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 |

# Final Design

## Design Approach and Overview

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

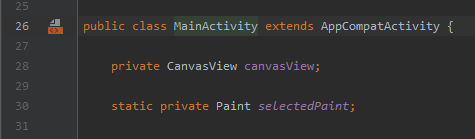
“Our internal software design is based on the Model – View – ViewModel approach. We thought, this might be a good idea, due to the model’s ability to provide cohesive, externally low coupled classes and structure highly interactive systems in a maintainable manner. The “View-part” of the model is responsible 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.”

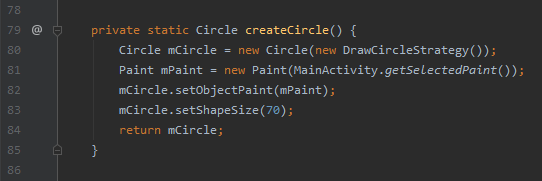
From the start on, we strived to structure our code in accordance with the Model-View-ViewModel approach. As we now have realized, our implementation at the point of SUPD was not well done and we did not quite grasp the concept in its entirety. While the approach was correct, the implementation could 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.

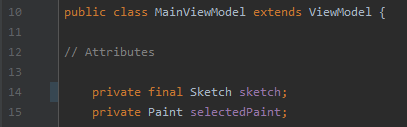
Code snippet: git: 36dfce0476edee184943df8547d6a644ca7ddb37 (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\view\MainActivity.java

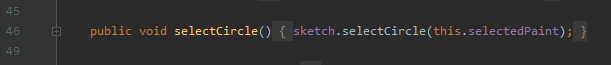




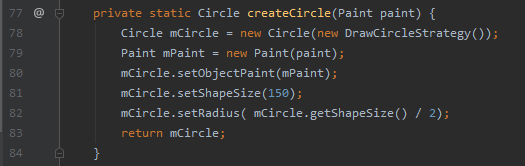
In the next iterative step, we move the attribute from the View class to the ViewModel class. From where it was further passed on to the GraphicalElementFactory.

Code snippet: git: 6de8bc6aa4f08ae30abf8d29b0de74d312575602 (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\viewmodel\MainViewModel.java



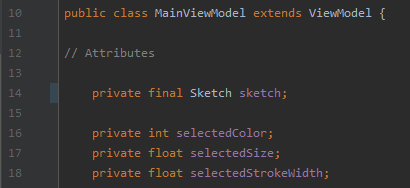


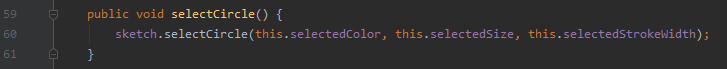
Code snippet: git: 6de8bc6aa4f08ae30abf8d29b0de74d312575602 (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\GraphicalElementFactory.java)



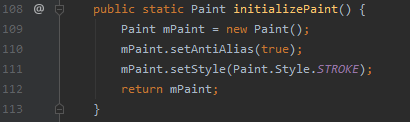
In order to adhere to the MVVM approach the creation of said paint object was moved to the model (“GraphicalElementFactory”). The ViewModel class only passed primitive values for colour, stroke width and size.

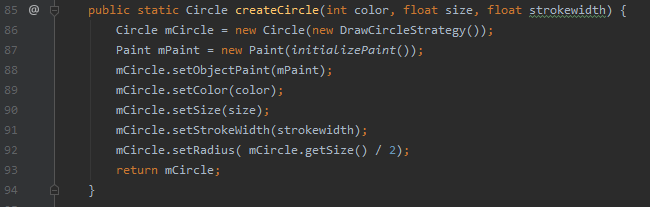
Code snippet: git: b107e7f9e88e87c64d2b2a8aa7d579f4deeebab4 (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\viewmodel\MainViewModel.java





Code snippet: git: b107e7f9e88e87c64d2b2a8aa7d579f4deeebab4 (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\GraphicalElementFactory.java)



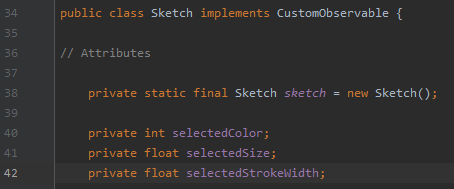


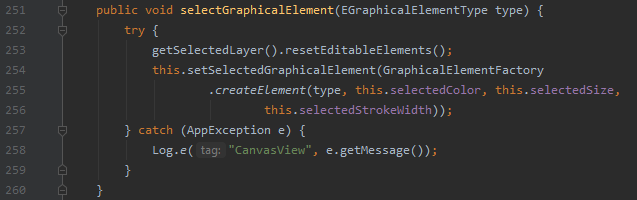
After another iteration of the paint object, we decided that we want to create the paint object just in time when it is necessary. Hence, it is created within the concrete “DrawingStrategy” classes where the canvas method “draw” relies on having a paint object. Outside of these classes, we only work 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.

Code snippet: (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\viewmodel\MainActivityViewModel.java

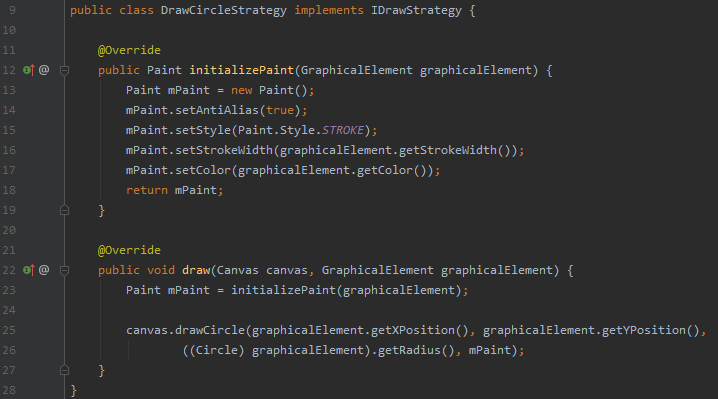


Code snippet: (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\Sketch.java)





Code snippet: (\implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\draw\DrawCircleStrategy.java

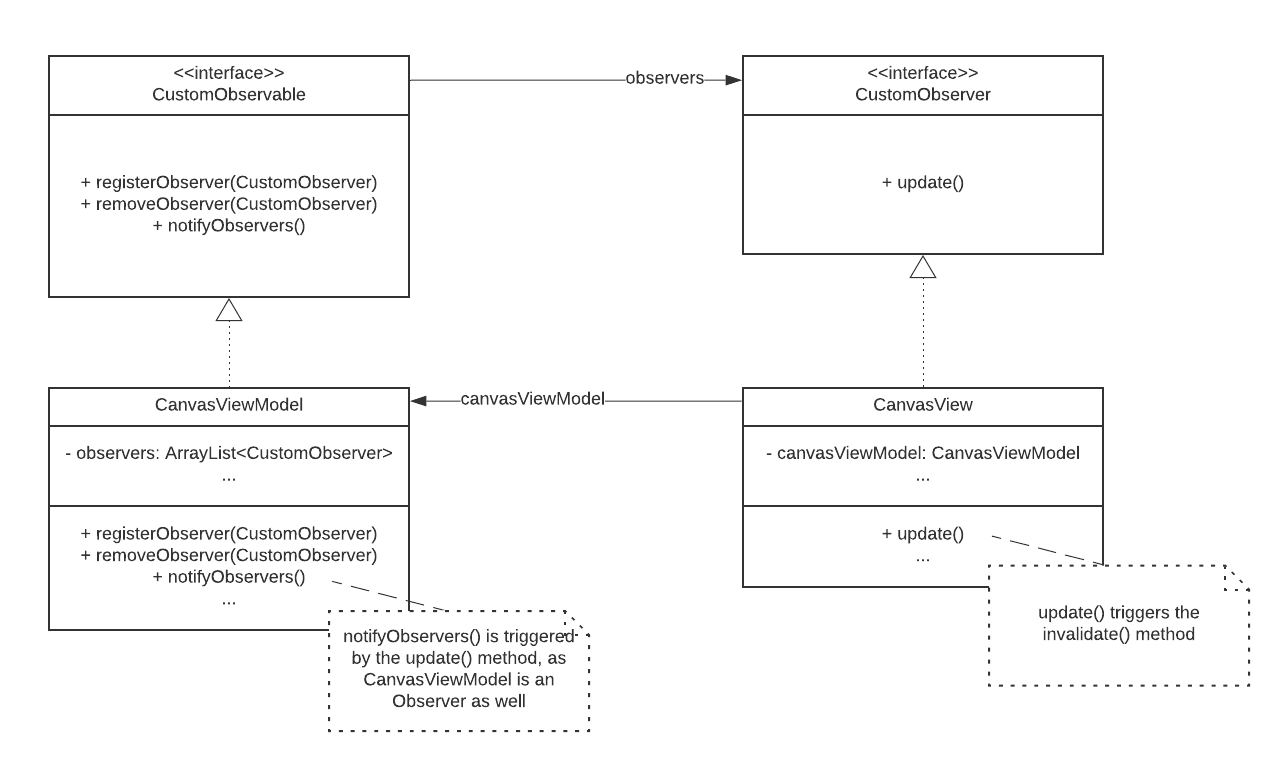


This leads us to another active design decision. We felt that our ViewModel class was overloaded, so we decided to divide it into two classes, one that is related to the “Canvas” class itself and one for operations that deal with buttons and menus. Hence, the previously mentioned View and ViewModel classes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

**UML NEU VON CANVAS UND VIEWMODEL aus dem klassendiagram**

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” and “MainActivityViewModel” react on changes in Model classes, while “CanvasView” and “MainActivity” react on changes in the ViewModel classes.

Another influential decision is connected to the implementation of the iterator pattern. 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.

Code SNippet neu und code snippet alt

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 since 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.

**SNIPPET**

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 **???.** At a certain point of our implementation, this was not long 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. Thereby all objects can stay at their place within the “ElementCollection” and are still distinguishable, due their unique indices.

Code SNippet neu und code snippet alt

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.

**UML DIAGRAMM DRAFT?**

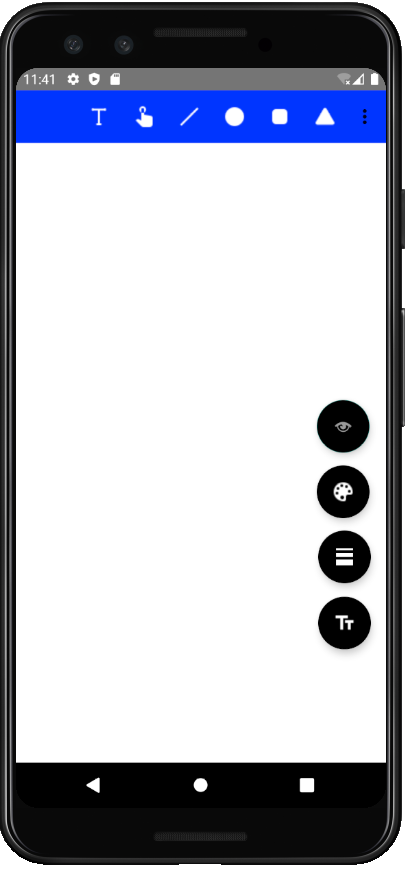
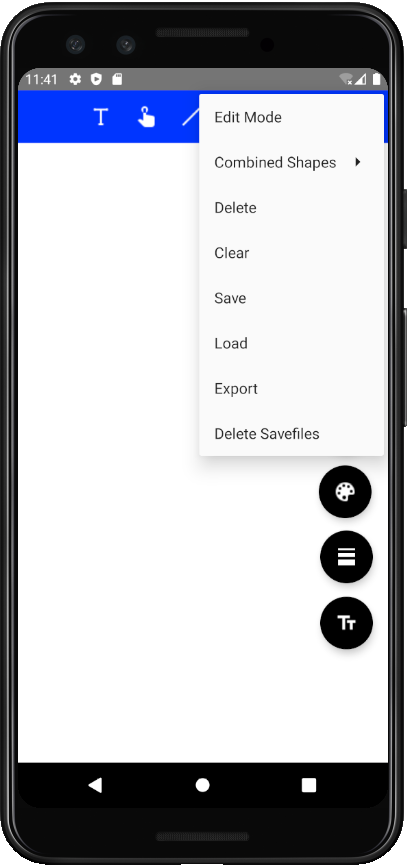
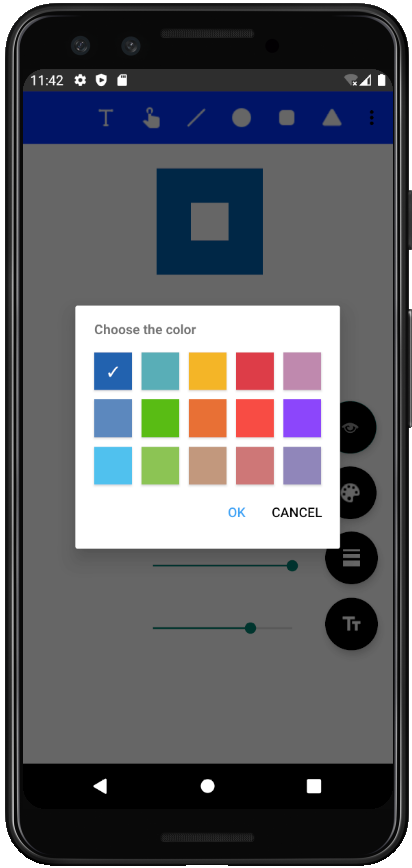
Regarding the technology stack, we only added two third party libraries before testing our implementation. The first one is the ColorPicker which was introduced early on, in order to provide a reliable source for modifying the colour of our graphical elements. Alternatives like the Holopicker and the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

**SNIPPET COLOUR picker**

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.

**SNIPPET gson**

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., saving, loading and clearing 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.

SNIPPET selectGraphicalElement

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 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 describes in 1.3).

SNIPPET

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

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” variable is added to the data collection on the active layer which is called “ElementCollection”. Also, a corresponding index is added to an additional data collection. Hence, the object is distinct and we can easily refer to it. Invoking the “setCoordinates” method in the “Sketch” class adds the coordinates which we received from the “TouchDownEvent” on the CanvasView to our selected graphical element. Then we use the iterator pattern (see 1.3 for more details) to traverse over the data collection of each active layer.

SNIPPET

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.

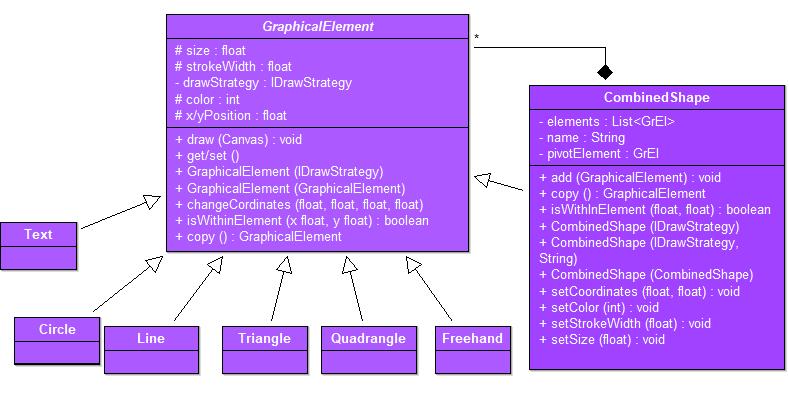
UML DIAGRAM

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.

UML DIAGRAM oder SNIPPET

Once, this chain of observers is notified, based on the described change in the business logic, the “onDraw” method in the “CanvasView” is invoked which in return triggers

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).



Decorator,

Template MEthode

The other

SNIPPET

### Class Diagrams

Class diagrams (for example, see Figure 1) describe the structure of a design. Try to omitt unnecessary detail (e.g., getter and setter operations), and focus on giving a good structural overview. You may also use multiple levels of details (zoom levels), or present multiple class diagrams showing parts of the overall solution that are (mostly) independent from each other.

Ein Bild, das Text, Karte enthält.

Automatisch generierte Beschreibung  
Figure 1: Sample Class Diagram. TODO: remove in final submission!

### 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  For the test cases, we used the Frameworks of JUnit and Mockito. 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. |
| Junit | https://junit.org/junit4/ | [4.13.1](https://github.com/junit-team/junit4/blob/HEAD/doc/ReleaseNotes4.13.1.md) |

We tested our application with the “Pixel\_3\_API\_30” Emulator. The corresponding configurations are attached at the end of the report.

## Major Changes Compared to SUPD

Give a list of major changes of your software project compared to the SUPD milestone (briefly summarize and possibly reference the SUPD report contents and/or source code paths). In case you have been given recommendations in the SUPD feedback, please briefly report on how you have implemented them.

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 more.

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. This structural design is enabled by the observer pattern. 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 has to be updated. 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

SNIPPET alt und neu

Addressing the implementation of layer objects, as stated in section 1.1, this can be seen as 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.

UML DIAGRAM ODER ETWAS VERGLEICHABRES

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:???

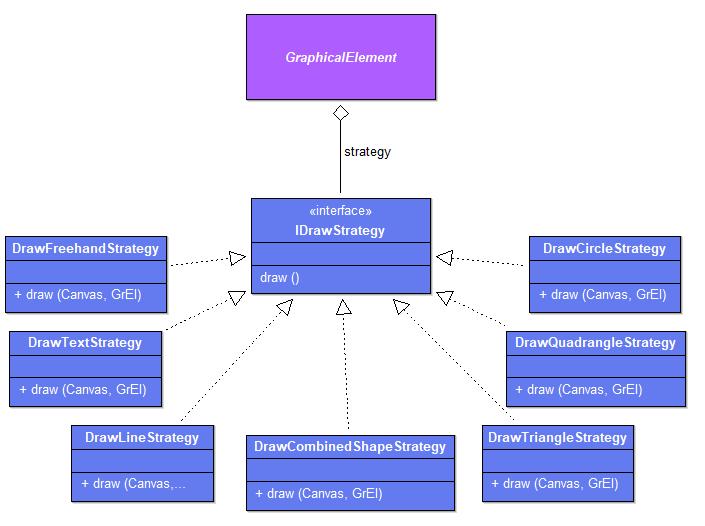
SNIPPET alt und neu

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.

## Design Patterns

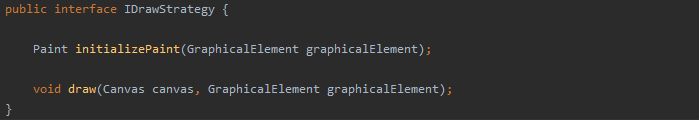
### Strategy Pattern

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.



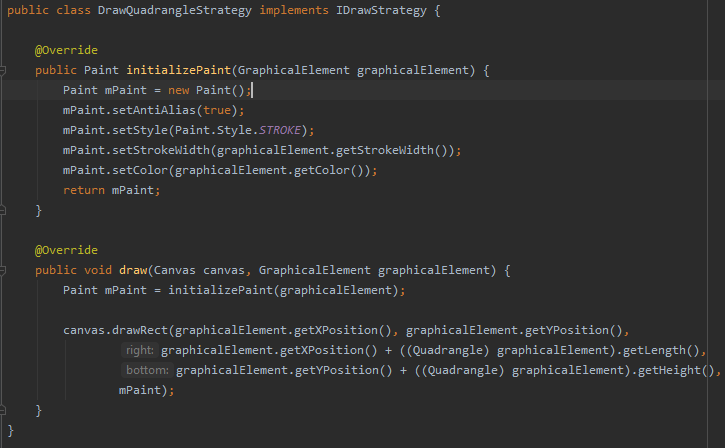
In our project, the Strategy Pattern turned out useful to avoid adding logic for drawing on the Canvas directly in our concrete subclasses of “GraphicalElement”. The logic for drawing is different for each graphical element, different Strategies have been implemented.

*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\draw\IDrawStrategy.java)* 

*Code Snippet:*

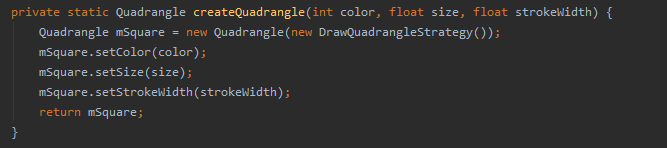
*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\draw\DrawQuadrangleStrategy.java)*



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.

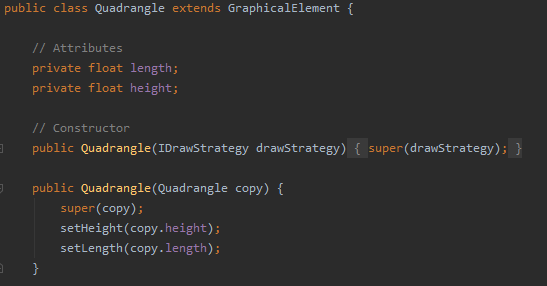
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\*GraphicalElementFactory.java)



*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\graphicalElements\Quadrangle.java)*

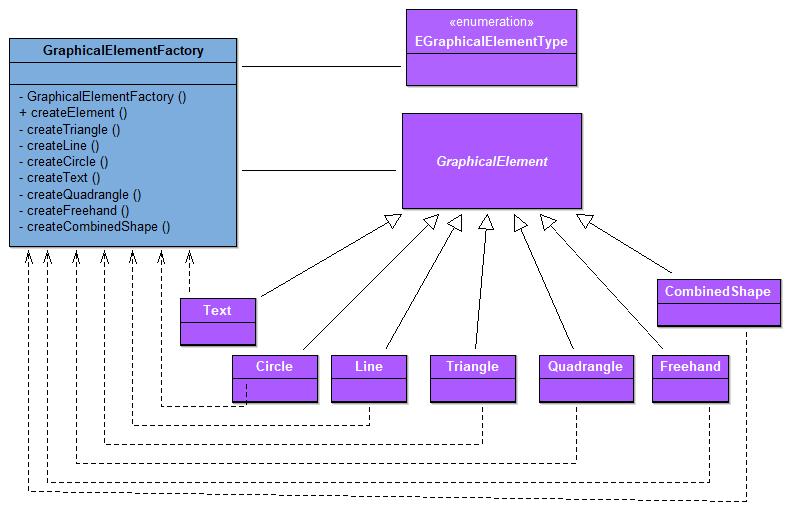


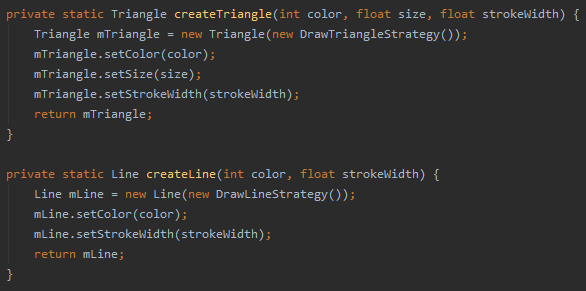
### Factory Pattern

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.

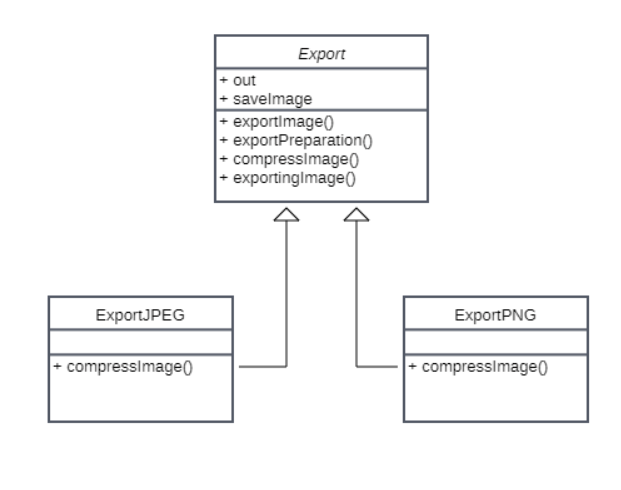
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\*GraphicalElementFactory.java)



### Template Methode Pattern

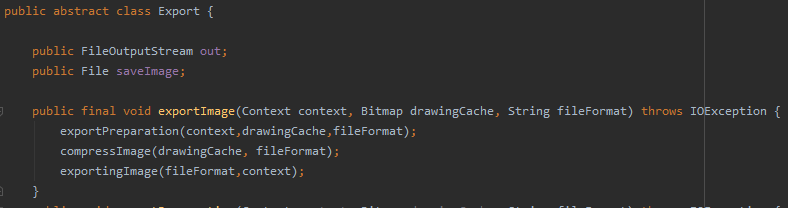
The template method pattern defines a guide how a certain set of methods should be carried out and distributes the individual tasks to itself and more specific subclasses. It consists of an abstract class which describes a template for a process, by declaring a final method with predefined functional 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.

*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\export\Export*.java)



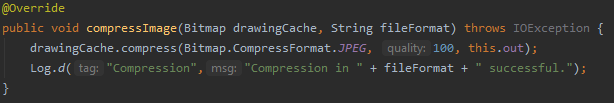
We created the abstract class “Export” which inherits its properties to its children classes: “ExportJPEG” and “ExportPNG”. Furthermore, we divided the export process in three steps: “exportImage”, “compressImage” and “exportingImage”.

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.

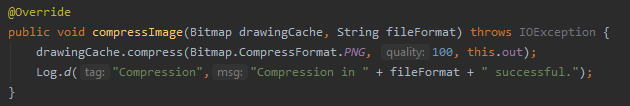
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\export\ExportJPEG*.java)



*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\export\ExportPNG*.java)

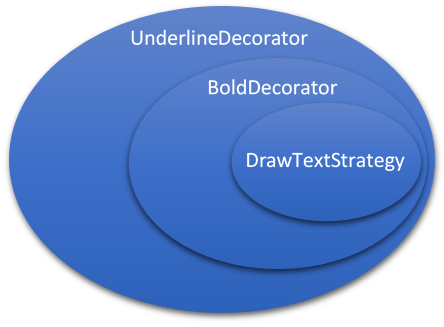


Thus, there is no duplicate code and the general approach to the export of files remains unchanged. Furthermore, the client has no contact to the subclasses, since they are only called by the abstract class itself, resulting in well encapsulated algorithms.

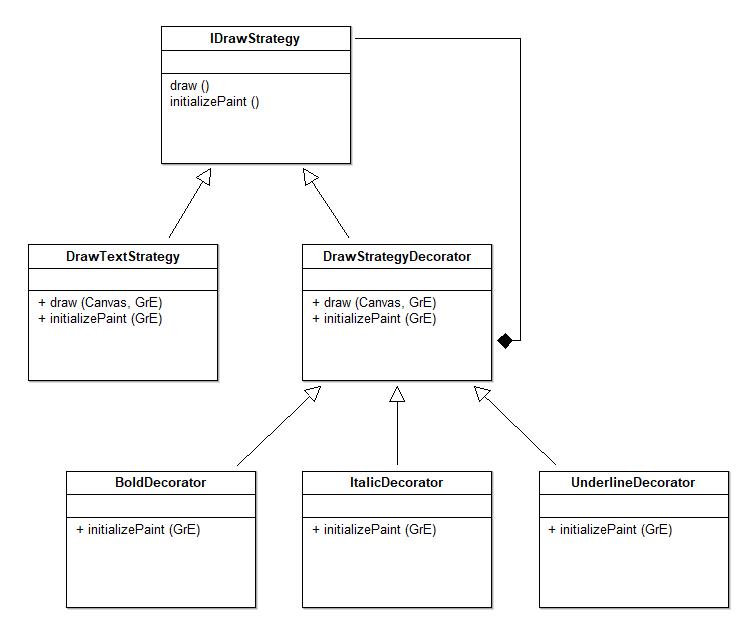
### Decorator Pattern SANDRA

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.

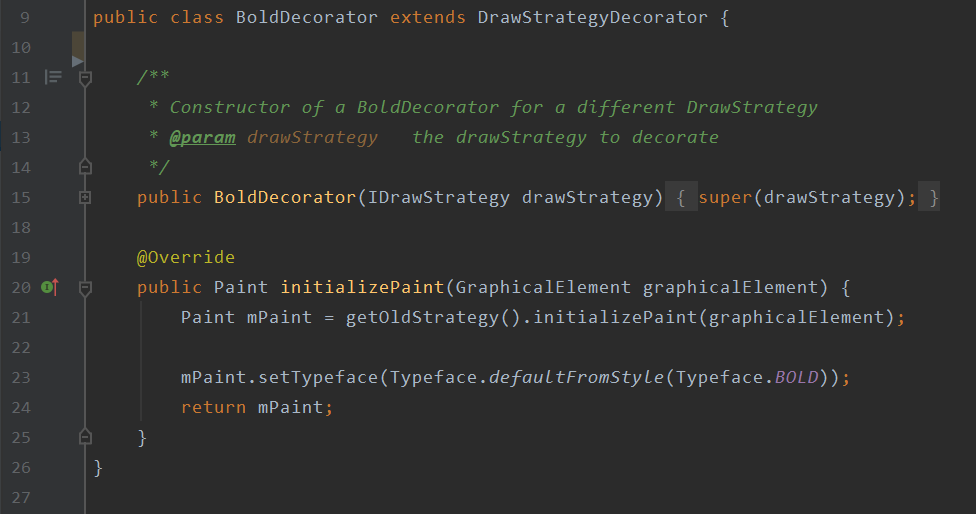
We decided to use the decorator pattern to decorate text objects with different styles.



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".

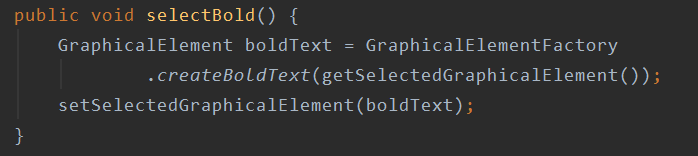


The "DrawStrategyDecorator" holds an attribute oldstrategy, which stores the drawStrategy that is currently in use. The concrete decorator classes, for example "BoldDecorator" then call the initializePaint method of the drawStrategy and add the desired functionality, here for example bold text.

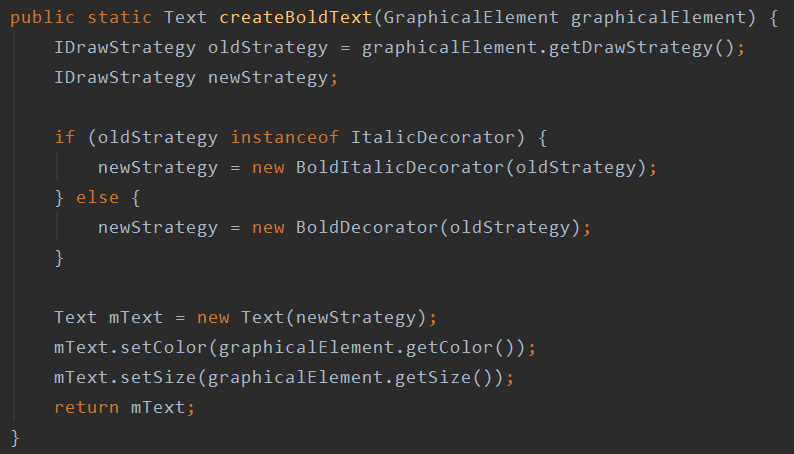


This explains the underlying logic.

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.



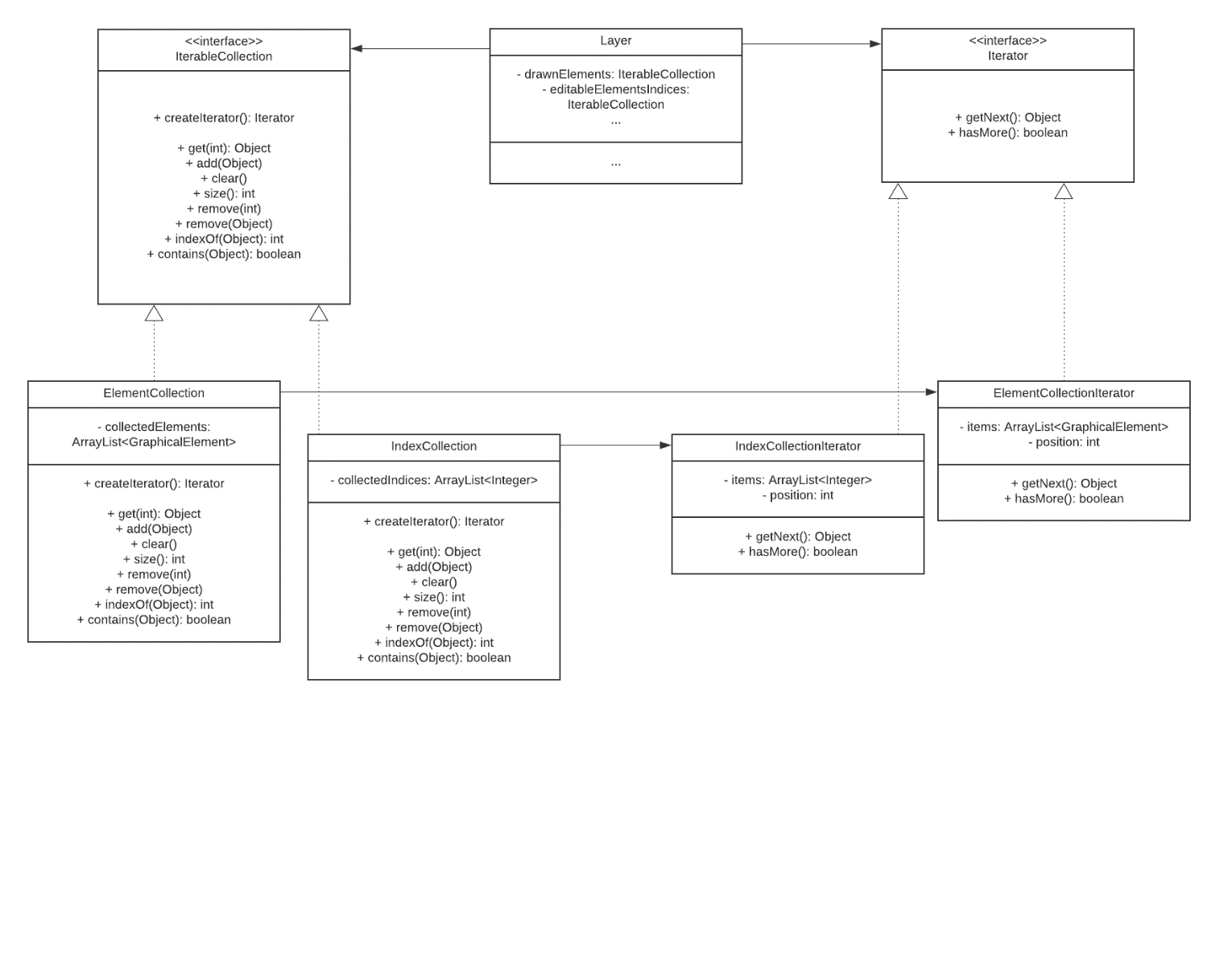
These methods then call “create” methods in the "GraphicalElementFactory", such as the “createBoldText” method here.



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

### Iterator Pattern

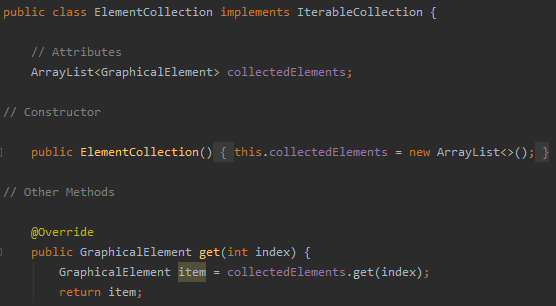
The iterator pattern is used to iterate over any kind of data collection without exposing what kind of collection it is currently traversing. The iterator pattern relies on two interfaces. Firstly, the iterator interface is created which has all the required methods for traversing over a data structure. 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.

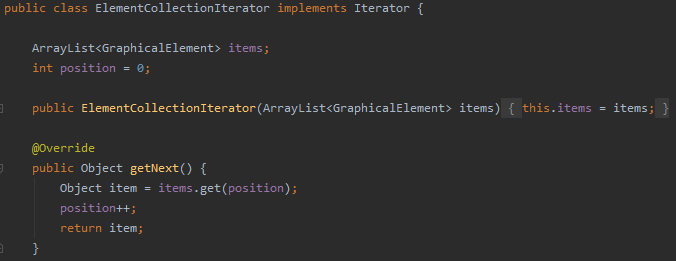
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\iteratorsAndCollections\ElementCollection*.java)



*Code Snippet:*

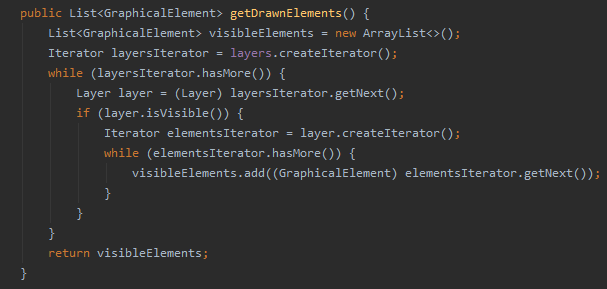
*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\iteratorsAndCollections\ElementCollectionIterator*.java)



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.

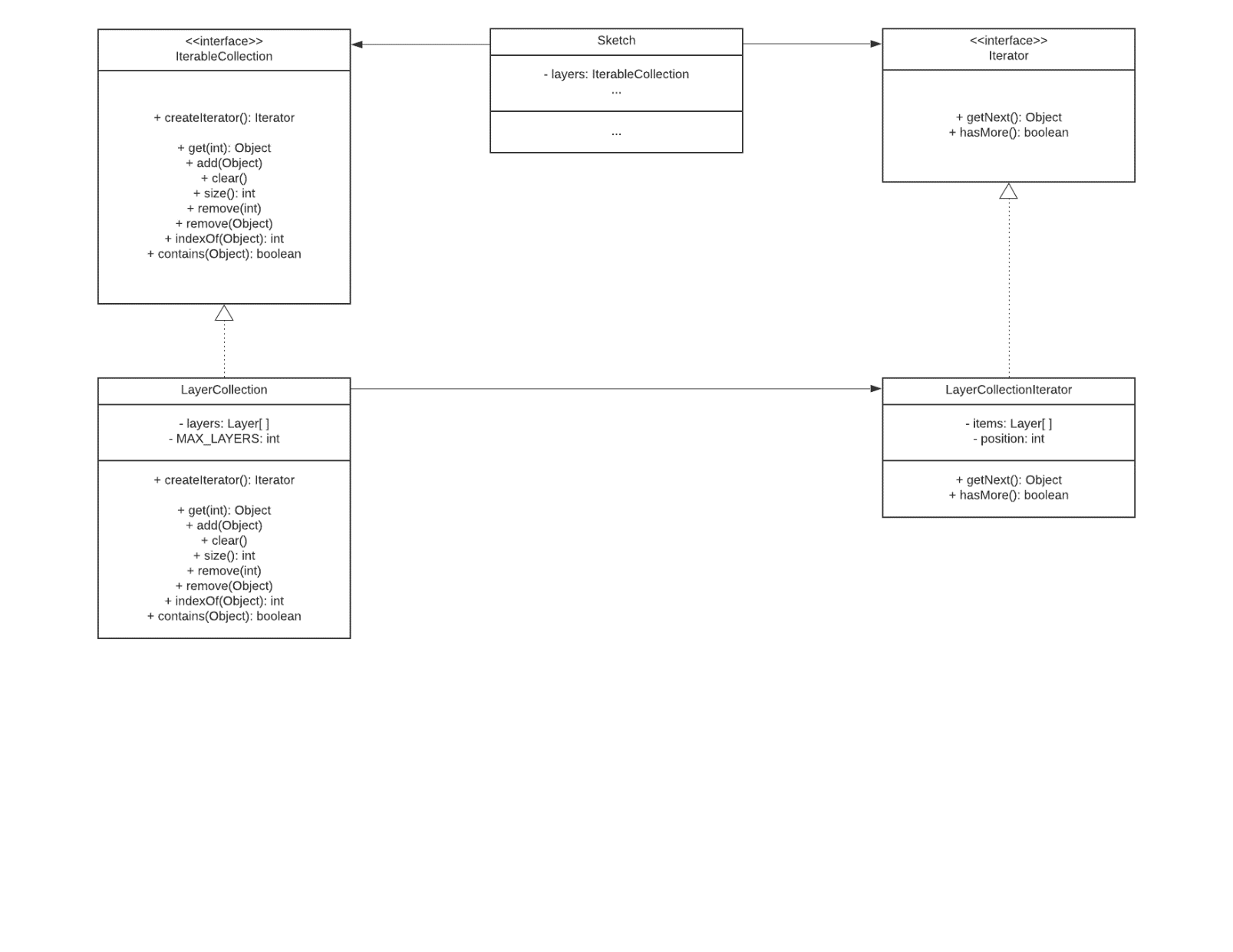
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\Sketch*.java)



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.

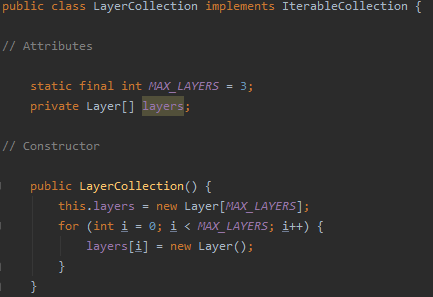
the iterator pattern lets us traverse over our different data collections, without having to treat them differently. This is especially useful, since there are several kinds of different data collections that are being traversed. The iteration process always looks the same and complex traversal methods 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.

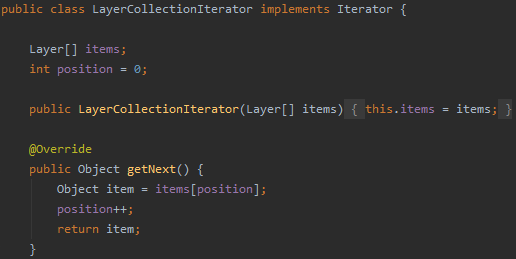
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\iteratorsAndCollections\LayerCollection*.java)



*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\iteratorsAndCollections\LayerCollectionIterator*.java)



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).

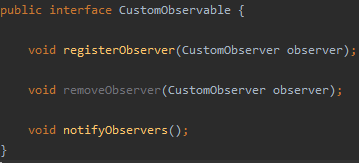
### Observer Pattern

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.

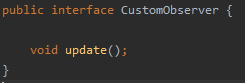
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\iteratorsAndCollections\observerPatternInterfaces\CustomObservable*.java)

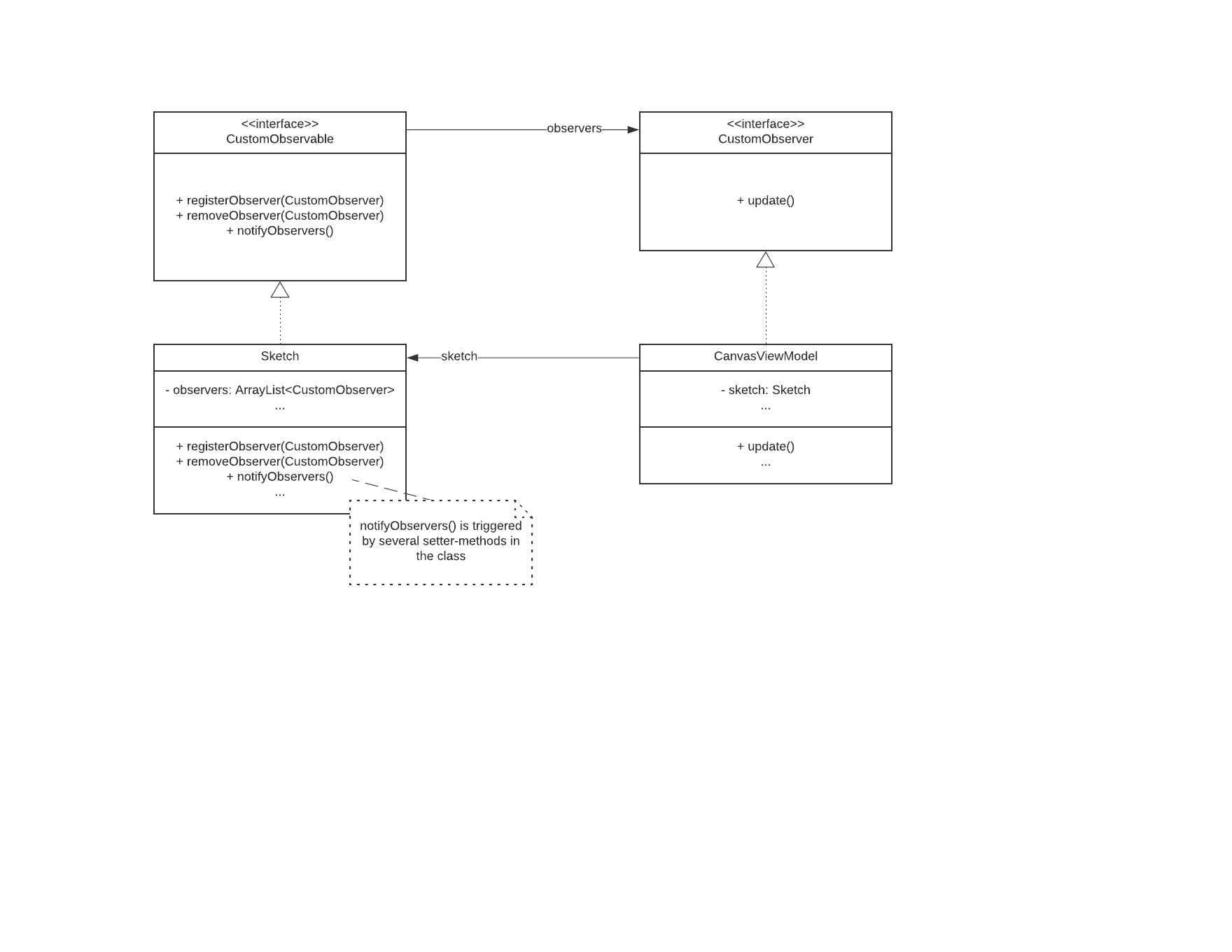


*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\iteratorsAndCollections\observerPatternInterfaces\CustomObserver*.java)



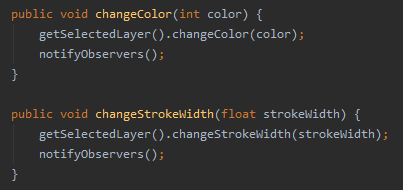
The interface “CustomObservable” acts as subject-interface and the interface “CustomObserver” as observer-interface. Both contain the methods that were discussed in the previous section. We have two occurrences of the observer pattern which are closely connected and are therefore, described as one. Both 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.



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.

*Code Snippet:*

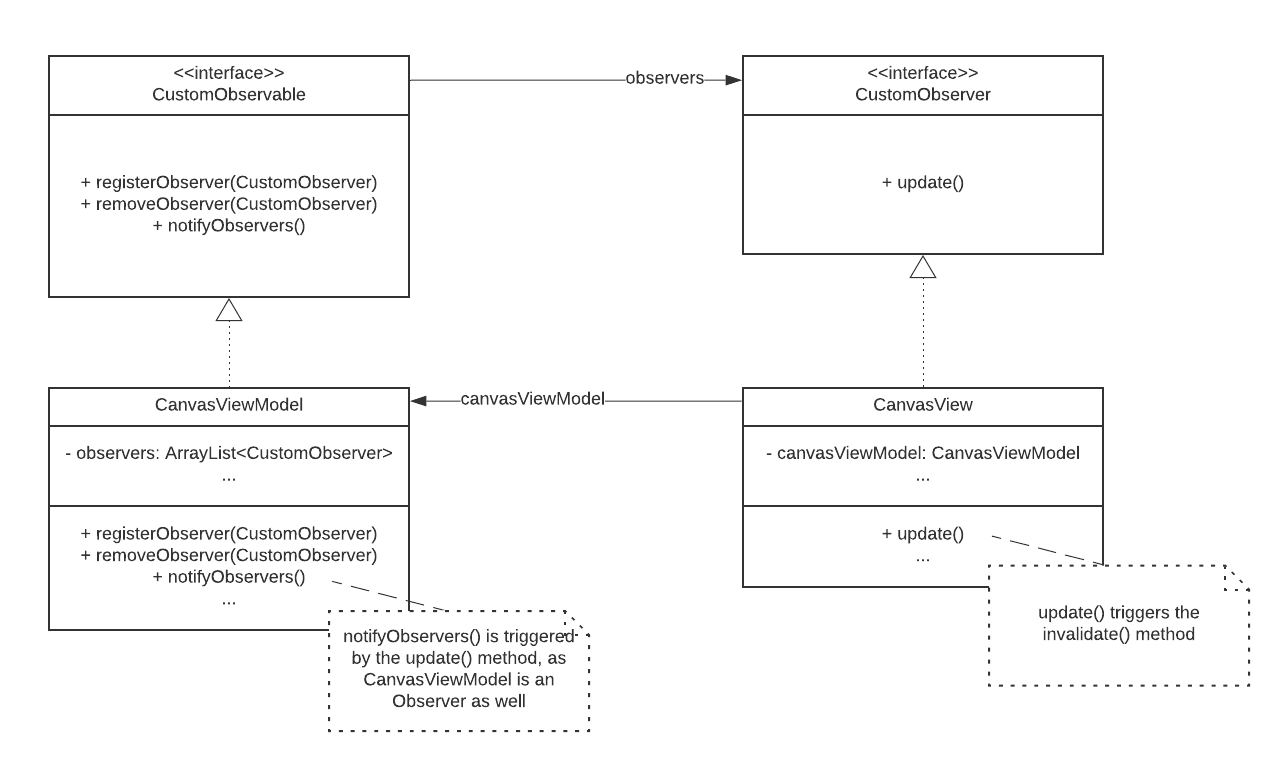
*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\Sketch*.java)



*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\viewModel\CanvasViewModel*.java)

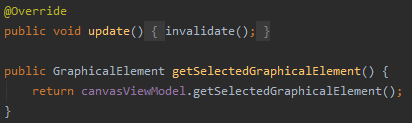




This is where the second application of the observer pattern comes into play. Our “CanvasViewModel” class does not only realize the “CustomObserver” interface and observes the “Sketch” class. It also realizes the “CustomObservable” interface, implements all its methods and therefore, has a double function 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.

*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\view\CanvasView*.java)



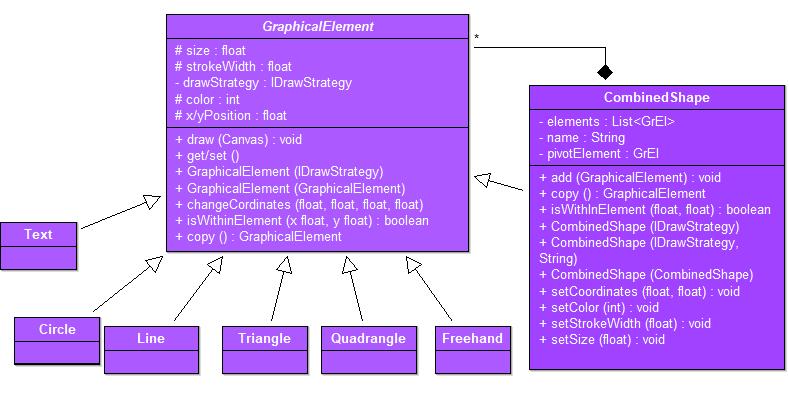
The observer pattern relates to FR 1 – FR 9, since the pattern is employed in any kind of action that involves an update of the display (e.g., modification, addition of objects).

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.

### Composite

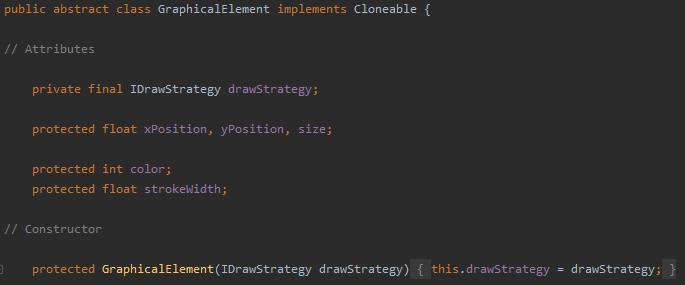
The composite pattern enables programmers to treat objects from a part-whole hierarchy, uniformly. This is achieved by connecting primitive “leaf classes” and more complex “container classes” to a shared interface. Containers can hold any number of primitive objects and each request is passed down to 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.



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 is also achieved since modifications are possible for combined shapes aswell. 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”.

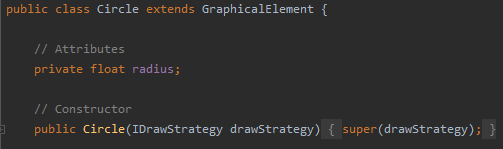
*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\graphicalElements\GraphicalElement*.java)



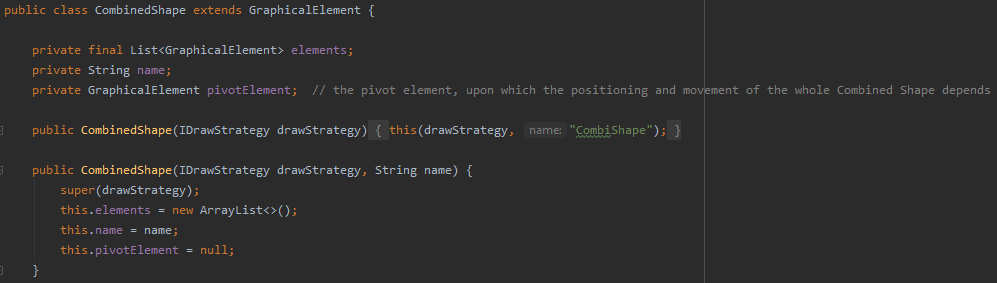
*Code Snippet:*

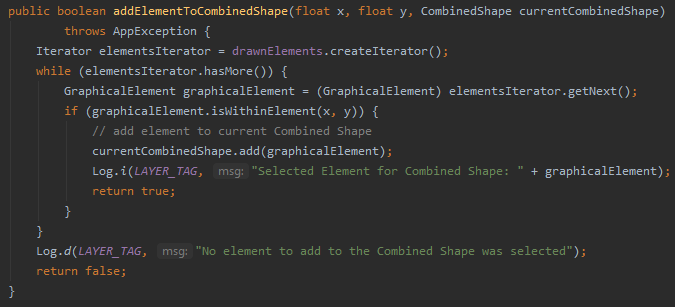
*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\graphicalElements\Circle*.java)



*Code Snippet:*

*(implementation\Sketch\_App\app\src\main\java\at\ac\univie\se2ws2020team0310\sketch\_app\model\graphicalElements\CombinedShape*.java)





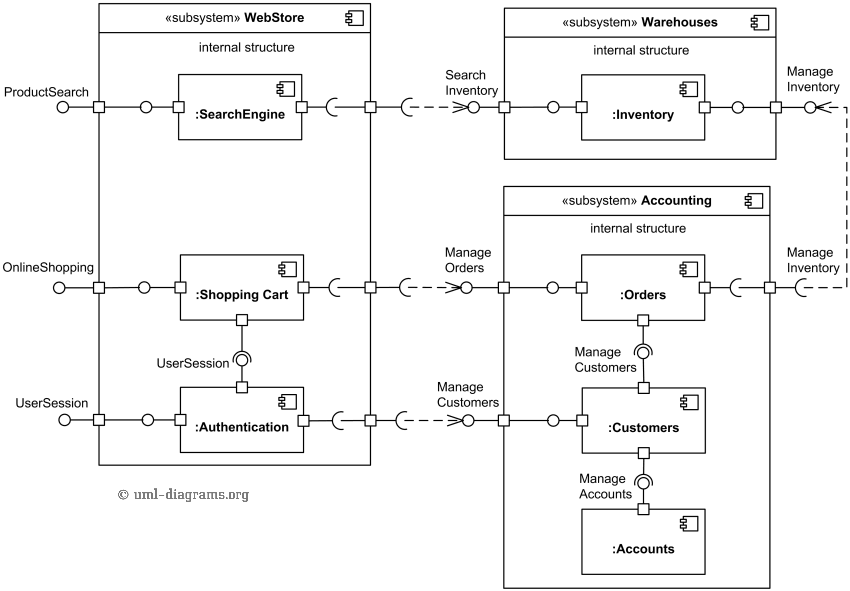
This interface enables the client to treat primitive and compound objects alike, by passing the request t. Container classes consist of interfacelike prcombined graphical elements should be editable in the same way as single graphical elements.--> All via the list (layer)

abstract class that represents primitives and container (Compound shape / picture)🡪 graphical element defines methodes for all graphical elements. Picture is an aggregate of graphical element

# Implementation

## Overview of Main Modules and Components

Give a brief overview of the implementation of your design. Besides a textual description you may use UML component diagrams (see for example, Figure 2).

  
Figure 2: Sample Component Diagram from : <https://www.uml-diagrams.org/examples/online-shopping-uml-component-diagram-example.html> TODO: remove in final submission!

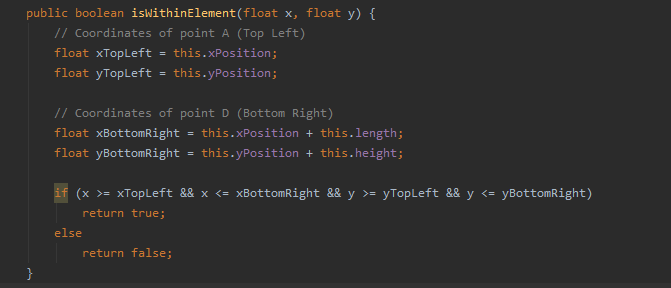
## Coding Practices

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:

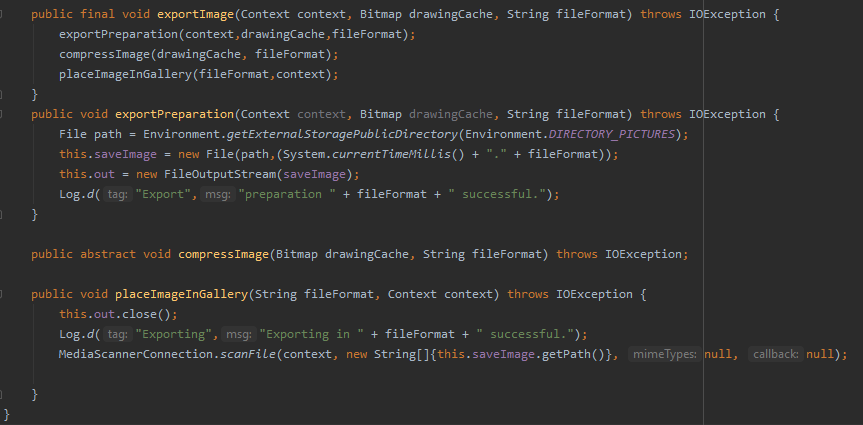
Names should be intention revealing and avoid disinformation. We chose names that describe as close as possible what is inside a variable or what the task of a specific methods is. Thereby, we reduced the necessity of commenting and improved readability as well as maintainability.

*Code snippet: (Circle.java):*



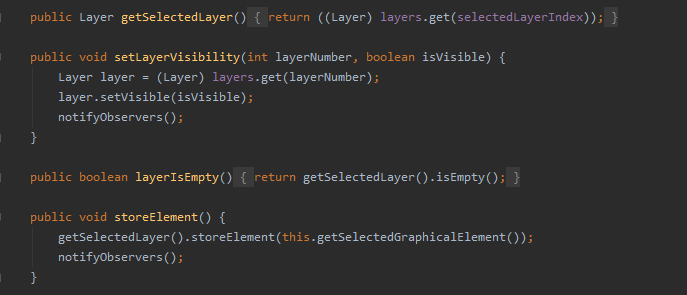
Furthermore, our names are making meaningful distinctions between comparable or connected methods where it is feasible. Therefore, a reader does not get confused, due to similar naming schemes.

*Code snippet: (Export.java):*



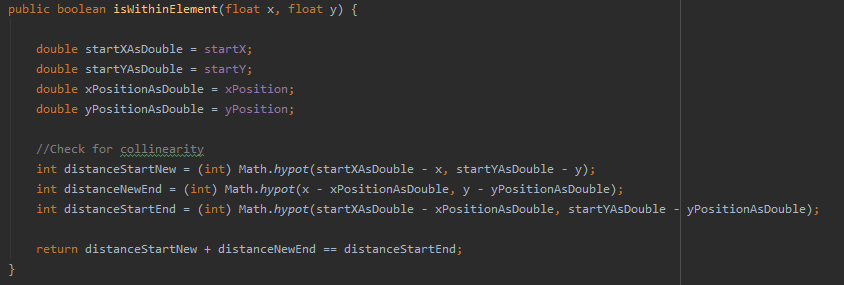
Lastly, our names are pronounceable and searchable which also enhances readability and maintainability of the code. While creating the app, this circumstance was very helpful, since we collaborated mostly virtually and being able to search for currently discussed methods or pronounce them without difficulties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

*Code snippet: (Sketch.java):*



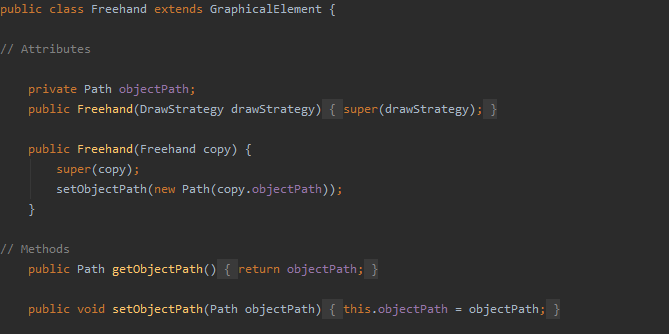
Another major player in our code is commenting. One the one hand, there are ordinary comments which explain situations and details which are not obvious or exceed the explanatory capabilities of names. On the other hand, we documented with JavaDocs which we will cover in section 3.3.

*Code snippet: (Line.java):*



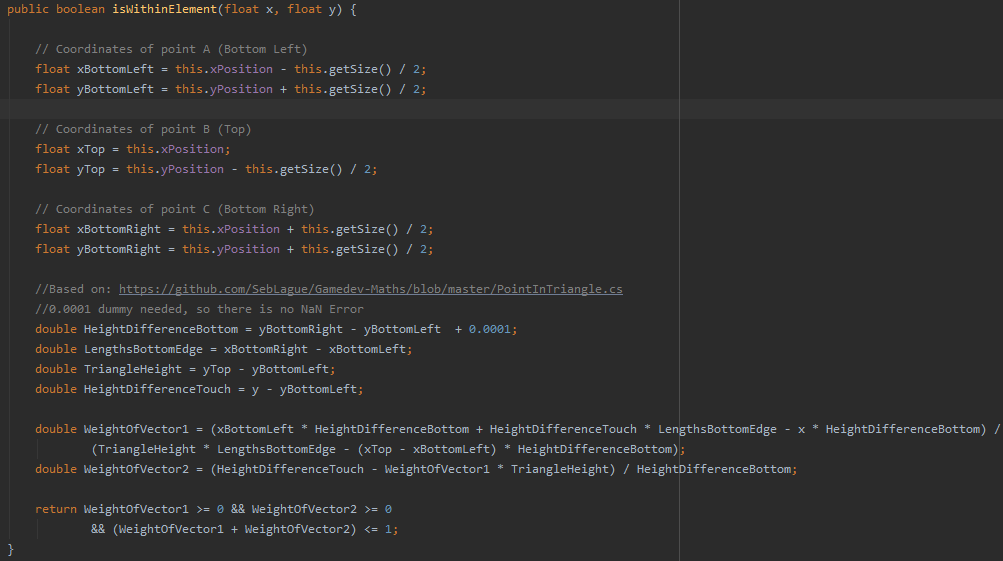
In general, we tried to employ a low number of additional comments. However, we also added comments for the general structure of our code which might not be a proven approach, but in our opinion, it had a positive effect on the readability of our code.

*Code snippet: (Freehand.java):*



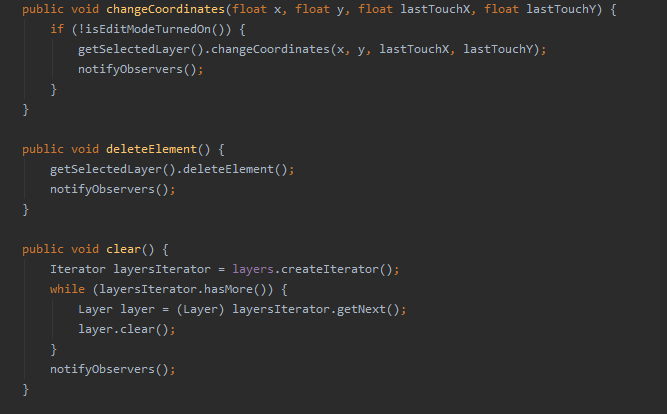
Moreover, comments are used to mark code sections for which we our solution is heavily influenced by existing logic as described in other sources.

*Code snippet: (Triangle.java):*



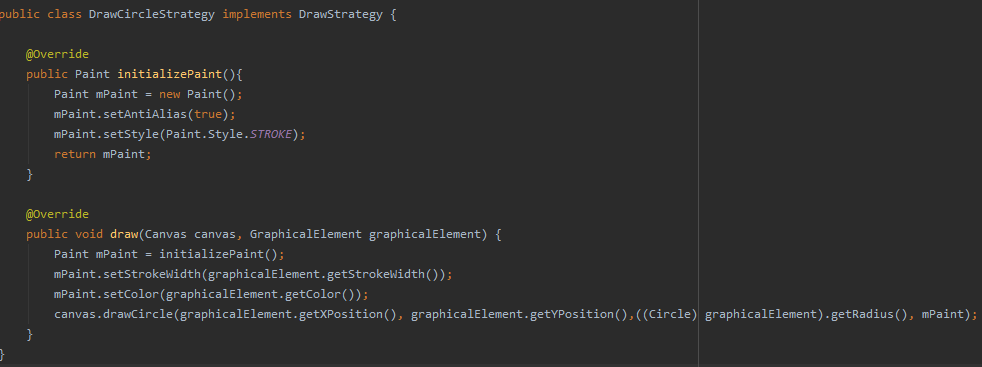
The appearance of our code in terms of form and style is oriented towards the Google Java Style Guide. This is also covered in greater detail in section 3.3. We consistently use an indent size of 4 and utilize indents to logically structure our code. Furthermore, we paid attention to the alignment of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

*Code snippet: (Sketch.java):*



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.

*Code snippet: (DrawCircleStrategy.java):*



## Defensive Programming

Discuss how and to what extent you have considered defensive programming in your project implementation. Discuss and show examples from your code.

Purpose in general:♣ Leave in code that checks for important errors ♣ Remove code that checks for trivial errors ♣ Remove code that results in hard crashes ♣ Leave in code that helps the program crash gracefully ♣ Log errors for your technical support personnel ♣ Make sure that the error messages you leave in are friendly

Assertions: ( Assertions are used to handle errors that should never occur in the code)

Errorhandling techniques (Return a neutral value, Substitute the next piece of valid data., Return the same answer as the previous time, Substitute the closest legal value🡪 Log a warning message to a file ♣ Return an error code ♣ Call an error-processing routine/object ♣ Display an error message wherever the error is encountered ♣ Handle the error in whatever way works best locally ♣ Shut down

Exceptions: errors that can be ignored by the system

# Software Quality

## Code Metrics

Provide information on at least the number of packages, lines of code, comment lines of code, number of classes and code bugs with regard to your current state of the implementation. It is recommendable to use a static code analysis tool (e.g. SpotBugs). Compare the metric results with your SUPD results and discuss your findings.

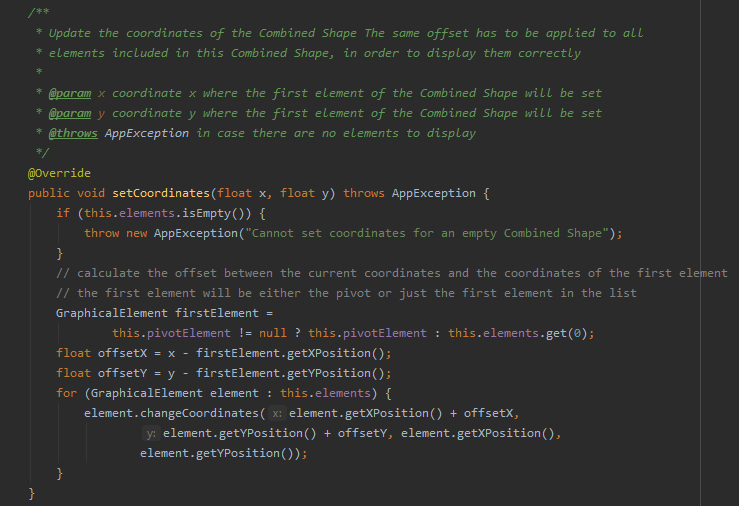
## Testcases for Functional Requirements

Discuss for each functional requirement how one or more of your testcases cover/s it. In case of limitations document them!

## Quality Requirements Coverage

Discuss for each quality requirement what activties reported and/or what artifacts implemented in your deliverables cover it. Only summarize briefly and provide references to report sections, or source code files.

**QR1 Comment your code and provide code documentation in an appropriate manner:**

* Code snippets with comments, Javadoc. Welche Methoden oder Klassen wollen wir mit JavaDoc kommentieren?--> eigenständiges Kommentieren und dann review in der Gruppe
* 
* Irgendwo in sketch?

**QR2 Your implementation must be in compliance with a style guide:**

* Reasons for using a **style guide**: Be consistent within the team, outer teams or distributions are not important to us: “Within this article, we have briefly illustrated how to easily and quickly integrate the Google Java Style Guide in a Java project. Even though this convention is widely spread it’s not the only candidate, you may also consider the [Sun Code Conventions](https://checkstyle.org/styleguides/sun-code-conventions-19990420/CodeConvTOC.doc.html) and [Checkstyle](https://checkstyle.sourceforge.io/), even though it’s antiquated, verbose and quite difficult to integrate.”

<https://medium.com/@alexprut/integrate-google-java-style-guide-in-a-java-project-567abb6d7987>

https://github.com/autyzm-pg/friendly-plans/wiki/How-to-install-the-Google-Style-Guide-settings-in-Android-Studio

**QR3 Apply common coding practices. 🡪 Summarize chapter 2.2**

**QR4 Apply defensive programming**. 🡪 Summarize chapter 2.3

**QR5 Apply key design principles**

Abstraction means to focus on the essential features of a design element

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules

Information Hiding / Encapsulation: as follows: “Every module … is characterized by its knowledge of a design decision which it hides from all others

Separation of concerns: Different aspects of a problem should be separated from each other, and each aspect of the problem should be treated on its own

**QR6 Testing and other general quality measuring tools** (informal technical review, “formal” technical review, peere review – four eye principle)

**GENERAL DESIGN PRINCIPLES:**

Not important for this chapter but important in general: Did we make propper use of the general design Requirements:

• At least 1 interface (with one or more implementations)

* Idraw strategy

• At least 5 classes (implementing the application logic)

• At least 1 abstract class (with subclasses)

• At least 1 custom exception class (with usages in your code)

• At least 1 level of depth in inheritance

GDR3 Apply principles of and follow best practices for object-oriented design, such as:

• Use data encapsulation

• Use inheritance, abstraction and polymorphism properly, e.g., follow SOLID6

* Polymorphism im strategy pattern: es gibt 5 mal draw, die alle etwas verschiedenes / factory: wir rufen nur eine einzige Funktion Draw auf, die quasi verschiedene Ausprägungen hat
* Overriding of setter in graphicalElement: Line

• Favor object composition over class inheritance

• Use exception handling

• Program to an interface, not an implementation

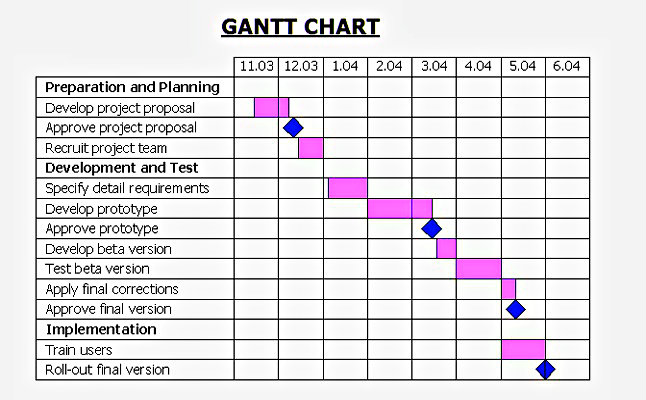
• Follow the principles of strong cohesion and loose coupling

* Factory pattern: is the only class that accesses the concrete graphical element classes. Tight coupling internally / low coupling externally

# Team Contribution

## Project Tasks and Schedule

Try to break down the overall semester project in several tasks and schedule them appropriately and in accordance with the SUPD and DEAD milestones. Gantt charts like shown in Figure 3 may help to guide and visualize this process. However, feel free to use any visualizations, methods, and tools that support you in meeting the project deadline, and report this here.

  
Figure 3: Sample Gantt Chart from <http://diagramscharts.com/gantt-chart/>. TODO: remove in final 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 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Appendix

## 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