**Description of software architecture**

For our geometric drawing program we have chosen to use a Model-View-Controller Architecture. We have chosen this kind of architecture mainly because it is useful to distribute the application in terms of responsibility. In order to develop a GUI application by using a Java programming language we have chosen to adopt the JavaFX framework as it supports events, bindings and it uses FXML documents to define the structure of the GUIs. The framework itself makes use of the MVC pattern, which emphasizes our choice to use this architectural pattern. In fact, the MVC is mainly used in developing graphical user interfaces so it is the best choice for our application.

Making use of the Model-View-Controller architectural style implies the definition of a Model, a View and a Controller; in our case, the elements of the project that cover these roles are the following:

* the View is represented by “FXMLDocument”, which is a FXML file that our JavaFX-based application uses to render the GUI and send events caused by user interactions to the Controller;
* the Controller is distributed in multiple classes with different roles in terms of handling business logic, to be more specific
  + the “FXMLDocumentController” Java class acts as the first event dispatcher of the application as, according to the JavaFX framework, it directly catches events coming from interactions with the interface during runtime and executes the instructions defined in the methods with the “FXML” annotation associated with the interaction; the “FXMLDocumentController” executes very little business logic other than initializing elements of the application and triggering the execution of methods that belong to other classes, as it only handles the saving of the drawing on a file and the loading of a drawing file;
  + the “CanvasClass” Java class includes most of the business logic that concerns elements that are graphically present on the canvas, as it triggers the insertion of shapes, color changing, font size changing, zoom in and zoom out, and all the commands, while also managing the selection of shapes and labels;
  + the “Command” Java abstract class and all its implementing classes define the logic associated with the operations accessible through the context menu of both the shapes and the canvas, while also managing the font size and moving of shapes and labels, the resizing and color changing of shapes and the undo operations;
  + the “Receiver” Java class, which is strongly related to “Command” and its implementing classes, includes all the logic that triggers upon the detection of mouse events on shapes;
  + the “ColorEventManager” and “KeyboardEventManager” Java classes aid in event dispatching triggered by “FXMLDocumentController” with the help of the “ColorSubscriberInterface” and “KeyboardSubscriberInterface” interfaces respectively;
  + the classes and interfaces defined in the “shapes.creators”, “shapes.wrappers”, “tools” packages include logic associated with the creation and geometrical definition of shapes and labels;
  + the content of the “strategy” package strongly aids in the definition of the logic of shape resizing:
* the Model is distributed in very few classes, which are
  + the “FXMLDocumentController”, that keeps references of the graphical objects of the FXML file, the singleton instances of “CanvasClass”, “KeyboardEventManager” and “ColorEventManager”, and the few references needed for the business logic contained in the “FXMLDocumentController” class;
  + the “CanvasClass” Java class that includes references of all shapes on the canvas, the shapes on the canvas that are selected, the clipboard, the command history, the shape creator and all the variables needed for the execution of the business logic of “CanvasClass”;
  + the “Receiver” Java class includes most of the data intensive elements related to the “Command” class and its implementing interfaces, while also keeping track of the command history.

**Design Pattern Used**

**Creational Patterns:**

* **Factory Method**

We applied the creational Factory Method pattern for the definition of the “ShapeCreator” class and its subclasses “TextCreator”, “EllipseCreator”, “RectangleCreator” and “LineCreator” in order to make the definition of shape-instantiating classes flexible and easy to extend. We decided to adopt the variant that included the definition of a concrete method as we already defined in the parameters informations that were detailed enough to act as configuration data to aid the superclass in deciding which concrete creator call.

* **Singleton**

We made use of the creational Singleton pattern for the instances of the “CanvasClass”, “KeyboardEventManager” and “ColorEventManager” classes as the application had no need to define multiple instances of said classes, furthermore it was actually needed to keep track of just one instance of the previously named classes as they logically represent single concepts: the “CanvasClass” identifies the canvas of the application and its logic, while the event managers each handle events in an independent and unchanging way.

**Structural Patterns:**

* **Façade**

We made use of the structural Façade pattern in the definition of the “CanvasClass” as it allowed us to remove most unneeded links between objects that implement long algorithms and the “FXMLDocumentController” class while gaining a simple interface that is able to mask the complex passages between classes for the execution of the business logic.

**Behavioral Patterns:**

* **Command**

We applied the behavioural Command pattern for the definition of all classes in the “commands” package, with “Command” as the superclass of a series of classes that are called upon the execution of one of the actions of the context menu of shapes and canvas, insertion of shapes and labels, and color changing of borders and interiors of shapes. The reason behind the use of the command pattern lies in the need of turning requests of execution of certain actions into handy objects to save for history managing.

* **Observer**

We made use of the behavioural Observer pattern for the definition of the “KeyboardEventManager” and “ColorEventManager” classes and their associated interfaces “KeyboardSubscriberInterface” and “ColorSubscriberInterface” in order to make the instance of “CanvasClass” react to the events coming from the interface without relying on the “FXMLDocumentController” class for handling them.

* **State**

We made use of the behavioural State pattern for the definition of the content of the “tool” package as we needed to handle the behaviour of the instance of “CanvasClass” according to the shape selected in the shape menu and the insertion of said selected shape.

* **Strategy**

We made use of the behavioral Strategy pattern for the definition of the content of the “strategy” package as during the resizing of the selected shapes on the canvas we would have to apply slightly different algorithms according to the vertex of the selected shape from which the resizing started.