Software Engineering 2  
SUPD REPORT

Status Update 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 |

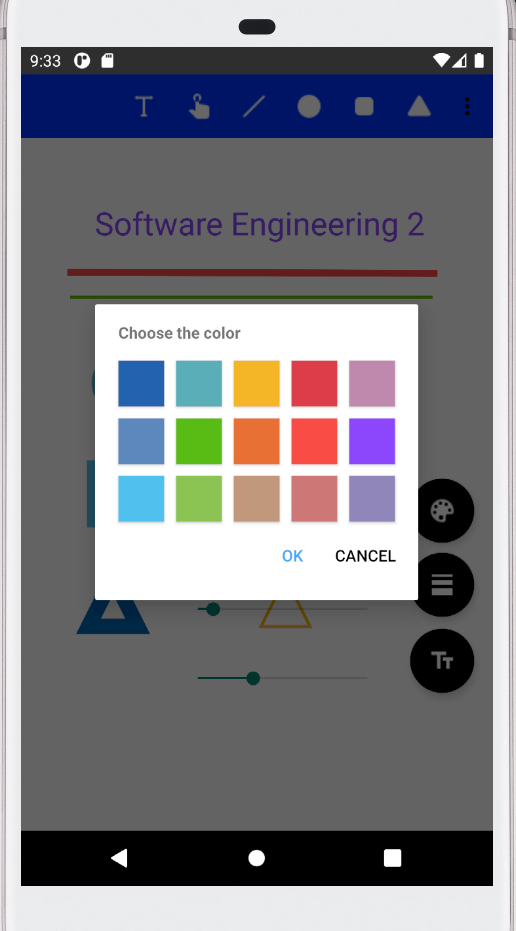
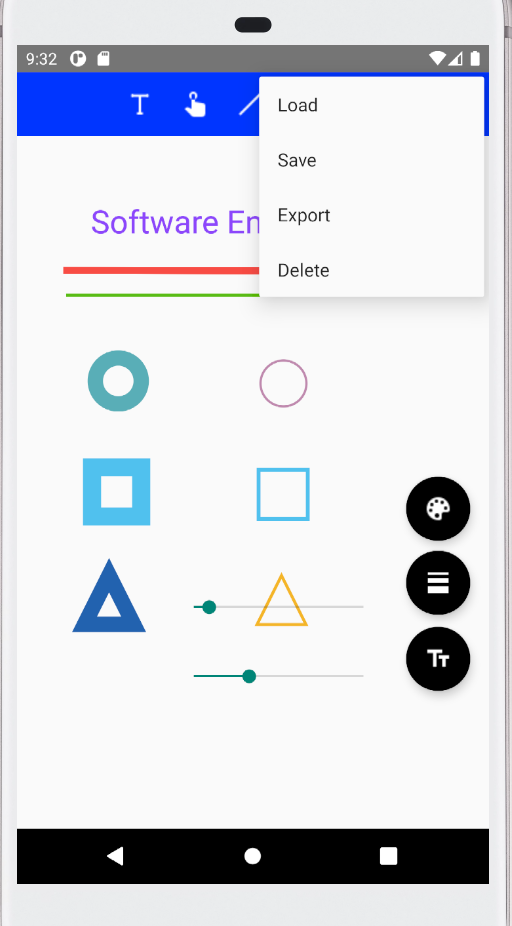
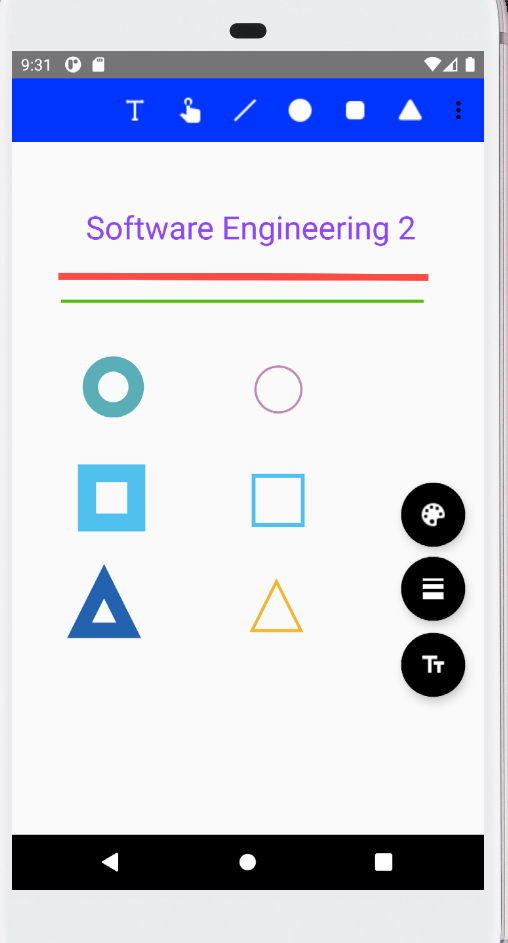
|  |  |
| --- | --- |
| **Team member 4** | |
| **Name:** | Leyla Durdyyeva |
| **Student ID:** | 01576824 |
| **E-mail address:** | a01576824@unet.univie.ac.at |

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

# Design Draft

## Design Approach and Overview

Our application presents itself to the user as a single screen on which all sketching-relevant functionalities (text field, freehand drawing & shapes) are attached to the upper menu in the AppBar. Additionally, on the right hand of this menu the user can access options like loading or deleting sketches. Lastly, on the bottom right corner we implemented three floating buttons which enable the user do further modify his drawings regarding size and colour.



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.

MODEL VIEW VIEWMODEL BILD

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 “graphical element”. From this class all sketching functionalities (freehand drawing, text & shapes) inherit their attributes and methods. Furthermore, this abstract class is connected to one of our two design patterns: “The Strategy pattern”. We implemented an Interface, which holds the information on how to create the different graphical elements. 🡪 Factory Pattern?

### Class Diagrams

Our class diagram follows the Model-View-Viewmodel pattern. Classes colored in green represent our View, classes in blue belong to the ViewModel and Classes colored in purple belong to the Model. Classes in grey will be implemented later.

Diagram

Description automatically generated

### Technology Stack

|  |  |  |  |
| --- | --- | --- | --- |
| 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. |

## Design Patterns

### Design Pattern 1: Factory Method 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.

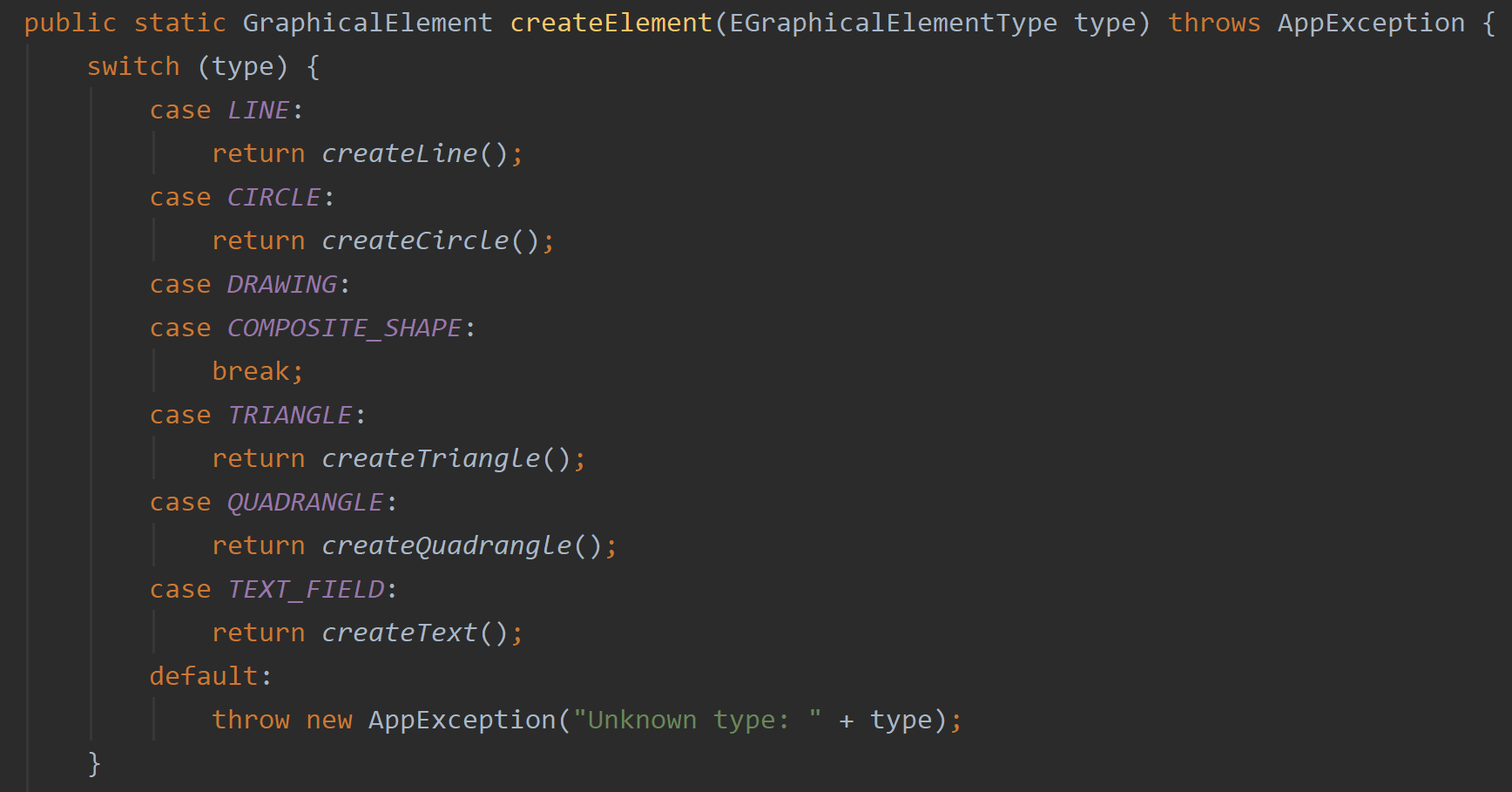
In our project we needed to implement the Factory Method pattern to create objects of GraphicalElement type from our Model. These objects are being later displayed on the View.

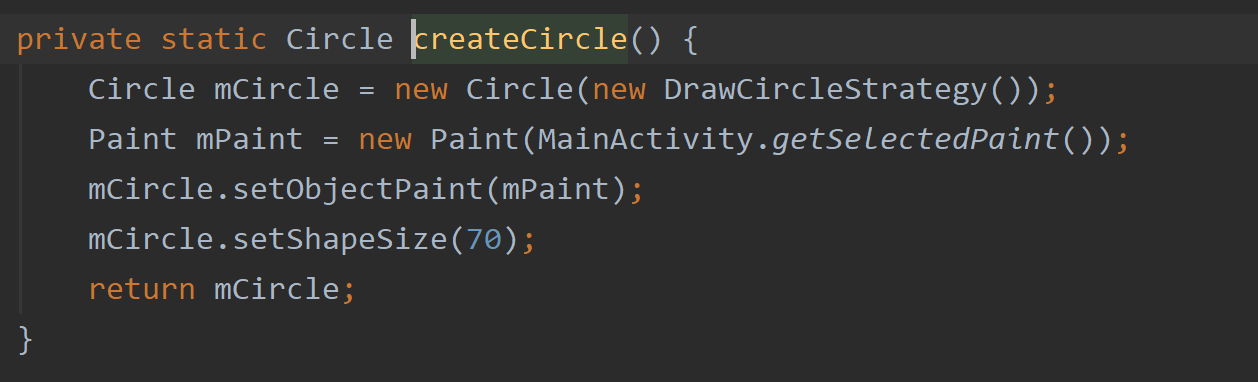
This design pattern relates to the Functional Requirements FR1-FR4, requiring new Graphical Elements to be created and displayed on the View. The creation happens after the user selects a graphical element. The creation is an intermediary step between the data saved in the Model and the representation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

Diagram

Description automatically generated

Code snippet (GraphicalElementFactory.java):





### Strategy Pattern

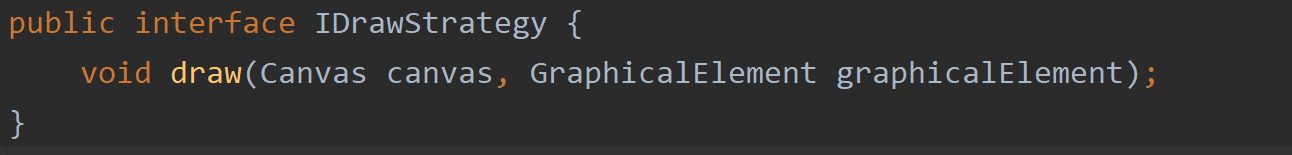
The Strategy Pattern is useful when there are multiple algorithms which solve a problem, and they can be used interchangeably, according to the concrete context. It is an alternative to implementing behavioural logic in subclasses of a Context, which has the benefit of separating business logic from 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.

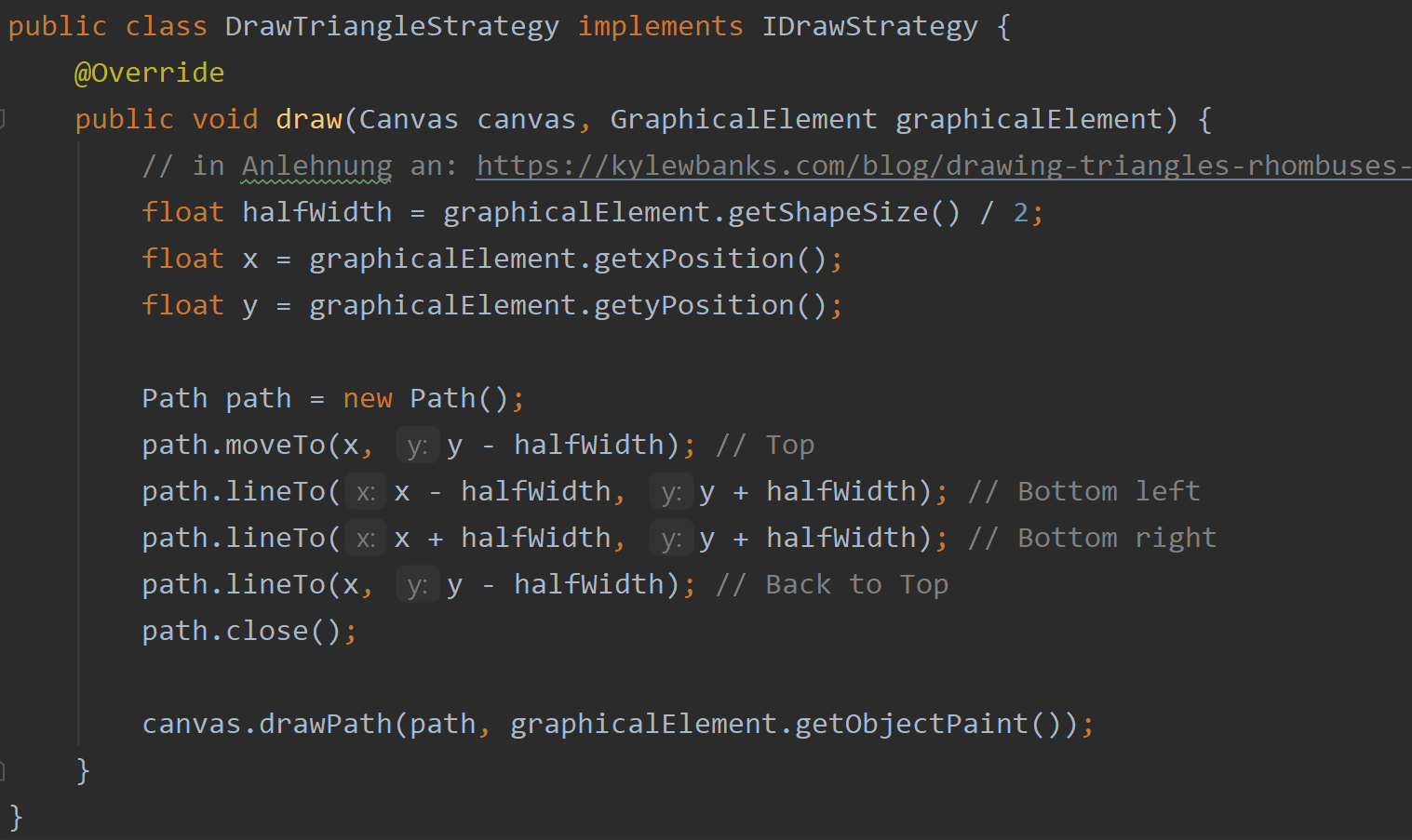
In our project, the Strategy Pattern turned out useful to avoid adding logic for drawing on the Canvas directly in our concrete subclasses of GraphicalElement. The logic for drawing is different for each graphical element, different Strategies have been implemented.

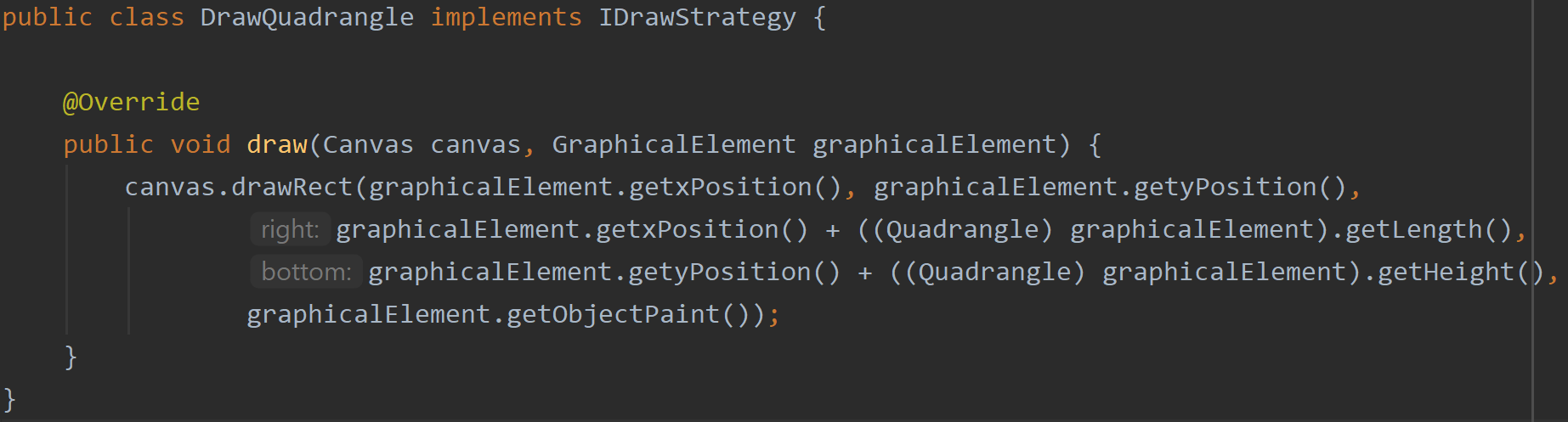
This design pattern relates to the Functional Requirements FR1-FR4, requiring new Graphical Elements to be created and displayed (drawn) on the View. The Strategy pattern is executed when the whole View is being drawn again, by going through each graphical element and invoking the corresponding Strategy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

Description automatically generated

Code snippet: (IDrawStrategy.java):







# Code Metrics

(Metrics calculated with the ‘Statistic’ plugin for Android studio)

Number of packages: 6

Lines of code: 1748

Comment lines of code: 70

Number of classes: 21

Code bugs and testing

Before running the final lint-test, the implementation showed around 70 errors. The majority of them were:

* Un-used resources which had piled up over the development process
* Display attributes of UI elements being missing (e.g. focusability)
* The non-local declaration of local variables
* Unused import-statements from the development process
* Usage of hardcoded values instead of flexible resources (strings)

The current implementation shows 21 warnings, whereas Java accounts for 8 and Android of the rest. The majority of Android errors recurs to our use of Resource ID’s in the switch-case statement in the main activity – which should be avoided in this context. We will eliminate this error and come up with another solution.

The rest of the warnings in the Android context recurs to minor aspects, we have considered these hints and came up with a suitable solution in these cases.

We have explicitly implemented three different Unit tests regarding methods in the Class CanvasViewTest for the setting of X and Y, as well as the selection of graphical elements. Through the latter, we gained an insight on the bug of selecting a text size/and or color when there has been no graphical element instantiated. This bug could be fixed. Furthermore, we had multiple phases of code reviews. Each team member checked the other program parts and used the TODO-function of Android studio to make suggestions on code improvement.

Moreover, we have conducted multiple End-to-End/Integration tests via the espresso testing framework. We have included a Shape construction test in the view folder on androidTest.

# Team Contribution

## Project Tasks and Schedule

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | **October** | | | | **November** | | | | **December** | | | | **January** | |
| JanPreparation/Planning |  | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **1** | **2** |
| 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 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DEAD submission |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## 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 | 2h |
| Strategy pattern implementation | Leyla | 3h |
| 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 | 6h |
| 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 | Felix & Christian | 3h |
| Code review | All | 10h |
| Documentation | Leyla, Christian, Felix | 9h |