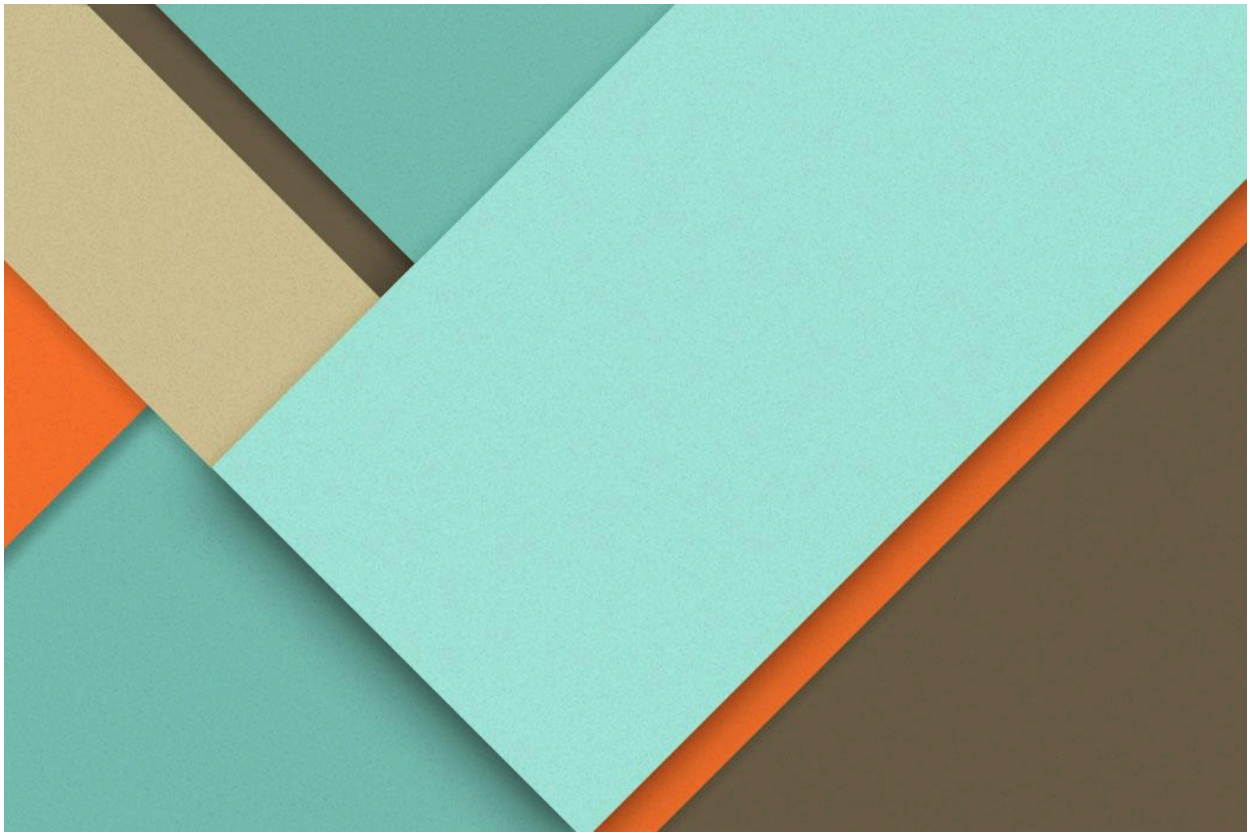


DASS Assignment 4

Drawing Editor Tool

Swarang Joshi Roll no:- 2022114010

Himanshu Singh Roll no:- 2023121013

Bassam Adnan Roll no:- 2023121003

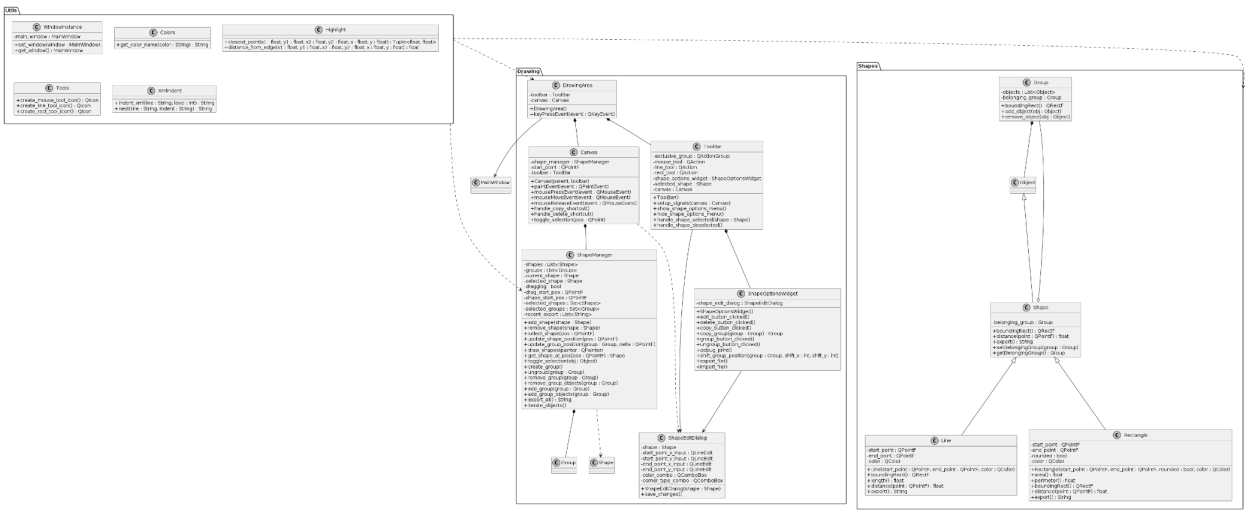
Aditya Kulkarni Roll no:- 2023121005

Rajendra Prasad Roll no:- 2023121004

Introduction

In this assignment, we have designed and implemented a Drawing Editor Tool.We have used Python for designing the tool and PlantUML for UML modeling. The goal of the assignment was to model and express the portal using classes, relations and other aspects of UML Models.

UML Class Diagram



Note-

LINK TO THE UML IMAGE HERE.

Responsibilities of Major Classes in the Drawing Application

Drawing Package:

- **DrawingArea:**
    - Manages the overall drawing application window.
    - Contains a `ToolBar` and a `Canvas`.
    - Handles keyboard events (e.g., key press).

- **ToolBar:**
    - Manages the toolbar with drawing tools (mouse, line, rectangle).
    - Groups tools using `QActionGroup` for exclusive selection.
    - Tracks the currently selected shape and canvas.
    - Shows/hides shape options menu based on selection.
    - Handles selection and deselection of shapes.

- **ShapeOptionsWidget:**
  - Provides options for editing, deleting, copying, grouping/ungrouping shapes.
  - Interacts with `ShapeEditDialog` for detailed shape editing.
  - Manages functionalities like copying groups, shifting groups, exporting/importing shapes.

- **ShapeEditDialog:**
  - Provides a dialog to edit properties (start/end points, color) of a specific shape.
  - Allows saving the changes made to the shape properties.

- **Canvas:**
  - Manages the drawing area where shapes are displayed.
  - Interacts with `ShapeManager` to draw and manage shapes.
  - Handles mouse events (press, move, release) for drawing and selection.
  - Provides functionalities like copy, delete, and selection toggling.

- **ShapeManager:**
    - Manages the collection of shapes and groups in the drawing.
    - Keeps track of the current, selected shape, and dragging state.
    - Provides methods for adding, removing, selecting, updating positions of shapes and groups.
    - Draws shapes on the canvas using the `draw_shapes` method.
    - Handles shape selection based on click position.
    - Groups and ungroups shapes as needed.
    - Provides functionalities for exporting all shapes and iterating over objects in the drawing.

Shapes Package:

- **Group:**
    - Represents a group of shapes that can be treated as a single unit.
    - Contains a list of objects (can be shapes or other groups).
    - Calculates the bounding rectangle for the entire group.
    - Provides methods for adding/removing objects from the group.

- **Shape (Abstract Class):**
    - Base class for all specific shapes (line, rectangle).
    - Provides abstract methods for calculating bounding rectangle and distance from a point.
    - Defines methods for exporting the shape data and managing belonging to a group.

- **Line:**
    - Represents a line shape with start and end points, and color.
    - Calculates its bounding rectangle, length, and distance from a point.
    - Provides a method to export line data in a specific format.

- **Rectangle:**
    - Represents a rectangle shape with start and end points, roundness attribute, and color.
    - Calculates its area, perimeter, bounding rectangle, and distance from a point.
    - Provides a method to export rectangle data in a specific format.

Utils Package (Helper Classes):

- **WindowInstance:**
    - Singleton class that manages the main application window.

- **Colors:**
    - Provides utility methods for color manipulation (e.g., getting color name from a string).

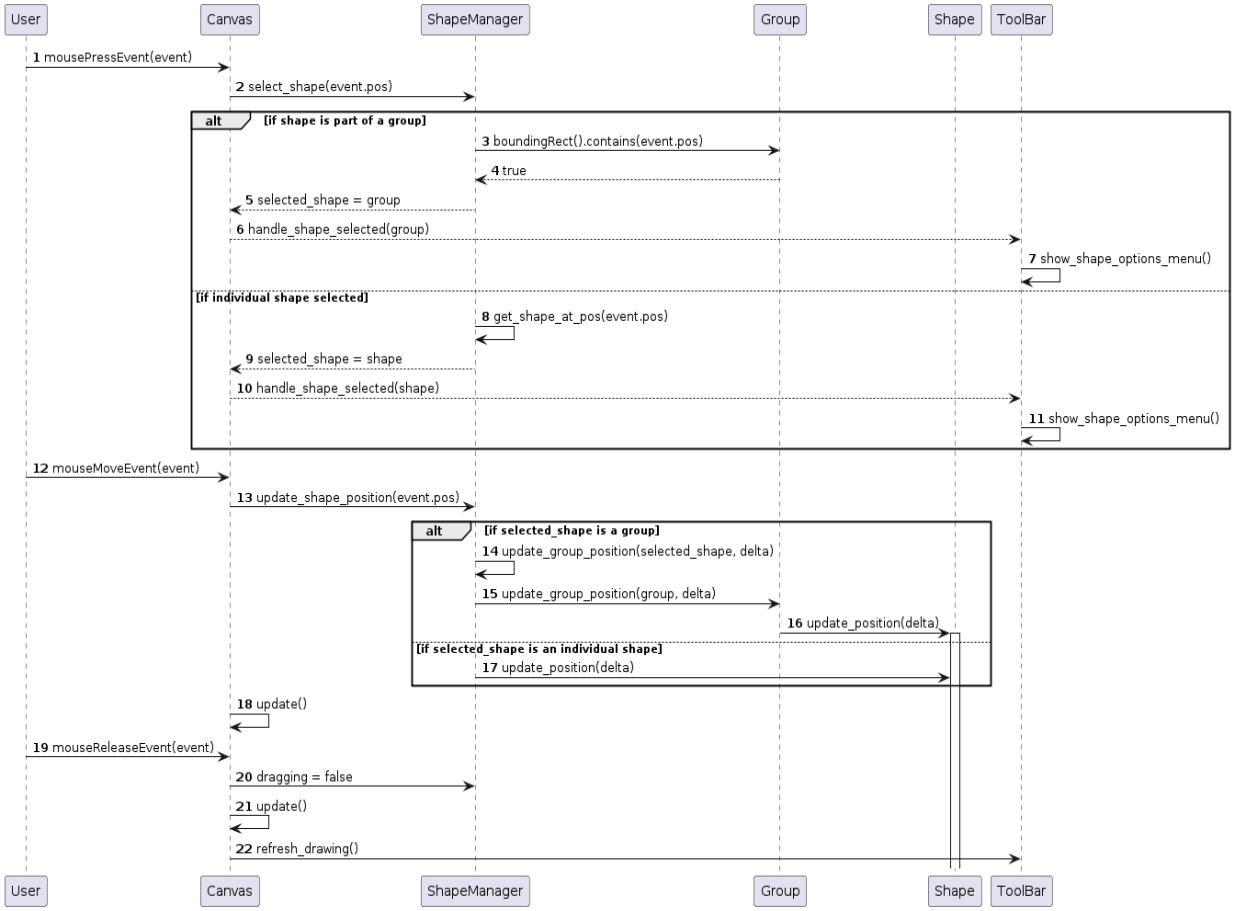
- **Highlight:**
    - Calculates the closest point on a line segment to a given point.
    - Calculates the distance from a point to the edge of a line segment.

- **Tools:**
    - Provides methods to create icons for different drawing tools (mouse, line, rectangle).

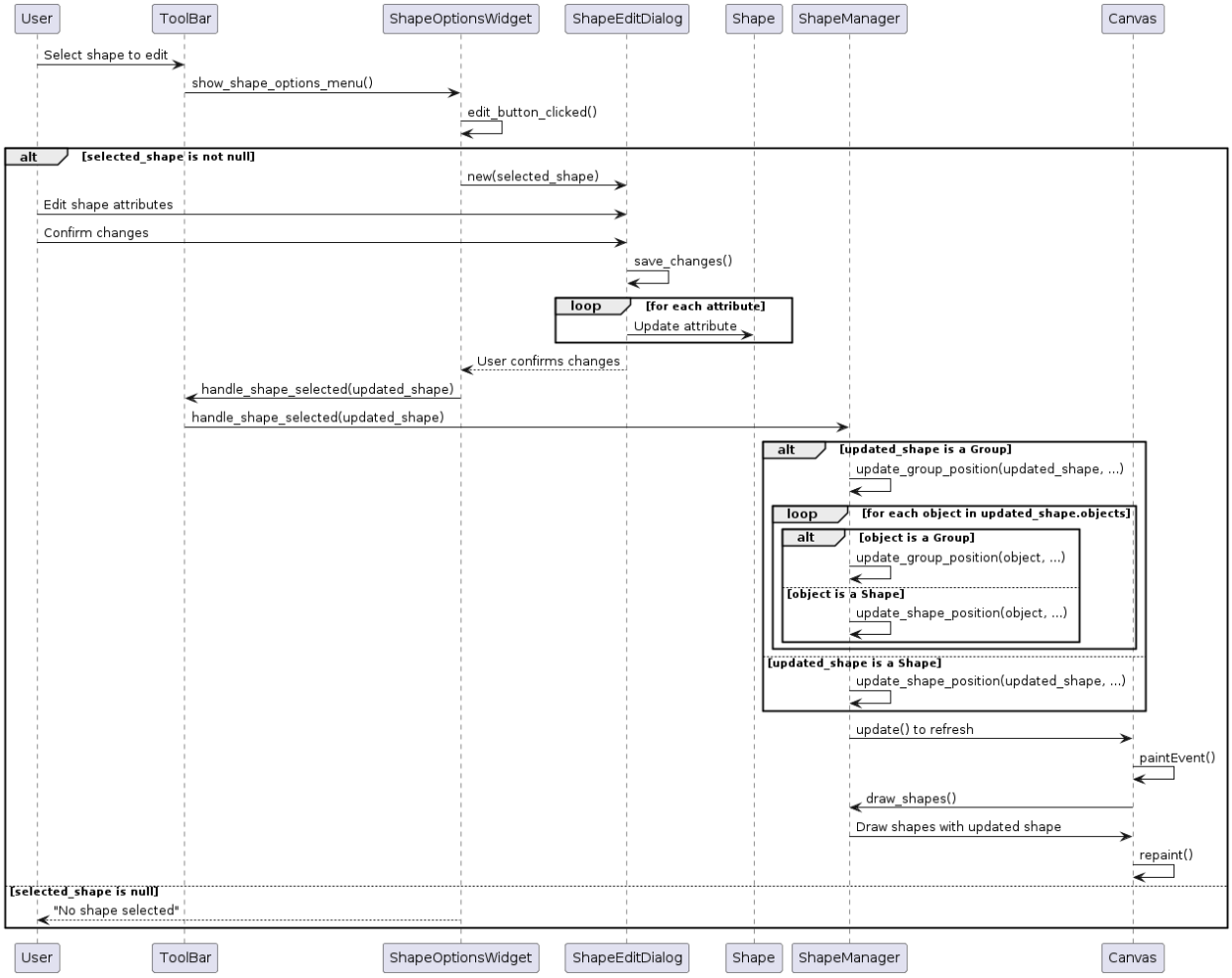
- **XmlIndent:**
    - Utility class for indenting XML code during export.

UML Sequence Diagrams -

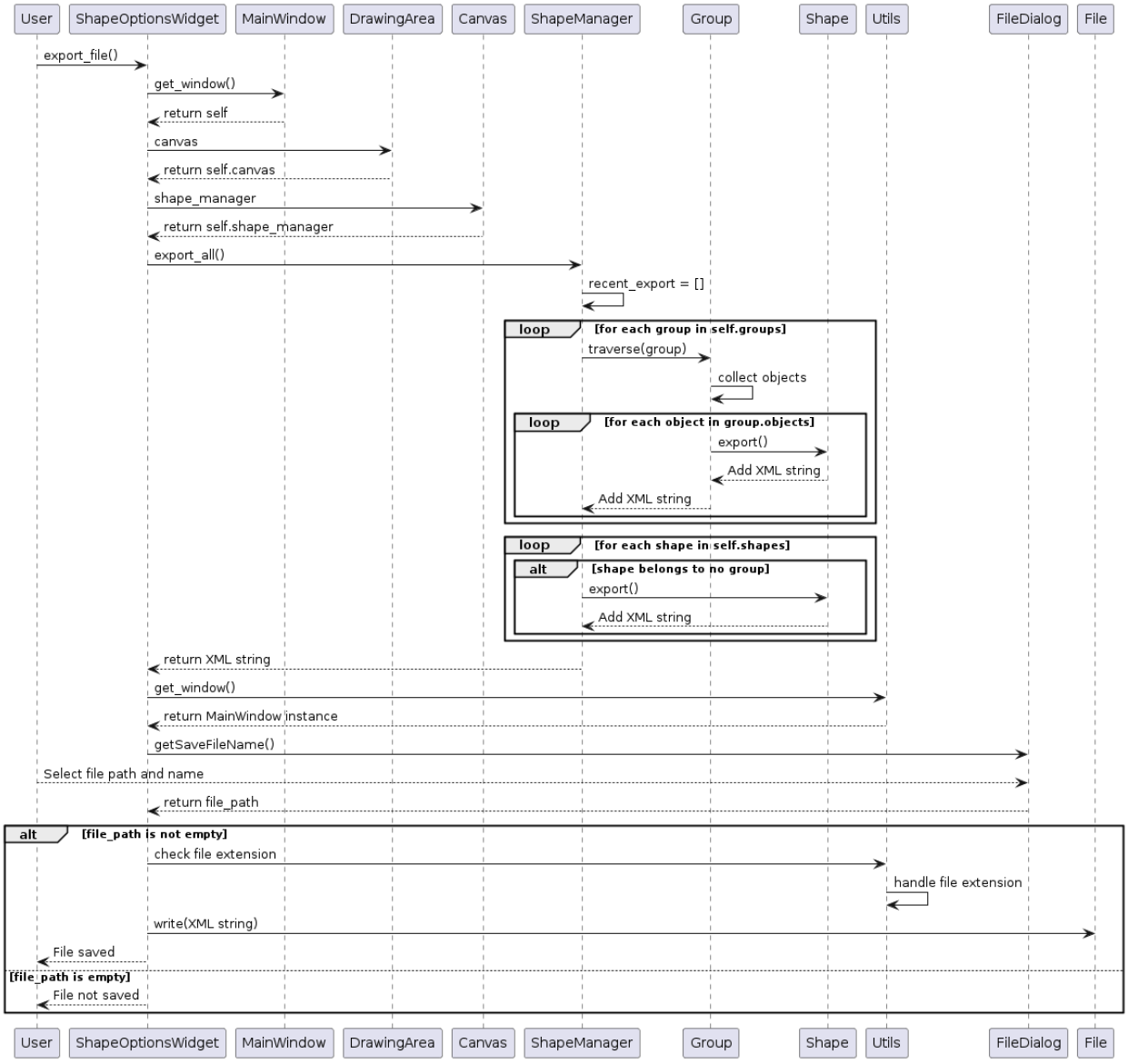
- Sequence 1- [Link]



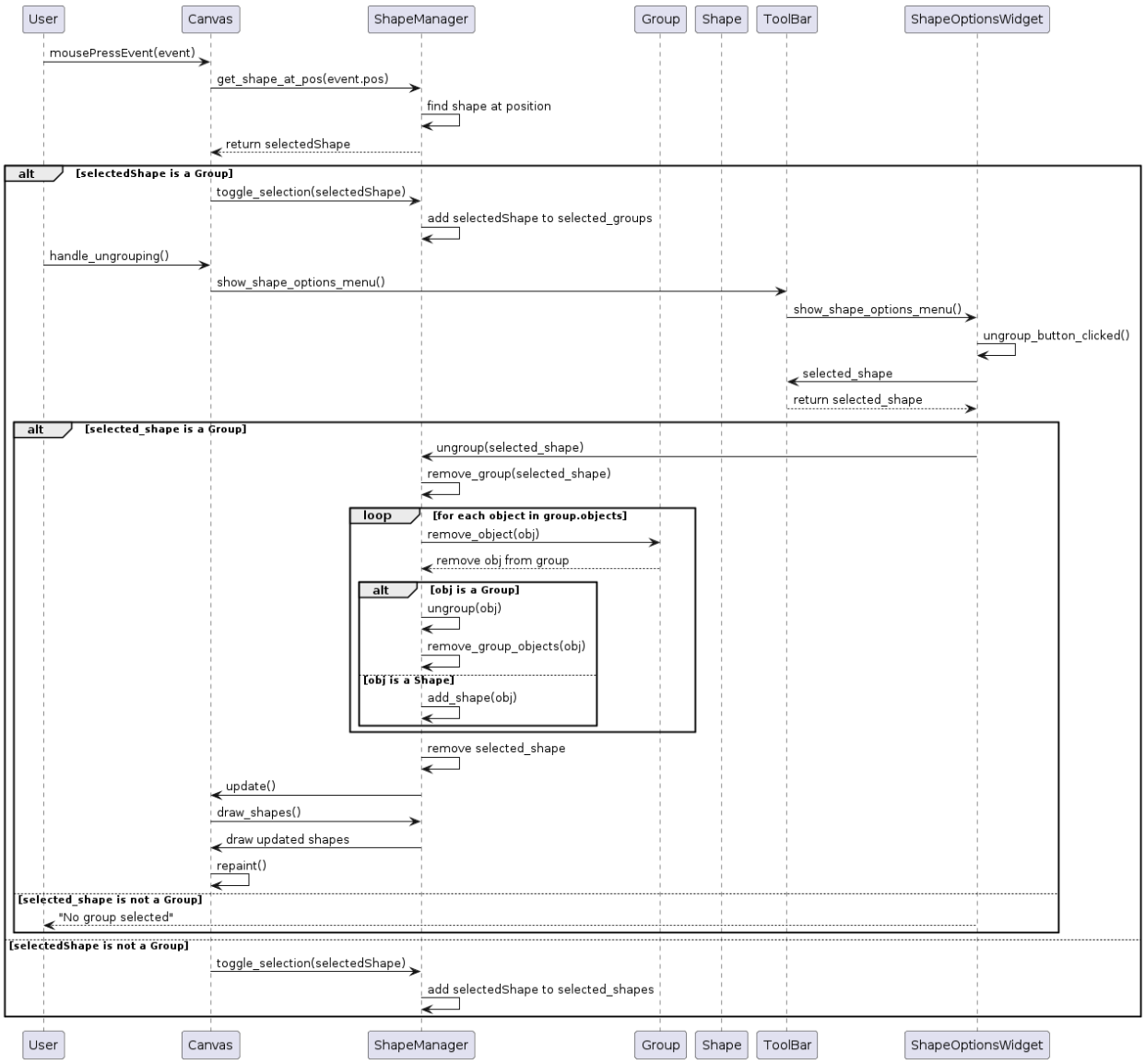
- Sequence 2- [Link]



- Sequence 3- [Link]



- Sequence 4 - Link



Designing Principles

Key Points

- **Low Coupling, High Cohesion:** The codebase is organized into modules with well-defined responsibilities (e.g., Shapes for shapes, Draw for drawing area and canvas). This minimizes dependencies and promotes focused functionality within each module.
- **Separation of Concerns:** Distinct layers handle specific aspects (e.g., ShapeManager manages shapes, Canvas handles rendering). This improves code readability and reduces the impact of changes on other areas.
- **Information Hiding:** Classes encapsulate internal details, exposing only necessary interfaces (e.g., Shape hides specific shape implementation details). This promotes modularity and simplifies modifications without affecting the entire system.
- **Law of Demeter:** Classes interact through defined interfaces, limiting knowledge of internal structures (e.g., Canvas interacts with ShapeManager for drawing, not directly accessing individual shapes). This promotes loose coupling and maintainability.

Extensibility and Reusability

- **Strategy Pattern:** Tools (mouse, line, rectangle) are implemented using QAction and QActionGroup for flexibility. New tools can be added without modifying core functionality.
- **Observer Pattern:** The Canvas class emits signals (e.g., shapeSelected) to notify observers (e.g., ToolBar) about selection changes. This allows adding or removing observers without modifying the Canvas class.
- **Composite Pattern:** The Group class and its relationship with Shape objects leverage the Composite pattern. This allows treating individual shapes and groups uniformly for operations like drawing and selection.
- **Abstraction and Polymorphism:** Abstract base classes (e.g., Shape) and interfaces (e.g., QWidget) enable polymorphic behavior. New shapes or GUI components can be added by inheriting from these, promoting code reuse.

Expected Product Evolution

The design facilitates future enhancements:

- **Adding New Shapes:** The Shape base class allows easy addition of new shapes by implementing required methods. Existing classes (ShapeManager, Canvas) work with any shape type.
- **Enhancing User Interface:** The separation of concerns allows independent evolution of user interface components (drawing area, toolbar, shape options) without affecting core functionality.
- **Introducing Persistence:** Existing export/import functionalities (ShapeManager.export_all, ShapeOptionsWidget) provide a foundation for more robust persistence mechanisms (databases, cloud storage).
- **Adding Collaboration Features:** The modular design and Qt's signals and slots make it easier to introduce collaborative features (real-time sharing, multi-user editing) through new components interacting with existing ones.
- **Enhancing Shape Editing:** The ShapeEditDialog class can be extended to support more advanced editing features (rotation, scaling, custom properties) without impacting core drawing functionality.

Conclusion

The drawing application's design balances various software design principles and patterns, promoting maintainability, code quality, and future extensibility. The chosen design patterns and modular structure position the codebase to readily adapt to future enhancements and product evolution.