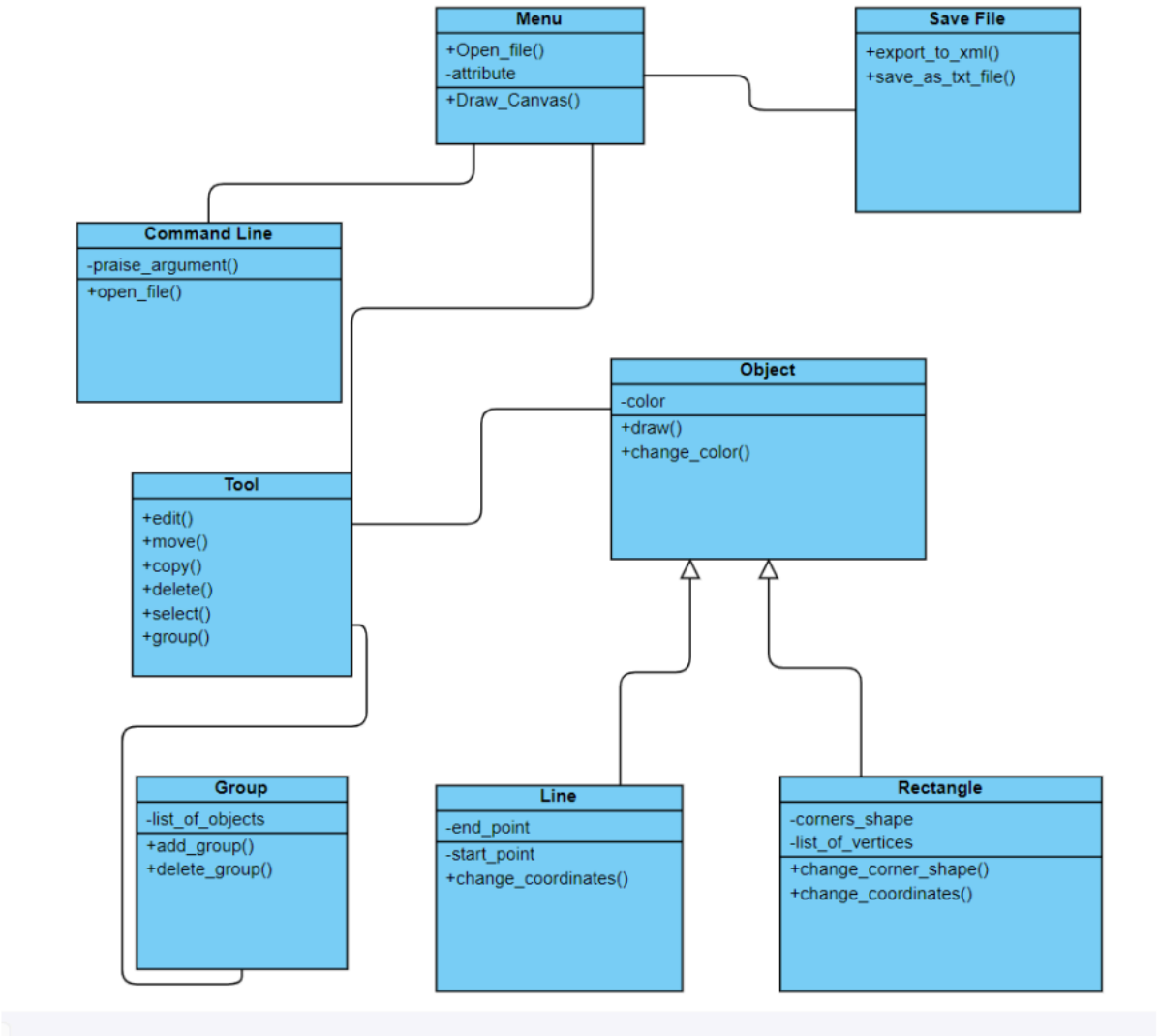
Team- Coders

Product - IIIT-Paint

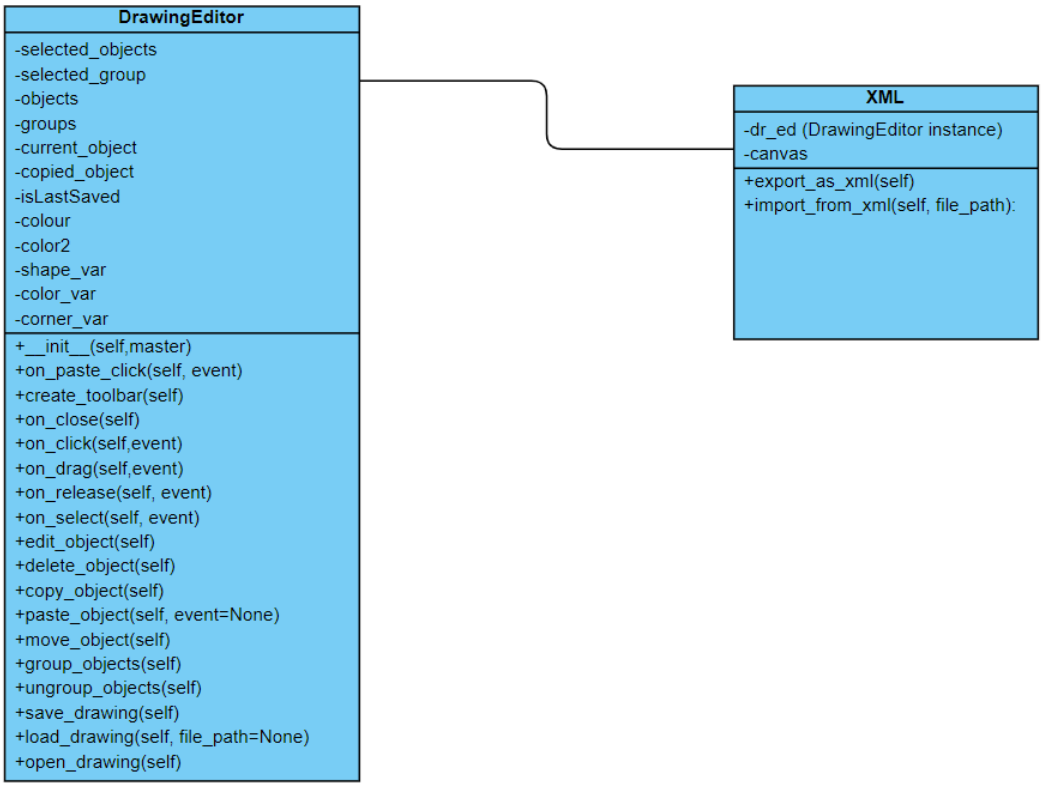
Date: 6th May 2024

Harsh Gupta (2022101067)
Kunal Angadi (2022111005)
Mohak Somani (2022101088)

UML Class Diagram (Submitted in Class)



New Class Diagram



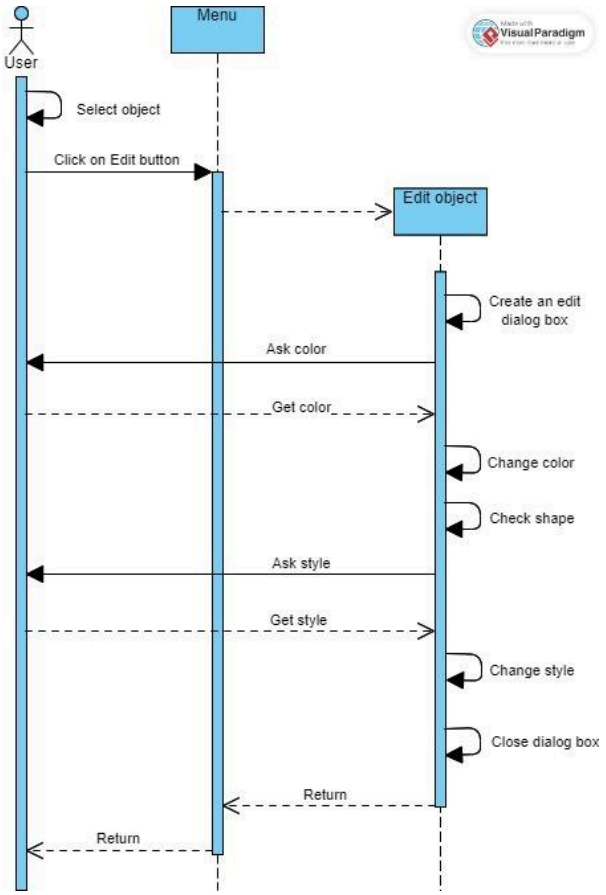
Responsibility Table for Major classes

DrawingEditor	- Initialises the main window and canvas - Creates the toolbar and handles tool selection - Manages object creation, selection, editing, deletion, copying, pasting, and moving - Handles saving, loading, and
xml.etree.ElementTree (imported)	- Provides XML parsing and creation functionality for exporting

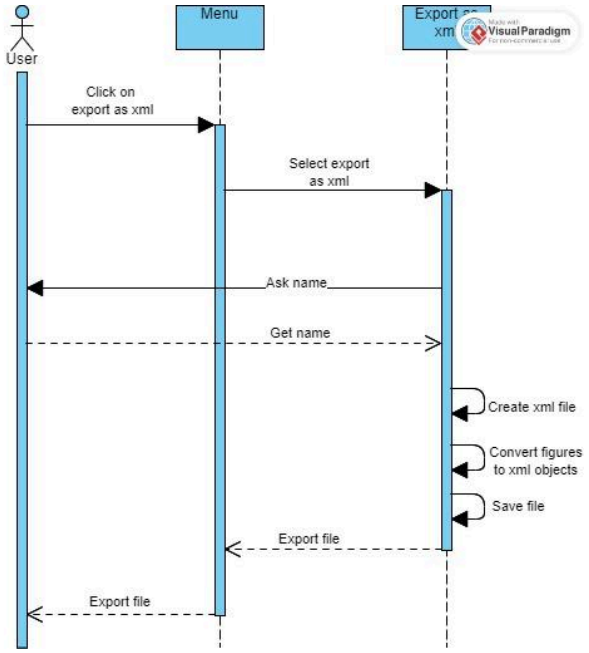
	and importing drawings
messagebox (imported)	- Provides message boxes for displaying warnings or confirmations
XML	Handles Import and Export of xml files
simpledialog (imported)	- Provides dialog boxes for user input during object editing

Sequence Diagrams

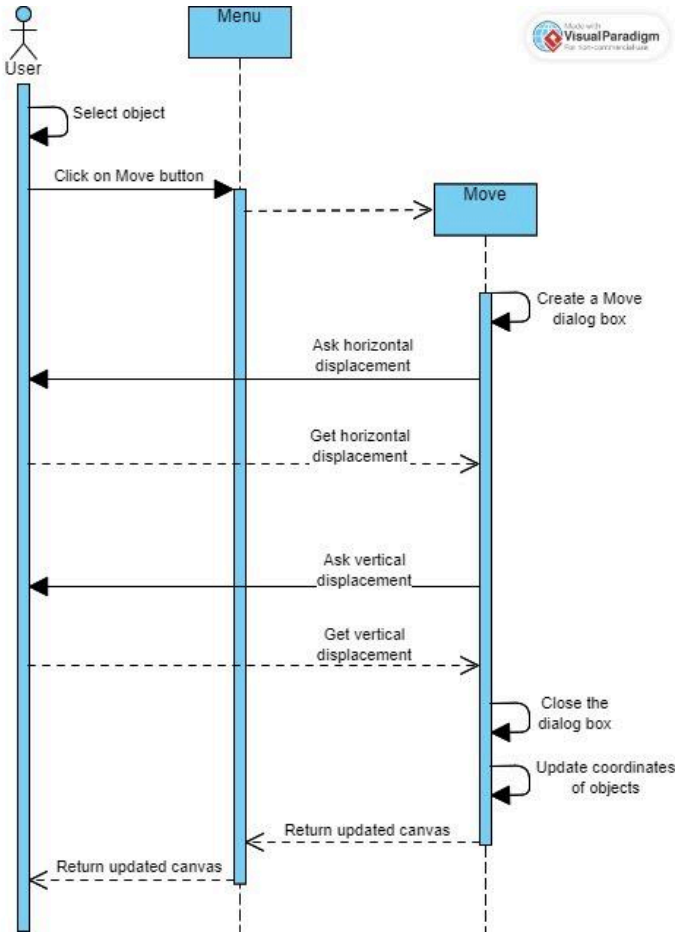
1.Edit Object



2.Export As XML



3.Move



Narrative outlining how the design reflects a balance among various software design principles and patterns:

The DrawingEditor class encapsulates the main functionality and state of the drawing editor application. It follows the Single Responsibility Principle by focusing solely on managing the drawing canvas, toolbar, and user interactions. This promotes high cohesion within the class.

The separate Tk canvas, toolbar, and event bindings help achieve a separation of concerns. The canvas handles the rendering, the toolbar provides the user interface controls, and the event bindings manage user interactions. This separation improves maintainability and extensibility.

Information hiding is applied by keeping certain attributes and methods private to the DrawingEditor class

The use of the Template Method pattern is evident in the create_toolbar() method. It defines the overall structure of the toolbar creation process while allowing subclasses to override specific steps if needed. This improves extensibility and code reuse.

The State pattern is utilised to manage the different drawing modes (line, rectangle) and colour options. The current state is stored in the shape_var and color_var attributes, which determine the behaviour of the drawing operations. This encapsulates state-specific logic and allows easy addition of new states.

The Command pattern is employed for actions like editing, deleting, copying, and pasting objects. Each action is encapsulated as a separate method, providing a clear separation of concerns and enabling extensibility by adding new command methods as needed.

The Memento pattern is used for saving and loading drawings. The save_drawing() and load_drawing() methods handle the serialisation and deserialization of the drawing state,

The Composite pattern is applied when grouping objects. Grouped objects are treated as a single entity, and operations performed on the group are propagated to its individual objects.

While the Law of Demeter is not strictly followed in all cases, the design tries to minimise unnecessary object traversals and keep interactions localised to immediate collaborators. This promotes lower coupling and improves maintainability.

The modular structure of the code, with separate methods for each functionality and clear naming conventions.

However, there are areas where the design could be improved further:

- The DrawingEditor class could be split into smaller, more focused classes to adhere to the Single Responsibility Principle more strictly. For example, separate classes could handle the toolbar, canvas, and object management.
- The use of global variables and direct access to object attributes could be reduced to improve encapsulation and minimise coupling between different parts of the code.
- The XML export and import functionality could be extracted into separate classes or modules to keep the DrawingEditor class focused on its primary responsibilities.