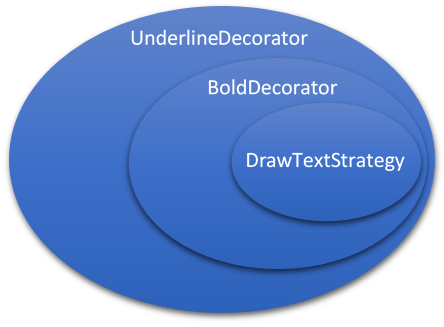
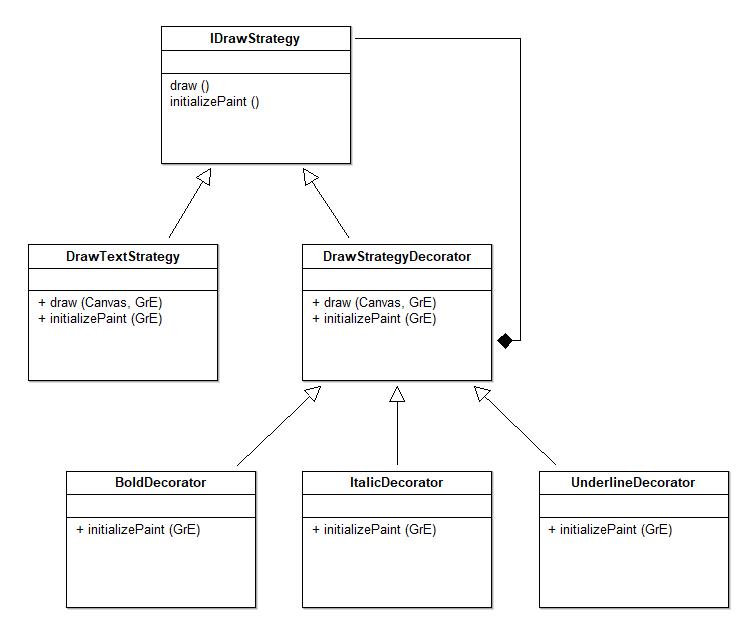
With the Decorator pattern, new functionalities can be added dynamically during runtime. Classes can be extended by "wrapping" them in any number of decorator classes. For this purpose, the class to be decorated and the decorator classes inherit from the same supertype.

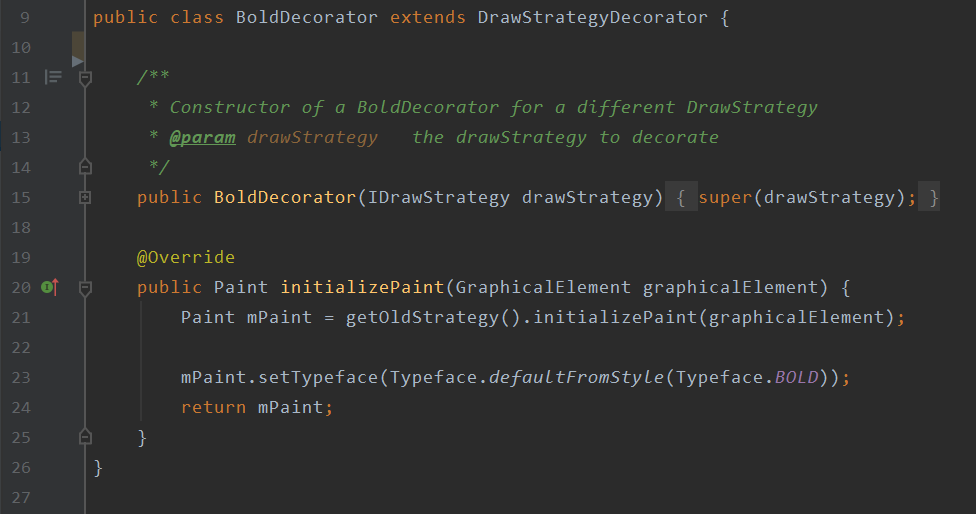
We decided to use the decorator pattern to decorate text objects with different styles. For example, a DrawTextStrategy can be "wrapped" with the BoldDecorator and then the result can be “wrapped” with the UnderlineDecorator.



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".

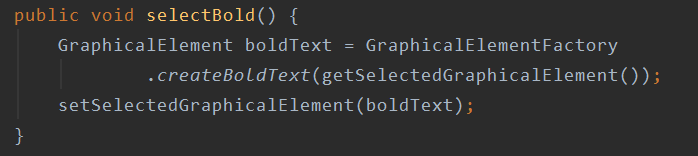


The "DrawStrategyDecorator" holds an attribute oldstrategy, which stores the drawStrategy that is currently in use. The concrete decorator classes, for example "BoldDecorator" then call the initializePaint method of the drawStrategy and add the desired functionality, here for example bold text.

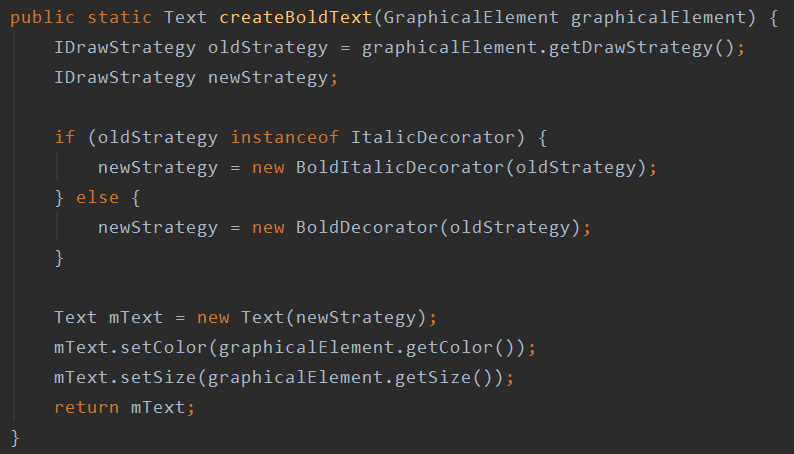


This explains the underlying logic.

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.