Analysis and Justification of design choices

Student name: Ayush Kamleshbhai Patel

Student ID: s3891013

Inclusion of MVC design pattern to build this desktop-based application.

The project is highly adhered to Model-View-Controller based design pattern as there are three major packages with "src" folder which are: -
javaFiles.model: This package includes all the java files which stores and handles persistent data. The file in this package also helps to create a communication between view and controller.

javaFiles.view: This package includes all the .fxml files which helps us to create Graphical User Interface(GUI).

javaFiles.controller: this package includes the all the controllers corresponding to the .fxml files. Controllers helps to fit in appropriate functions to view files.

Strictly Sticking to SOLID Principles.

- Single Responsibility: My code structure follows Single Responsibility as each class's name is self-explanatory and they only have those methods which are only relatable to them. For example, Users class have only those methods and attributes which are highly related to Users such as firstname, lastname, HashPassword(method) etc.

- Open for Extension, Closed for Modification: I have implemented parent-child class at many parts of code. For example, a Shapes class which has its appropriate methods, and it is extended at specific shapes. Thus, if we want to extend and add more shapes, it would be easy, and no modification will be required in Shapes class.

- Liskov Substitution: The methods into super classes have been added carefully such that Liskov Substitution principle is not violated. For example, chaneBorderColor(from Shapes super class) method is implemented into Circle, then if I change from circle to Rectangle, it wouldn't violate it as Rectangle can also change its border colour.

- Interface Segregation: The interfaces are properly segregated and highly specific according to their name. I have ensured that no interface provide dual functionality. For example, in case of Rotate interface, as its name suggests, it would only assist us to rotate elements i.e., only need to implement in elements which can rotate(Rectangle, Text but not Circle).

- Dependency Inversion: I have implemented abstractions in such a way that both high- and low-level modules depend on interfaces. For example, RectangleElement and CircleElement both are lower class, when it’s implemented by Resize, both elements’ size is changed in its own way.

Use of any other design Principles:

This time, I haven’t used any other design principle but tried to highly adhere to SOLID principles and MVC patter.