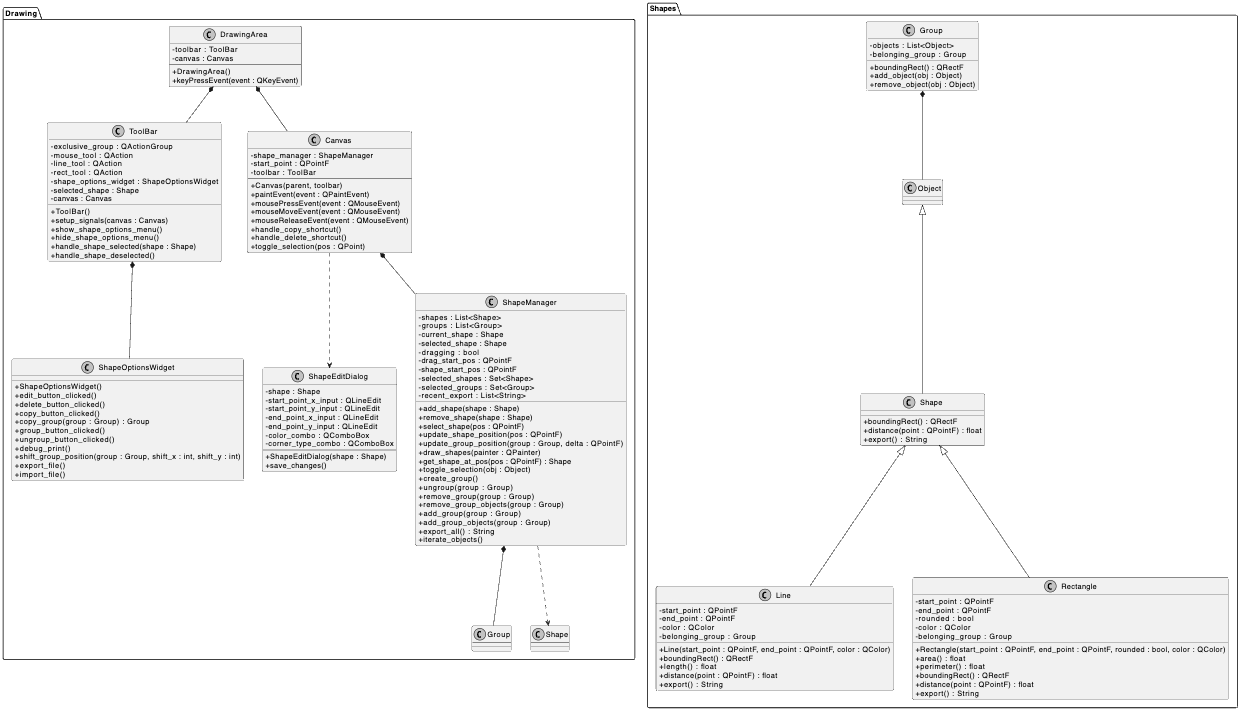
Drawing Editor Tool  
Team\_name

*Date: 05-05-2024*

Himanshu Singh 2023121013  
Bassam Adnan 2023121003

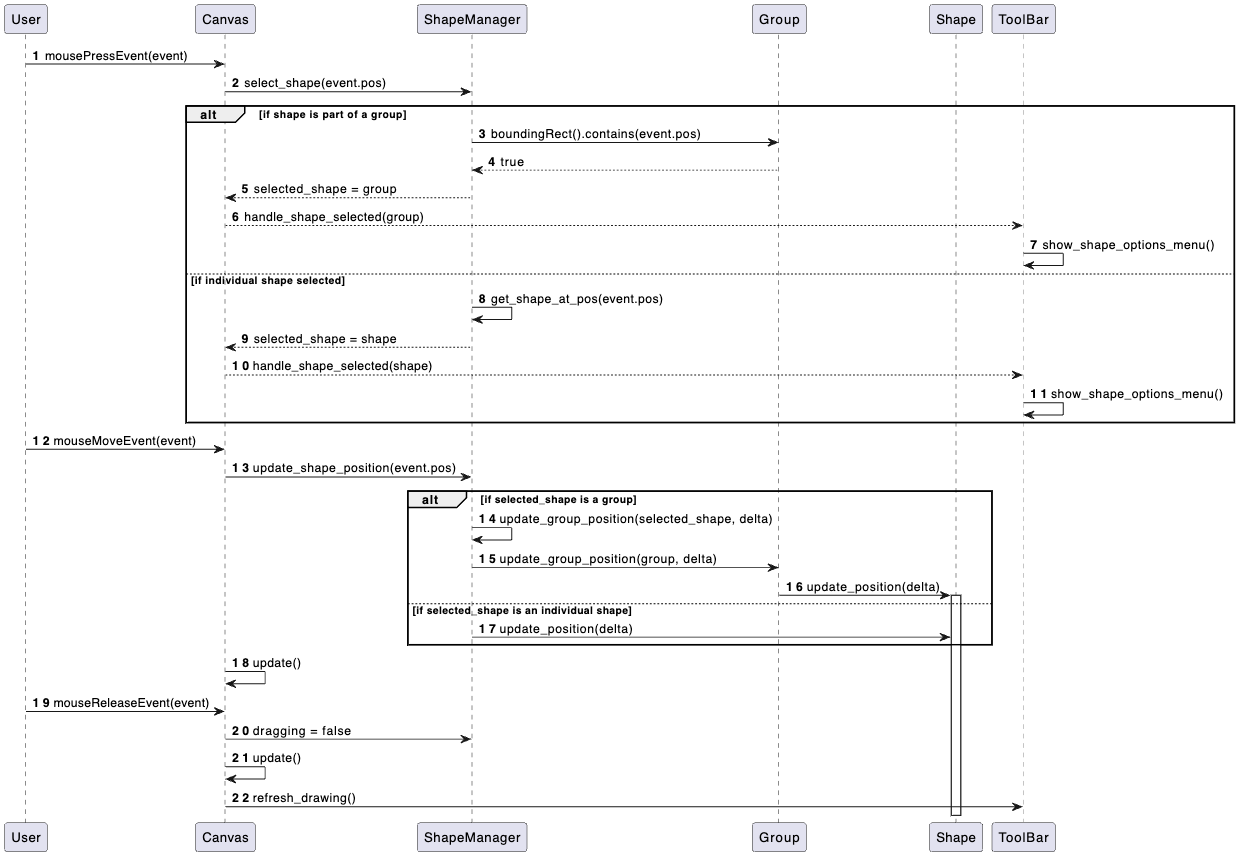
Rajendra Prasad 2023121004  
Swarang Joshi 2022114010  
Aditya Kulkarni 2023121005

**Class Diagram:**  
  


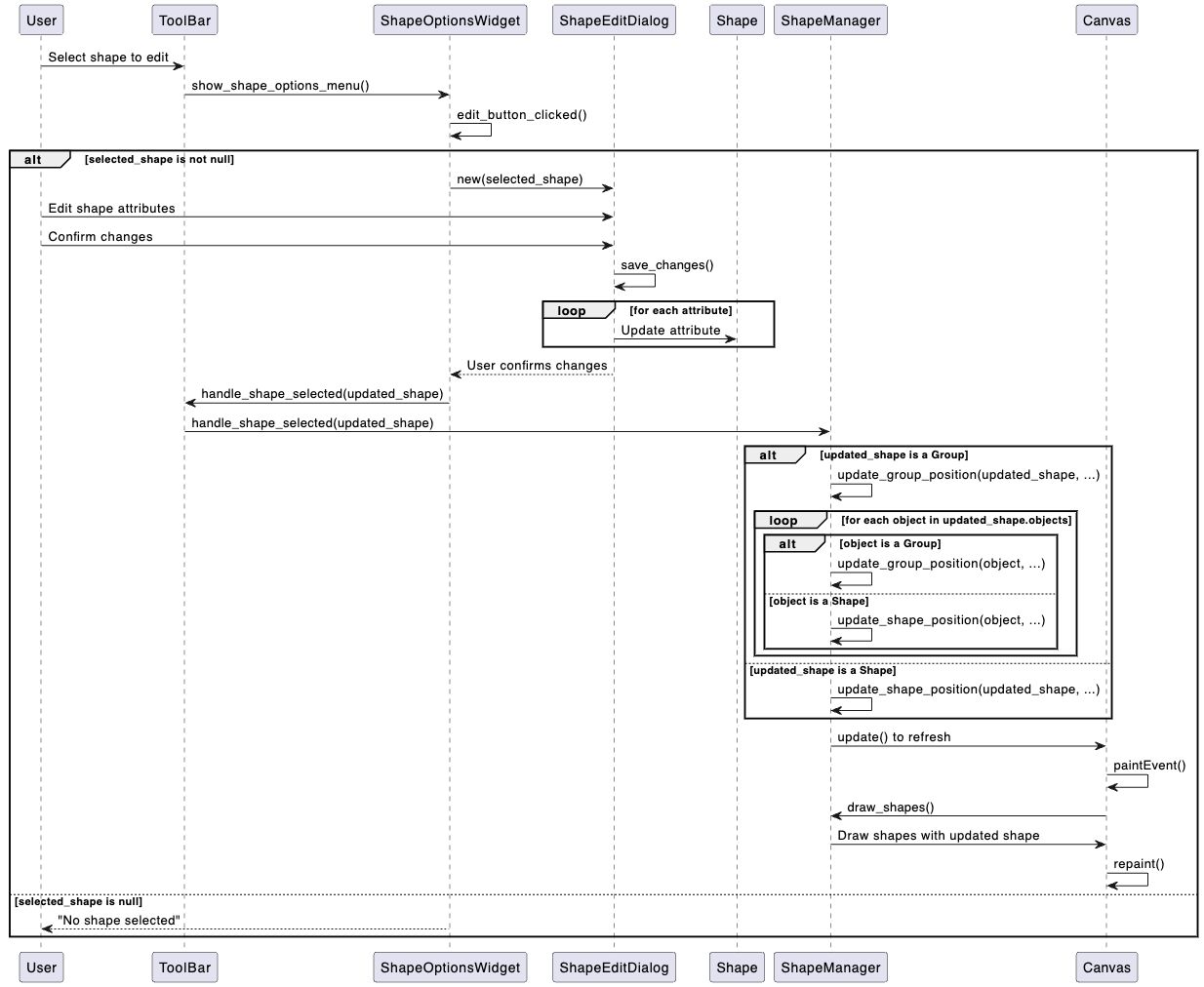
**A table summarizing the responsibilities of each major class:**

*To be done*

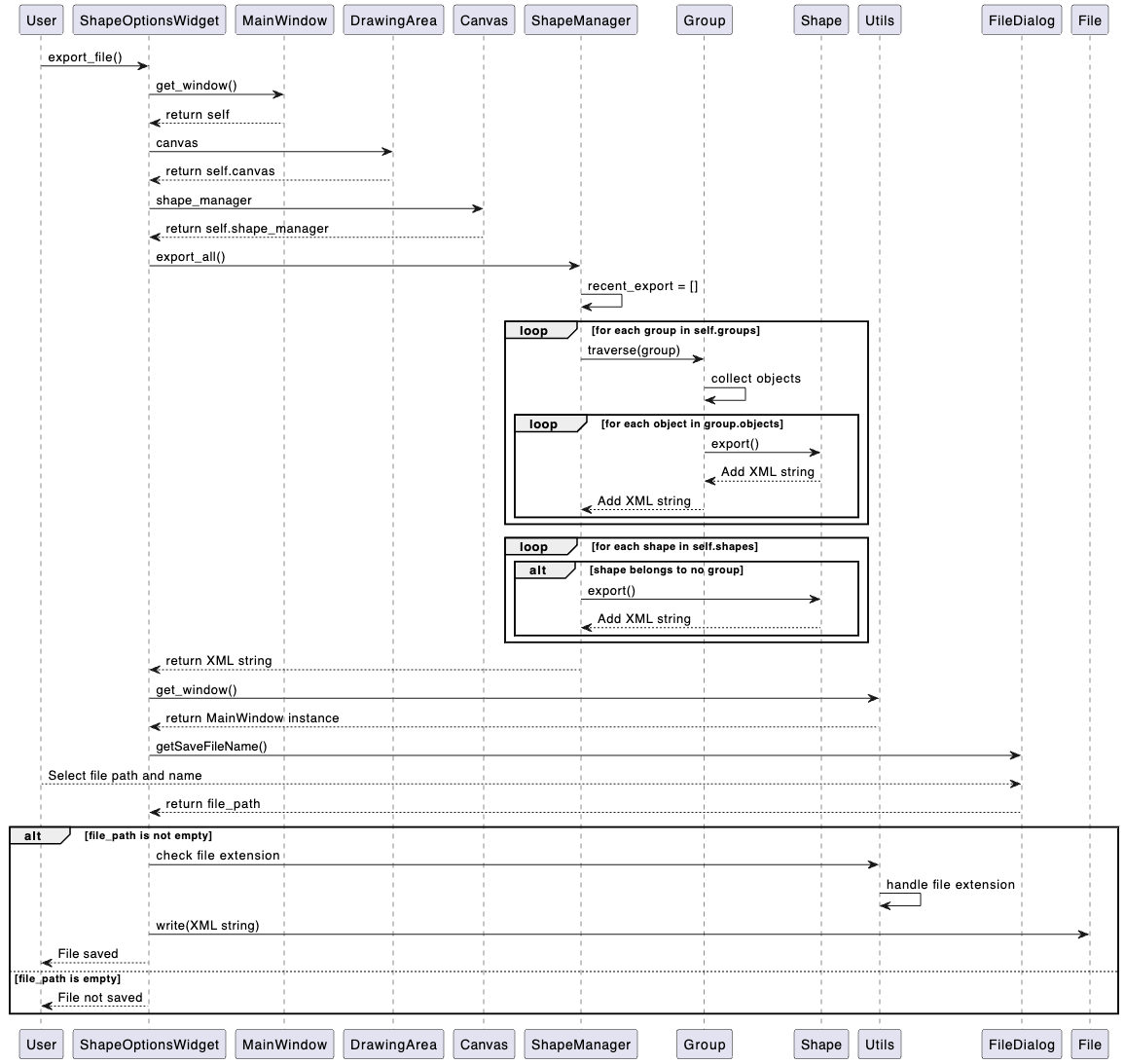
**Sequence Digrams:**  
  
**Seq1.** Mouse selection of an element that is in a group, drag to a new position, update of the model, refresh of the drawing.



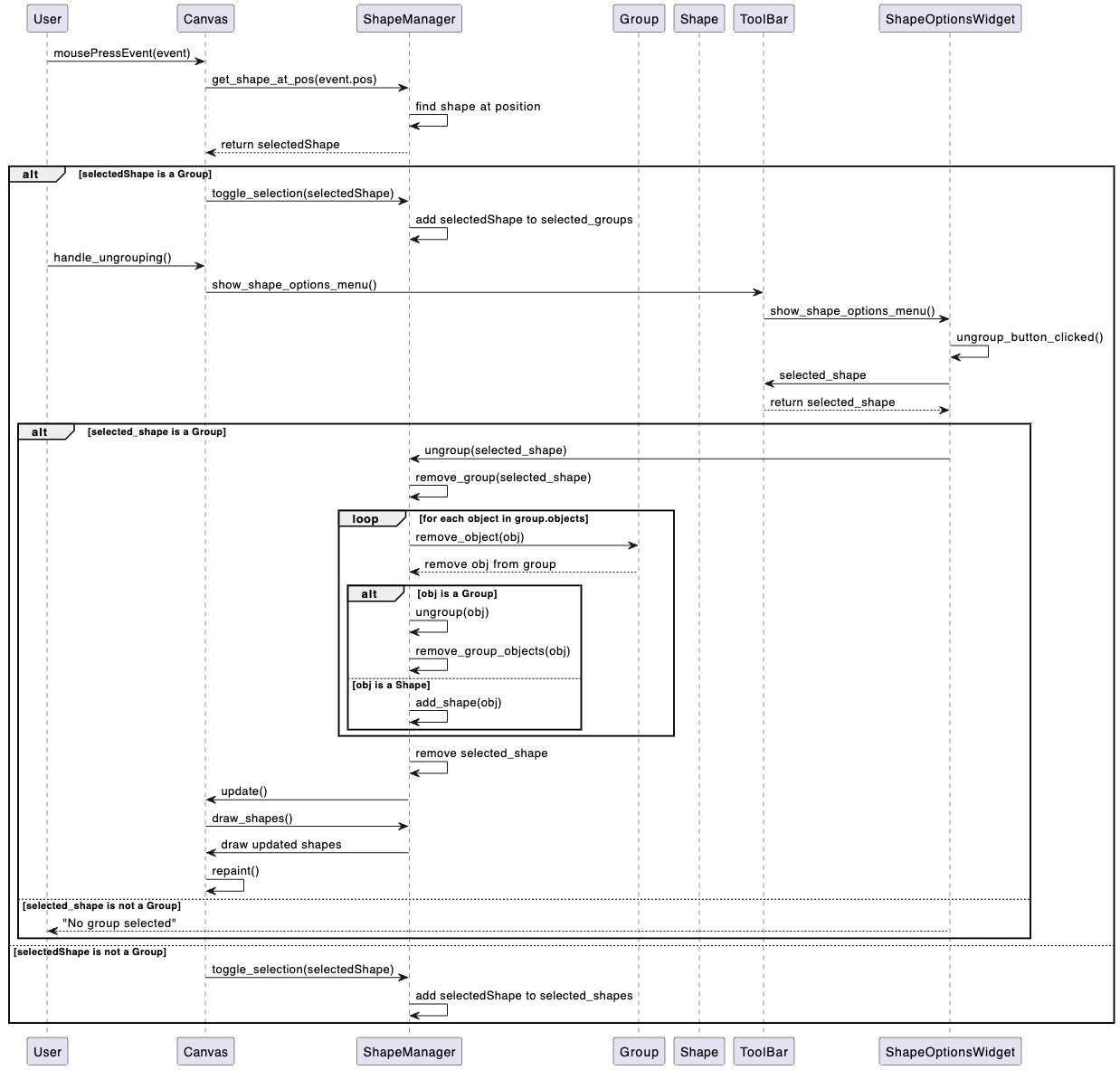
**Seq2.** Menu choice to edit the current object, user change of attributes, user confirmation of change, update of element in the model, refresh of the drawing.



**Seq3.** User indicates to save an XML file, contents of drawing being written to the file, file closed.



**Seq4.** Mouse selection of an element that is in a group, ungrouping of the elements in the group.



The design of the drawing application reflects a thoughtful balance among several key software design principles and patterns, aimed at promoting code quality, maintainability, and extensibility. Here's a narrative outlining how the design achieves this balance:

\*\*Low Coupling and High Cohesion\*\*:

The codebase is organized into several logical modules or components, each with a well-defined responsibility. For example, the `Shapes` package encapsulates the different types of shapes (Line, Rectangle, Group), while the `draw` package handles the drawing area, toolbar, and canvas. This modular design promotes low coupling between components, as they interact through well-defined interfaces or APIs. Within each component, the classes exhibit high cohesion, as they logically group related functionality together.

\*\*Separation of Concerns\*\*:

The design follows the principle of separating concerns by dividing the application into distinct layers or components, each responsible for a specific aspect of the system. For instance, the `ShapeManager` class manages the shapes and groups, while the `Canvas` class handles the rendering and user interactions. This separation of concerns makes the code easier to understand, maintain, and extend, as changes to one concern are less likely to affect other parts of the system.

\*\*Information Hiding\*\*:

The design employs information hiding by encapsulating the internal implementation details of classes and exposing only the necessary interfaces or APIs. For example, the `Shape` class is an abstract base class, hiding the specific implementation details of different shapes (Line, Rectangle) from the rest of the system. This promotes modularity and allows for seamless extension or modification of the shape types without affecting other parts of the application.

\*\*The Law of Demeter\*\*:

The design follows the Law of Demeter (or the Principle of Least Knowledge) to a reasonable extent. Classes interact with other classes through well-defined interfaces or APIs, limiting their knowledge of the internal structure of those classes. This can be seen in the way the `Canvas` interacts with the `ShapeManager` to draw shapes or handle user interactions, without directly accessing the internal details of the `ShapeManager` or individual shapes.

\*\*Extensibility and Reusability\*\*:

The design incorporates several patterns and principles that promote extensibility and reusability:

1. \*\*Strategy Pattern\*\*: The use of the `QAction` and `QActionGroup` classes from Qt allows for a flexible and extensible way of handling different drawing tools (e.g., mouse, line, rectangle). New tools can be easily added or removed without modifying the core functionality.

2. \*\*Observer Pattern\*\*: The `Canvas` class emits signals (`shapeSelected`, `shapeDeselected`) to notify interested parties (e.g., `ToolBar`) about changes in the selection state. This loosely coupled communication mechanism allows for easy addition or removal of observers without modifying the `Canvas` class.

3. \*\*Composite Pattern\*\*: The `Group` class and its relationship with `Shape` objects (Line, Rectangle) implement the Composite pattern. This pattern allows for treating individual shapes and groups of shapes uniformly, simplifying operations such as drawing, selecting, and transforming shapes or groups.

4. \*\*Abstraction and Polymorphism\*\*: The use of abstract base classes like `Shape` and interfaces like `QWidget` allows for polymorphic behavior, promoting code reuse and extensibility. New shapes or GUI components can be added by inheriting from these base classes or interfaces, without modifying the existing code.

\*\*Expected Product Evolution\*\*:

The design of the drawing application is well-suited for anticipated product evolution and future enhancements. Some potential areas of evolution and how the design can accommodate them are:

1. \*\*Adding New Shape Types\*\*: The existing design allows for easy addition of new shape types by inheriting from the `Shape` base class and implementing the required methods (e.g., `boundingRect`, `distance`, `export`). The `ShapeManager` and `Canvas` classes are designed to work with any shape type, promoting extensibility.

2. \*\*Enhancing the User Interface\*\*: The separation of concerns between the drawing area (`DrawingArea`), toolbar (`ToolBar`), and shape options (`ShapeOptionsWidget`) facilitates independent evolution of the user interface components without affecting the core drawing functionality.

3. \*\*Introducing Persistence\*\*: The design already includes provisions for exporting and importing shapes as XML files (`ShapeManager.export\_all`, `ShapeOptionsWidget.export\_file`, `ShapeOptionsWidget.import\_file`). This foundation can be extended to support more robust persistence mechanisms, such as database integration or cloud storage.

4. \*\*Adding Collaboration Features\*\*: The modular design and the use of signals and slots in Qt make it relatively straightforward to introduce collaborative features, like real-time sharing or multi-user editing of drawings, by creating new components or modules that communicate with the existing ones through well-defined interfaces.

5. \*\*Enhancing Shape Editing\*\*: The `ShapeEditDialog` class can be extended to support more advanced editing features, such as rotation, scaling, or custom shape properties, without disrupting the core drawing functionality.

Overall, the design of the drawing application strikes a balance between various software design principles and patterns, promoting code quality, maintainability, and extensibility. The modular structure, separation of concerns, information hiding, and the judicious use of design patterns like Strategy, Observer, and Composite, position the codebase to accommodate future enhancements and product evolution with relative ease.