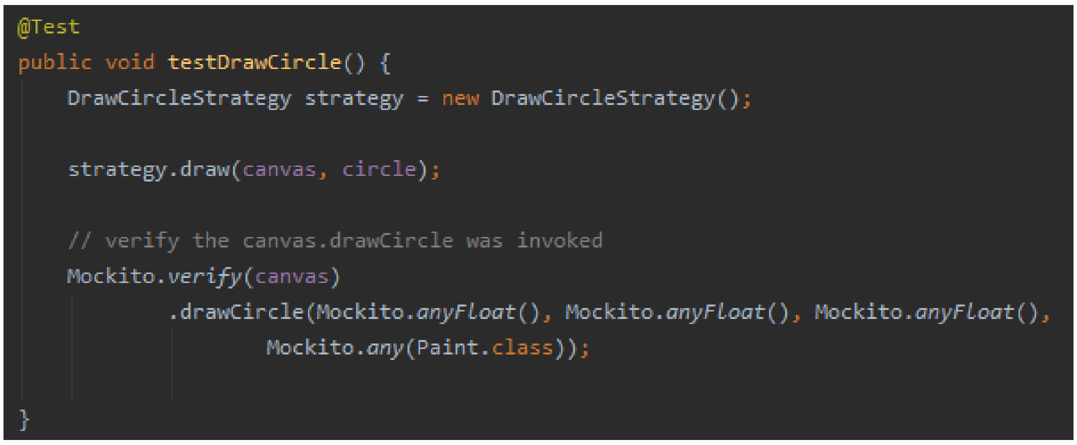
We established a testing strategy of multiple testing techniques such as peer code reviews, End-to-End-Testing, UI tests such as Espresso tests and Unit tests. The latter were constructed using the JUnit and Mockito frameworks and established for all the main classes in the model (DrawStrategies, export functionality, Graphical Elements), the ViewModel (CanvasViewModel) and the View (CanvasView and MainActivity). For the concrete test cases, see the section below.

### 3.1.1 Tests for Classes in Model

For the classes in the model, we first created test classes for the DrawStrategies. In the respective DrawStrategy test classes, mocks are first created with the @Mock annotation, i.e. dummy instances. With these mocks, the functionality of the method can now be tested. Mockito.verify checks whether the respective draw method is called. With the Mockito any methods, random values are assigned to the parameters of the draw method.

*Code snippet:*

*(implementation\Sketch_App\app\src\test\java\at\ac\univie\se2ws2020team0310\sketch_app\model\draw\DrawCircleStrategyTest.java):*

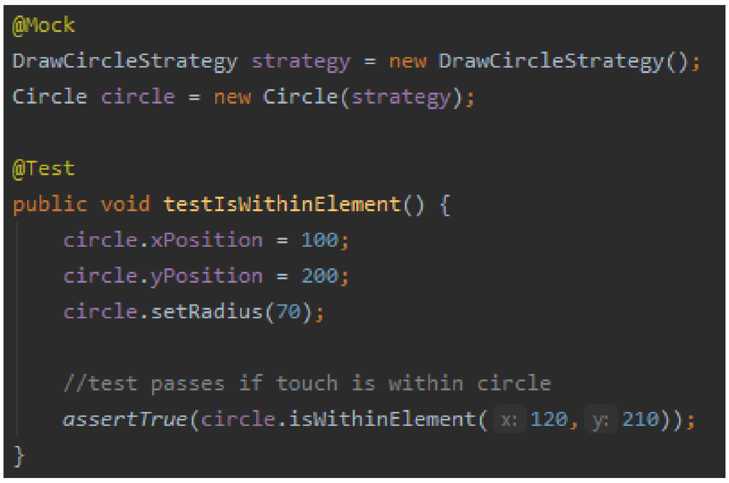


```
@Test
public void testDrawCircle() {
    DrawCircleStrategy strategy = new DrawCircleStrategy();

    strategy.draw(canvas, circle);

    // verify the canvas.drawCircle was invoked
    Mockito.verify(canvas)
            .drawCircle(Mockito.anyFloat(), Mockito.anyFloat(), Mockito.anyFloat(),
                    Mockito.any(Paint.class));

}
```

The tests for the export classes throw an IOException if an unsupported FileFormat is used, e.g.: pdf.

For the graphical element classes, we tested the functionality of the isWithinElement method, i.e. the method that checks whether a selected point on the screen is located within a graphical element. To do this, we first created dummy instances using @Mock annotation. The mocks get coordination and all other parameters that are needed for the calculation in the isWithinElement method. Then the test method uses assertTrue() to check whether the isWithinElement method is true for a specific point(x,y).

*(implementation\Sketch_App\app\src\test\java\at\ac\univie\se2ws2020team0310\sketch_app\model\graphicalElements\CircleTest.java):*



```
@Mock
DrawCircleStrategy strategy = new DrawCircleStrategy();
Circle circle = new Circle(strategy);

@Test
public void testIsWithinElement() {
    circle.xPosition = 100;
    circle.yPosition = 200;
    circle.setRadius(70);

    //test passes if touch is within circle
    assertTrue(circle.isWithinElement( x: 120, y: 210));
}
```

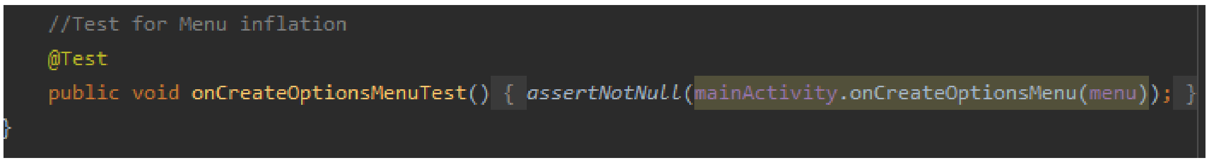
In the test class GraphicalElementFactoryTest, the create methods of the respective element types are checked. For the example Triangle. First attributes are defined, such as colour, size, strokeWidth, which must be passed to create a GraphicalElement. In the test method for Triangle, a Triangle is first created with the attributes defined above. Then assertTrue is used to check whether the created element has the correct type, i.e. Triangle, and whether the drawStrategy used is the DrawTriangleStrategy.

The assertEquals method is used to check whether the attributes defined above match the attributes that can be retrieved using getter methods.

In the ElementCollectionTest with the iterators and collections, various mock objects are first created, which can be added to the array list or otherwise retrieved for the respective methods. The testGet() method uses Mockito verify() to check whether the get method is called. In testAdd, the same method is used to test whether the add method is called when an element is added. In testIndexOf, assertEquals is used to check whether the expected index and the actual index output by the indexOf method match. In testContains it is checked whether contains() returns false for an element that is not in the array list.

### 3.1.2 Tests for Classes in ViewModel

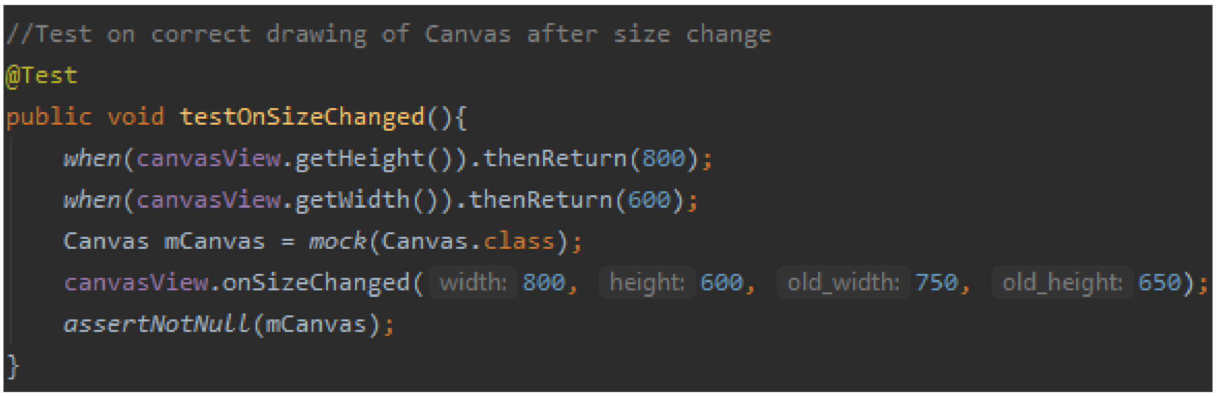
The test being conducted on the MainViewModel relates to the correct display of the Options Menu. It checks on whether it gets properly inflated On click of the Overflow menu.



```
    //Test for Menu inflation
    @Test
    public void onCreateOptionsMenuTest() { assertNotNull(mainActivity.onCreateOptionsMenu(menu)); }
}
```

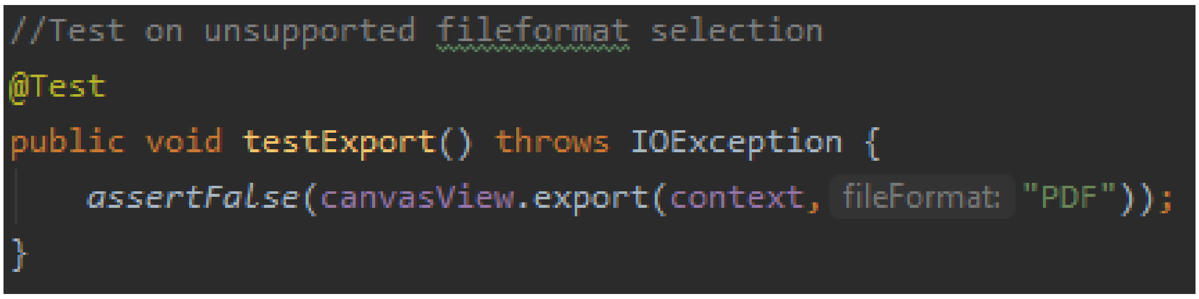
### 3.1.3 Tests for Classes in View

The view class is mainly tested on three different functionalities. The first one checks whether the CanvasView method sets the size of the Canvas correctly on size change of the latter.



```
//Test on correct drawing of Canvas after size change
@Test
public void testOnSizeChanged(){
    when(canvasView.getHeight()).thenReturn(800);
    when(canvasView.getWidth()).thenReturn(600);
    Canvas mCanvas = mock(Canvas.class);
    canvasView.onSizeChanged( width: 800, height: 600, old_width: 750, old_height: 650);
    assertNotNull(mCanvas);
}
```

Up next is the test on functionality in a case where an unsupported file format is chosen, in this case PDF. The test simply calls the export method with the PDF-parameter, and asserts whether this expression evaluates to false.



```
//Test on unsupported fileformat selection
@Test
public void testExport() throws IOException {
    assertFalse(canvasView.export(context, fileFormat: "PDF"));
}
```

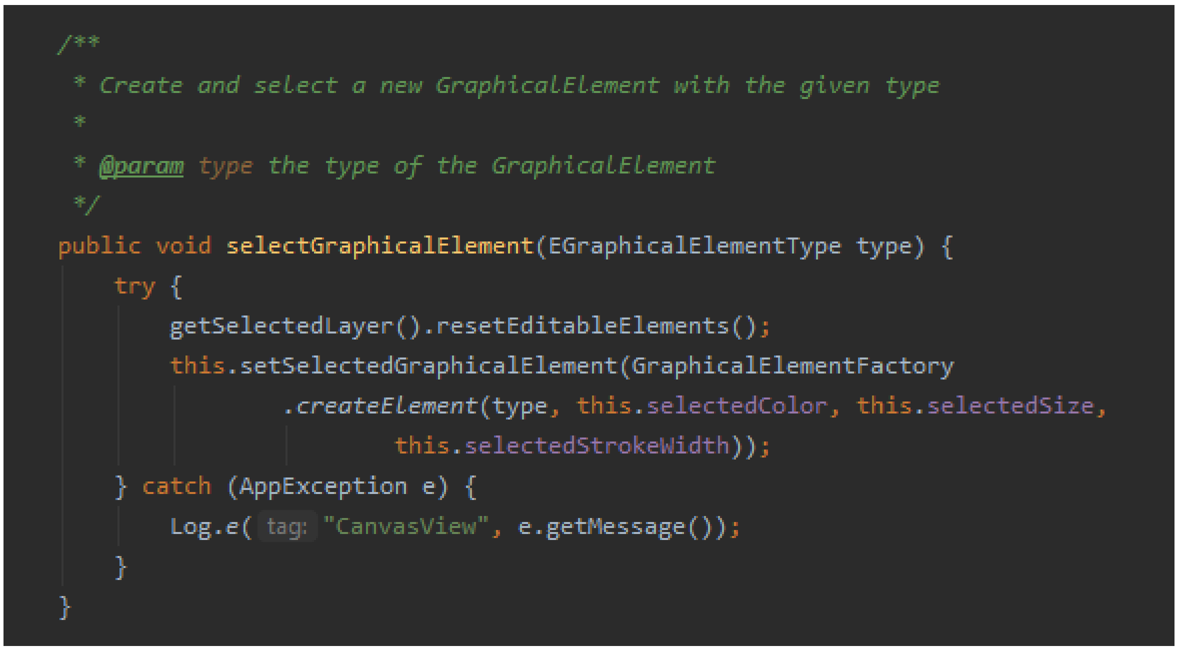
The last test in the View-Class refers to the detection of On-Touch-Events. The expression of event detection after mocking a specific user action (ACTION_UP) is checked on being not null.

## 3.3 Quality Requirements Coverage

In the following, we are discussing our implementation regarding our quality requirement adherence.

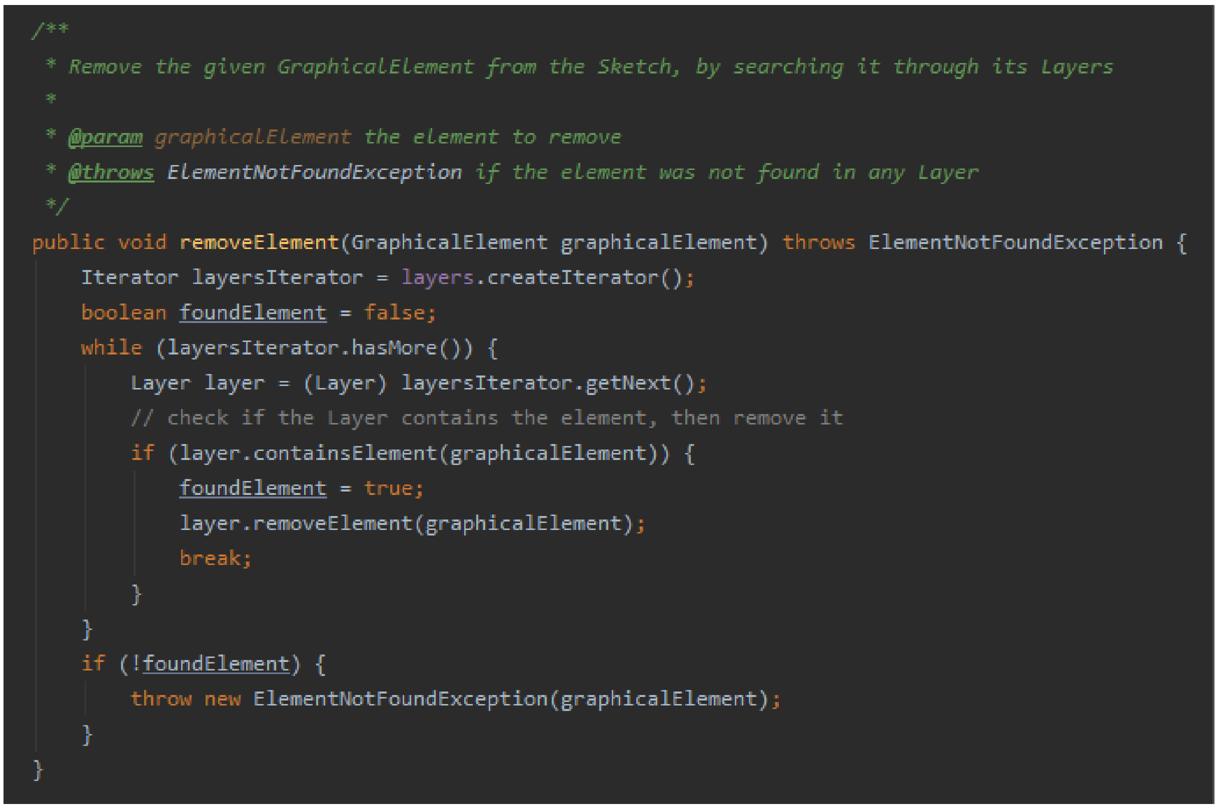
As described in section 2.2, we comment or code in two different ways. On the one hand we are using ordinary comments in order to increase its readability and maintainability. On the other hand, we created several Javadocs for methods which are either of great importance or difficult to understand, due to their complexity.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch.java)*


```
/**
 * Create and select a new GraphicalElement with the given type
 *
 * @param type the type of the GraphicalElement
 */
public void selectGraphicalElement(EGraphicalElementType type) {
    try {
        getSelectedLayer().resetEditableElements();
        this.setSelectedGraphicalElement(GraphicalElementFactory
                .createElement(type, this.selectedColor, this.selectedSize,
                        this.selectedStrokeWidth));
    } catch (AppException e) {
        Log.e( tag: "CanvasView", e.getMessage());
    }
}
```

Furthermore, our code is compliant to the Google Java Style Guide. We use the correct naming schemes, indent accordingly, follow the programming practices and format Javadocs and comments as described in the documentation.

*(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\Sketch.java)*

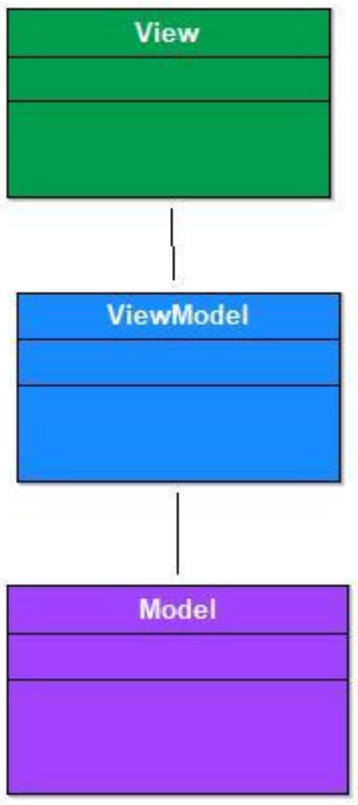

```
/**
 * Remove the given GraphicalElement from the Sketch, by searching it through its Layers
 *
 * @param graphicalElement the element to remove
 * @throws ElementNotFoundException if the element was not found in any Layer
 */
public void removeElement(GraphicalElement graphicalElement) throws ElementNotFoundException {
    Iterator layersIterator = layers.createIterator();
    boolean foundElement = false;
    while (layersIterator.hasMore()) {
        Layer layer = (Layer) layersIterator.getNext();
        // check if the Layer contains the element, then remove it
        if (layer.containsElement(graphicalElement)) {
            foundElement = true;
            layer.removeElement(graphicalElement);
            break;
        }
    }
    if (!foundElement) {
        throw new ElementNotFoundException(graphicalElement);
    }
}
```

We decided to comply with this style guides, since our main objective was to be consistent within our implementation and integrating the Google Java Style Guide in a Java project is quite easy.

As already described in detail in section 2.2, we tried to consider four main coding practices: Naming, Commenting, Form of code and appropriate creation of methods.

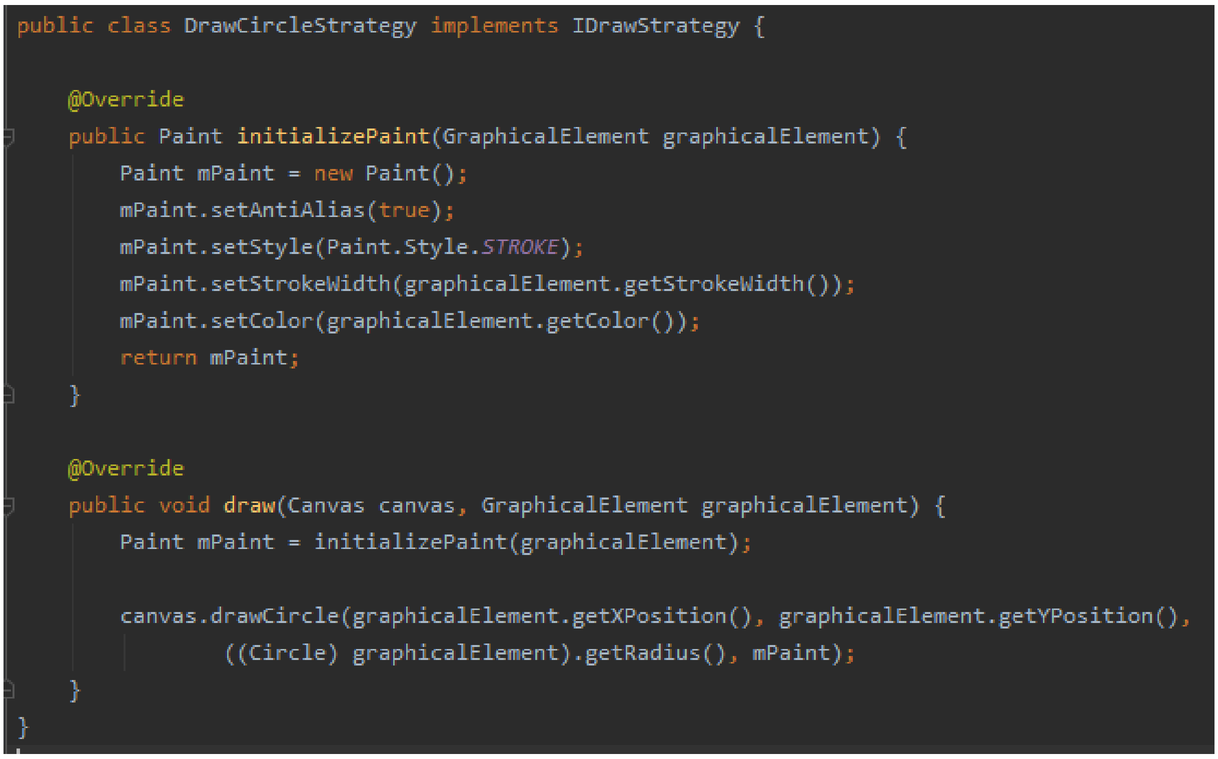
Further, as mentioned in section 2.3 our defensive programming relies on error-handling techniques, exceptions, and error message.

We apply various design principles in our implementation. The most obvious principle is modularity. Due to the implementation of the MVVM approach, our application is split into the separated modules which are loosely coupled.



Additionally, we comply to the principle of information hiding which we often use by programming to an interface. For instance, each "DrawGraphicalElementStrategy" has a specific way it must be drawn on the canvas. However only the object itself has this information. Hence, when the "draw" method is invoked on the object, the method does not specify how the object is drawn on the canvas, but the object itself provides this information.

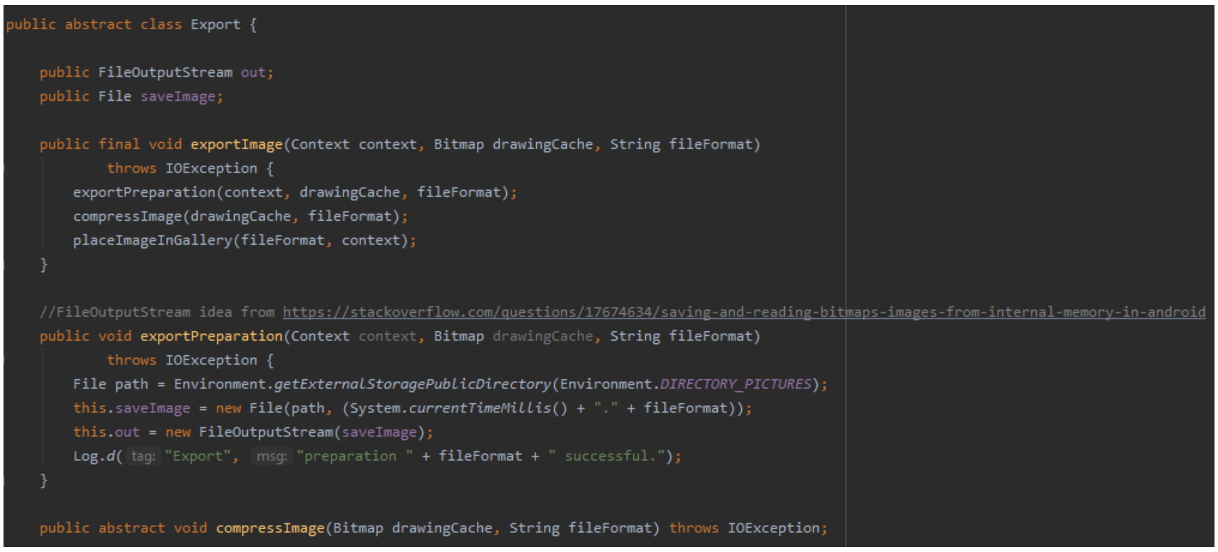
(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\draw\DrawCircleStrategy.java)



```java
public class DrawCircleStrategy implements IDrawStrategy {

    @Override
    public Paint initializePaint(GraphicalElement graphicalElement) {
        Paint mPaint = new Paint();
        mPaint.setAntiAlias(true);
        mPaint.setStyle(Paint.Style.STROKE);
        mPaint.setStrokeWidth(graphicalElement.getStrokeWidth());
        mPaint.setColor(graphicalElement.getColor());
        return mPaint;
    }

    @Override
    public void draw(Canvas canvas, GraphicalElement graphicalElement) {
        Paint mPaint = initializePaint(graphicalElement);

        canvas.drawCircle(graphicalElement.getXPosition(), graphicalElement.getYPosition(),
                ((Circle) graphicalElement).getRadius(), mPaint);
    }
}
```

The last design principle which we follow is the principle of "separation of concerns". A well-suited example for such a case is the "Export" class in which we implemented the template method. A rather complex function is split up into smaller methods, each handling a specific part of the main task. Even though, it is not the main scope of the template method, the pattern still does a great job at adhering to the principle of "separation of concerns"

(implementation\Sketch_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch_app\model\export\Export.java)



```java
public abstract class Export {

    public FileOutputStream out;
    public File saveImage;

    public final void exportImage(Context context, Bitmap drawingCache, String fileFormat)
            throws IOException {
        exportPreparation(context, drawingCache, fileFormat);
        compressImage(drawingCache, fileFormat);
        placeImageInGallery(fileFormat, context);
    }

    //FileOutputStream idea from https://stackoverflow.com/questions/17674634/saving-and-reading-bitmaps-images-from-internal-memory-in-android
    public void exportPreparation(Context context, Bitmap drawingCache, String fileFormat)
            throws IOException {
        File path = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
        this.saveImage = new File(path, (System.currentTimeMillis() + "." + fileFormat));
        this.out = new FileOutputStream(saveImage);
        Log.d( tag: "Export",  msg: "preparation " + fileFormat + " successful.");
    }

    public abstract void compressImage(Bitmap drawingCache, String fileFormat) throws IOException;
```

Lastly, our testing approach is based on multiple techniques: Peer code reviews, end-to-end-testing, UI tests such as Espresso tests and Unit tests (JUnit and Mockito). More details can be found in section 3.2.

# 1 Team Contribution

## 1.1 Project Tasks and Schedule

| | | October | | | | November | | | | December | | | | January | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 |
| JanPreparation/Planning | Requirements checking | 🟨 | 🟨 | | | | | | | | | | | | |
| | Task distribution | | | 🟨 | | | | | | | | | | | |
| | Architectural decisions | | | | 🟨 | | | | | | | | | | |
| Development/ Test | GitLab Setup | | | | 🟨 | | | | | | | | | | |
| | Implementation & Testing | | | | 🟨 | 🟨 | 🟨 | | | | | | | | |
| | SUPD submission | | | | | | | 🟥 | | | | | | | |
| | Further Implementation & Testing | | | | | | | | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 |
| Implementation | Check on Design Pattern Use | | | | | | | | | | 🟨 | 🟨 | 🟨 | | |
| | Final Testing | | | | | | | | | | | | | 🟨 | 🟨 |
| | Final Code Clean-Up | | | | | | | | | | | | | 🟨 | 🟨 |
| | Preparation of presentation materials | | | | | | | | | | | | | 🟨 | 🟨 |
| | DEAD submission | | | | | | | | | | | | | | 🟥 |

## 1.2 Distribution of Work and Efforts

| **Task** | **Distribution** | **Time** |
|---|---|---|
| Android research and tutorials | All | 30h |
| Design draft | All | 3h |
| Design Pattern research | All | 5h |
| UML class diagrams MVVM, Design Patterns | Leyla | 6h |
| UML class diagram Model | Felix | 1h |
| Graphical user interface | Felix | 11h |
| Factory pattern implementation | Leyla | 3h |
| Strategy pattern implementation | Leyla | 4h |
| Circle implementation | Jonas | 4h |
| Quadrangle implementation | Jonas | 4h |
| Line implementation | Jonas | 5h |
| Triangle implementation | Jonas | 3h |
| Text field implementation | Sandra | 8h |
| Free hand drawing implementation | Christian, Leyla | 9h |
| Draw width & and colour picker implementation | Felix | 5h |
| “Graphical element” functionality implementation | Jonas | 4h |
| Text size implementation | Sandra | 2h |
| “Clear” functionality | Sandra, Jonas | 1h |
| Software architecture review | Leyla & Jonas | 8h |
| Static code analysis tool | Felix | 4h |
| Testing SUPD | Felix & Christian | 3h |
| Code review SUPD | All | 10h |
| SUPD Documentation | Leyla, Christian, Felix | 9h |
| Combi Shapes | Leyla | 15h |
| Composite Pattern implementation | Leyla | 5h |
| Selection of multiple elements | Jonas | 4 |
| Decorator Pattern implementation | Sandra | 15h |
| Iterator Pattern Implementation | Jonas | 4h |
| Observer Pattern Implementation | Jonas | 5h |
| Template Pattern implementation | Felix, Christian | 5h |
| Export | Felix, Christian | 10h |
| Alternative Application approach | Felix | 10h |
| MVVM rework | All | 60h |
| Save and Load | Jonas | 10h |
| Layer / Sketch design | Jonas | 5h |
| GUI rework | Felix | 7h |
| Move objects | Sandra, Leyla, Jonas, Christian | 20h |
| Code review DEAD | All | 10h |

| | | |
|---|---|---|
| Unit tests DEAD | Sandra, Leyla, Jonas, Felix | 12h |
| End-to-end tests DEAD | Felix | 3h |
| DEAD documentation - Main part | Christian | 35h |
| DEAD documentation - input | All | 15h |
| UML – diagrams | Leyla | 10h |

# 2 Appendix

## 2.1 Emulator configurations

*Name: Pixel_XL_API_30*

*CPU/ABI: Google APIs Intel Atom (x86)*

*Target: google_apis [Google APIs] (API level 30)*

*Skin: pixel_xl_silver*

*SD Card: 512M*

*fastboot.chosenSnapshotFile:*

*runtime.network.speed: full*

*hw.accelerometer: yes*

*hw.device.name: pixel_xl*

*hw.lcd.width: 1440*

*hw.initialOrientation: Portrait*

*image.androidVersion.api: 30*

*tag.id: google_apis*

*hw.mainKeys: no*

*hw.camera.front: emulated*

*avd.ini.displayname: Pixel XL API 30*

*hw.gpu.mode: auto*

*hw.ramSize: 1536*

*PlayStore.enabled: false*

```
fastboot.forceColdBoot: no
hw.cpu.ncore: 4
hw.keyboard: yes
hw.sensors.proximity: yes
hw.dPad: no
hw.lcd.height: 2560
vm.heapSize: 384
skin.dynamic: yes
hw.device.manufacturer: Google
hw.gps: yes
hw.audioInput: yes
image.sysdir.1: system-images\android-30\google_apis\x86\
showDeviceFrame: yes
hw.camera.back: virtualscene
AvdId: Pixel_XL_API_30
hw.lcd.density: 560
hw.arc: false
hw.device.hash2: MD5:f78477654d5471c3d7f705251d14e72f
fastboot.forceChosenSnapshotBoot: no
fastboot.forceFastBoot: yes
hw.trackBall: no
hw.battery: yes
hw.sdCard: yes
tag.display: Google APIs
runtime.network.latency: none
disk.dataPartition.size: 800M
hw.sensors.orientation: yes
```

```
avd.ini.encoding: UTF-8

hw.gpu.enabled: yes
```