Design Documentation

Team Name - Loner Divyansh

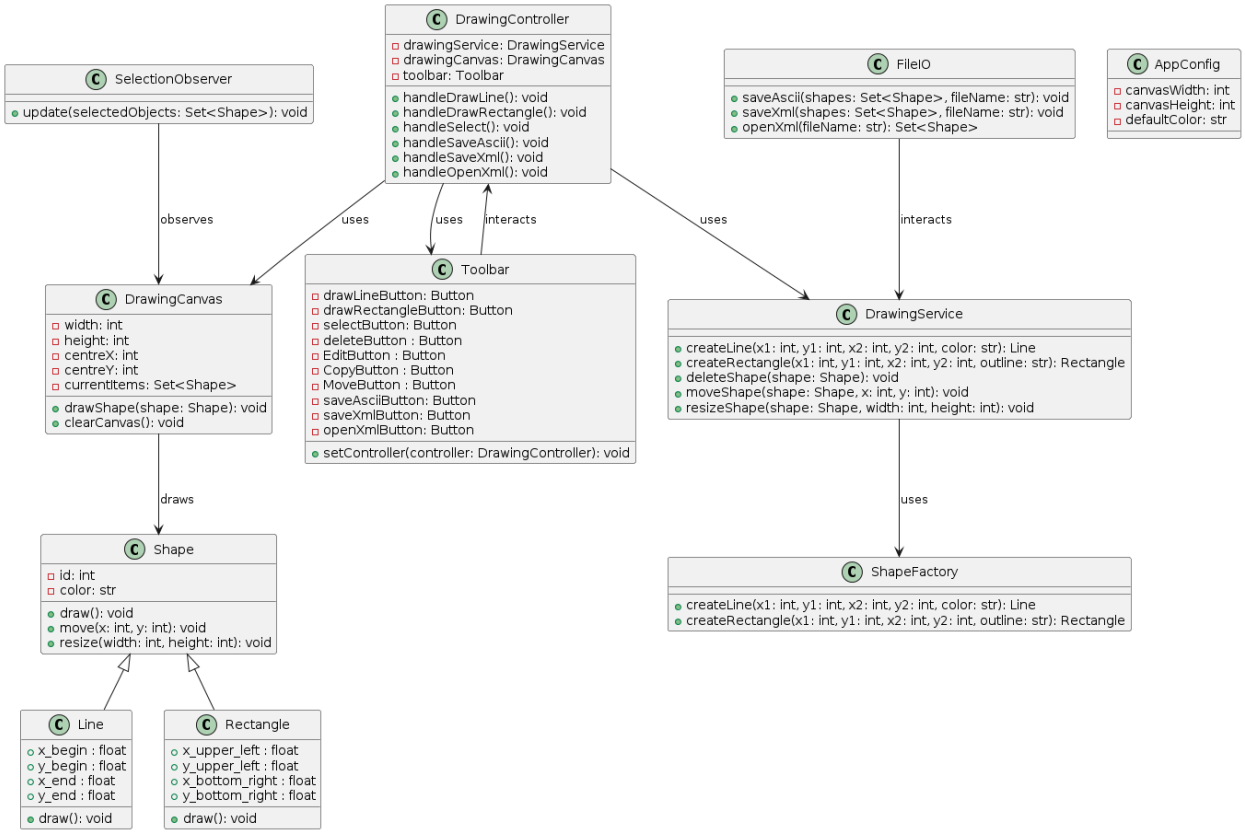
Team members - Divyansh Pandey (2022101111)

Product Name - Shape-ally

Introduction:

This Python-based drawing editor tool enables users to generate fundamental geometric shapes, manipulate them, and save them for later use. The design emphasizes simplicity, extensibility, and adaptability, making it easy to incorporate additional features according to user preferences. We have an editing window of size(800×800) where you can create lines and rectangles, edit them, move them and copy them.

UML diagram:



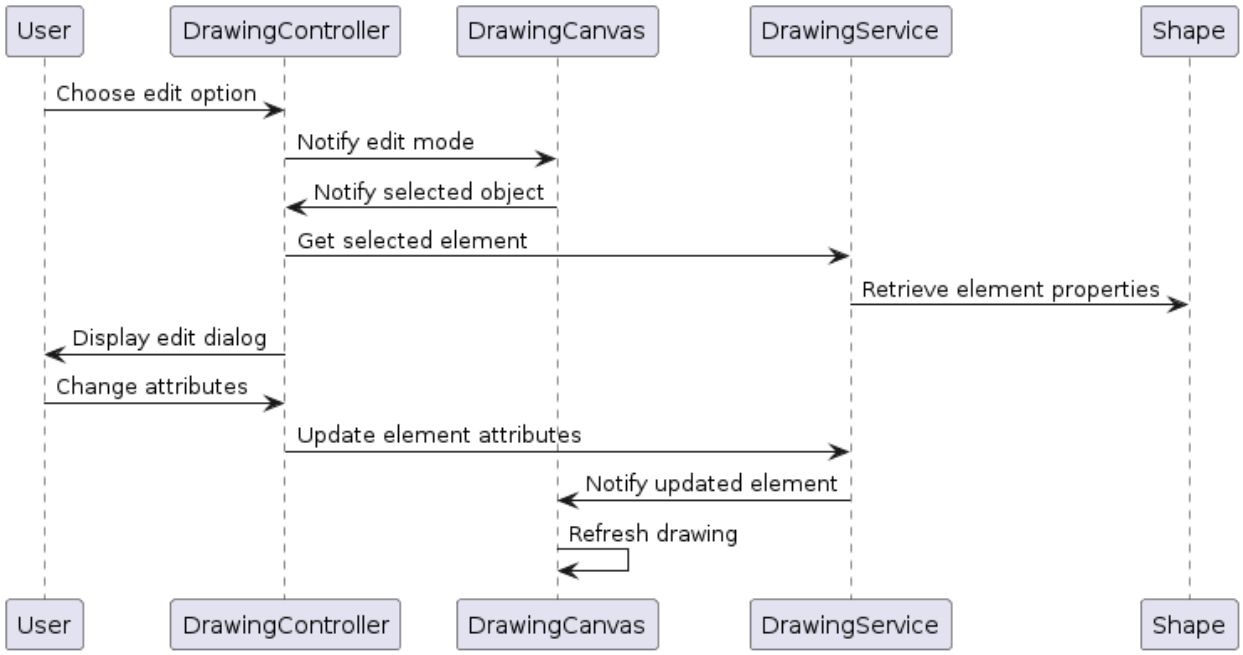
Class features:

Class	Responsibilities
`Shape`	Base class for all shapes. Defines the common properties and methods.
`Rectangle`	Represents a rectangle. Inherits from `Shape` and implements specific behavior.
`Line`	Represents a line. Inherits from `Shape` and implements specific behavior.
`Toolbar`	Contains all the utility buttons to manipulate and edit the drawing canvas.
`ShapeFactory`	Factory class to create instances of shapes (Factory pattern).
`DrawingCanvas`	Represents the canvas used for drawing objects containing details and methods of

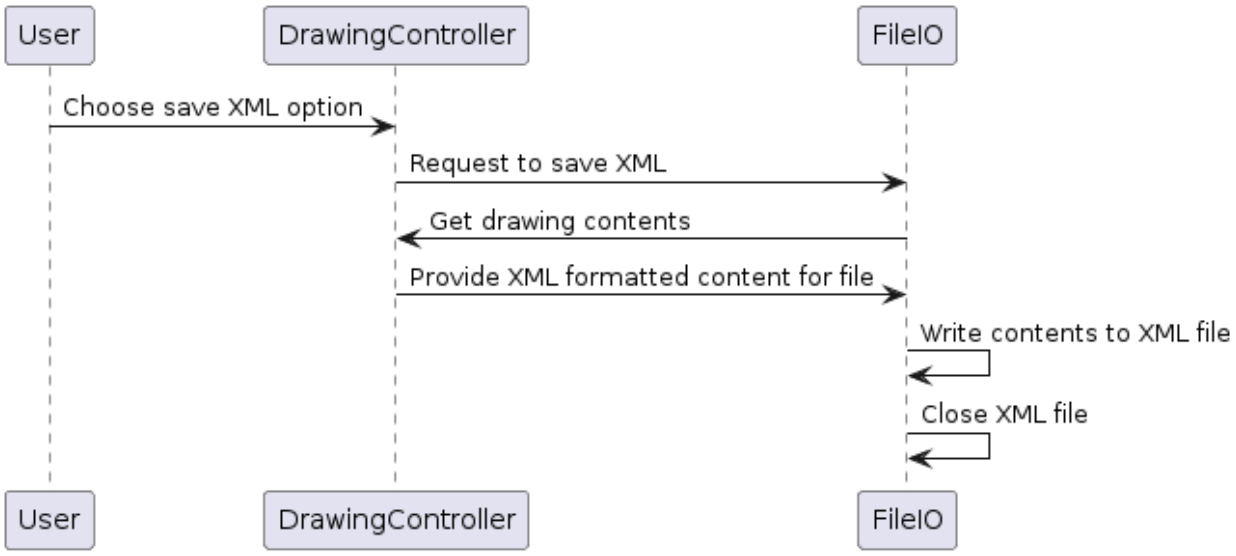
	the canvas.
`FileIO`	Represents the file management system of the editor with Import and Export features.

Sequence diagrams:

1. 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.



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



Design Features:

1. **Low Coupling**: The system demonstrates low coupling between its components by utilizing the MVC (Model-View-Controller) architectural pattern. The `DrawingController` serves as the controller, facilitating communication between the user interface (`DrawingCanvas`, `Toolbar`) and the backend logic (`DrawingService`, `ShapeFactory`, `FileIO`). Each component is responsible for a specific set of tasks, minimizing interdependence and allowing for easier modification or replacement of individual components without affecting others.
2. **High Cohesion**: The components within the system exhibit high cohesion by encapsulating related functionality within each class. For example, the `DrawingCanvas` class is responsible for drawing shapes on the canvas and managing user interactions related to drawing, while the `DrawingService` class handles operations related to manipulating shapes and managing the drawing state. This organization ensures that each class has a clear and focused purpose, enhancing readability and maintainability.
3. **Separation of Concerns**: The design separates different concerns into distinct components, adhering to the principle of separation of concerns. For instance,

user interface-related functionality is handled by the `DrawingCanvas` and `Toolbar` classes, while backend logic such as shape manipulation and file I/O operations is handled by the `DrawingService` and `FileIO` classes, respectively. This separation allows for easier understanding of each component's responsibilities and facilitates changes in one area of the system without impacting others.

4. **Information Hiding**: The design employs information hiding to encapsulate internal details and provide a clear interface for interaction with each component. For example, the `DrawingService` class hides the internal representation of shapes and exposes methods such as `createLine`, `createRectangle`, `deleteShape`, etc., to interact with shapes without exposing their internal structure. This abstraction allows for changes to the underlying implementation without affecting other parts of the system.
5. **Law of Demeter**: The design follows the Law of Demeter by promoting loose coupling between objects and limiting direct interactions between distant components. For example, the `DrawingController` interacts with the `DrawingService` to perform shape manipulation operations, rather than directly accessing shape objects or canvas properties. This reduces the complexity of dependencies and makes the system more maintainable and extensible.
6. **Extensibility and Reusability**: The design incorporates patterns such as the Factory Method pattern (`ShapeFactory`) to facilitate the creation of new shape objects with varying parameters. This promotes extensibility by allowing for the easy addition of new shape types without modifying existing code. Additionally, the MVC architecture and modular design make components easily reusable in other contexts or projects, contributing to the overall flexibility of the system.
7. **Expected Product Evolution**: The design anticipates future product evolution by providing clear separation between user interface components and backend logic. As the product evolves, new features can be added, or existing ones modified with minimal impact on other parts of the system. For example, if new drawing tools or file formats need to be supported, new classes can be added or existing ones extended without disrupting the existing functionality.