Paint
Programming II - Assignment 3

Team Members

Mohamed Tarek Hussien Hassan Ibarhim	21011161
Asser Ossama ElZeki	21010241
Ahmed Ayman Ahmed Abdallah	21010048
Ebrahim Alaa Eldin Ebrahim	21010017

Contents

1 Installation and Running Steps on Ubuntu and Windows

1.1 Install JDK 17

Ubuntu:

1. Open a terminal.
2. Update the package list: `sudo apt update`.
3. Install OpenJDK 17: `sudo apt install openjdk-17-jdk`.
4. Verify the installation: `java -version`.

Windows:

1. Download the OpenJDK 17 installer from here.
2. Run the installer and follow the on-screen instructions.
3. Verify the installation: `java -version`.

1.2 Install Maven

Ubuntu:

- Installing maven using the apt command is not recommended as it will install an older version incompatible with JDK 17.
- Instead, download the latest Maven binary from here.
- Extract the downloaded archive: `tar -xf apache-maven-VERSION-bin.tar.gz`.
- Move the extracted directory to a suitable location, such as `/opt/maven`: `sudo mv apache-maven-VERSION /opt/maven`.
- Add the Maven bin directory to your `PATH` environment variable:

```
echo 'export PATH=/opt/maven/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
```

- Verify the installation: `mvn -v`.

Windows:

- Download the latest Maven binary from here.
- Run the installer and follow the on-screen instructions.
- Verify the installation: `mvn -v`.

1.3 Server-Side

Both Ubuntu and Windows:

1. Open a terminal and navigate to the `server-side/paint` directory within your project folder.
2. Run `mvn clean install` to build the server-side application.
3. Run `mvn spring-boot:run` to start the server.
4. The server will be running at `http://localhost:8000`.

1.4 Client-Side

Both Ubuntu and Windows:

1. Open a new terminal and navigate to the `client-side/paint` directory within your project folder.
2. Run `npm install` to install the required dependencies.
3. Run `npm run dev` to start the client-side application.
4. The client will be running at `http://localhost:5174/` or the IP address displayed in the terminal.

1.5 Verification

- The server-side application should be accessible at `http://localhost:8000`.
- The client-side application should be accessible at `http://localhost:5174/` or the IP address displayed in the terminal.
- If both applications are running successfully, the client-side application should be able to communicate with the server-side application at `http://localhost:8000`.

2 Design Patterns Used

2.1 Factory Creational pattern

The `ShapeManager.ShapesFactory` class implements the factory pattern, and here's how it follows the pattern:

- **Encapsulates object creation:** The `getShape` method takes a string argument representing the type of shape desired and returns a new instance of that shape. This hides the concrete implementation details of creating different types of shapes from the client code.
- **Provides a centralized location for object creation:** The factory is the single source of truth for creating new shapes. This makes it easier to modify or extend the creation process without affecting the client code.
- **Decouples classes from concrete implementations:** By using the factory, the client code doesn't need to know the specific class names of the shapes it wants to create. This makes the code more flexible and easier to maintain.

2.2 Prototype Creational Pattern

- **Declares an interface for creating clones:** The `Shape` class has a public `clone` method that returns a new instance of the same type of shape. This allows clients to create new shapes by cloning existing ones.
- **Defines a concrete implementation for cloning itself:** The implementation of the `clone` method in the `Shape` class creates a shallow copy of the object. This means that it copies the primitive values and references to other objects, but it doesn't copy the objects themselves.
- **Allows subclasses to override the cloning behavior:** Subclasses of the `Shape` class can override the `clone` method to provide a custom implementation for cloning themselves. This allows them to create deeper copies if necessary.
- **The class `ShapeRegistry` is the PrototypeRegistry:** It is responsible for maintaining a collection of shapes and provides methods to add, remove, and retrieve shapes.

2.3 Command Behavioral Pattern

1. Events as Commands:

- Each event, like `CreateShapeEvent`, implements the `Event` interface, which enforces the `apply()` and `revert()` methods.
- These methods encapsulate the actions associated with each event, allowing them to be treated as commands.
- This separation of actions from the executor (the `EventsHandler`) promotes modularity and flexibility.

2. Events Encapsulating Data:

- Events like `LocationChangeEvent` store necessary data associated with the action, like oldX, oldY, newX, newY.
- This encapsulation allows the command to be self-contained and independent of the context in which it's executed.

3. Events Handler as Invoker:

- The `EventsHandler` class acts as the invoker of the commands.
- It maintains a list of events and provides methods to add, execute (`apply`), and undo (`revert`) them.
- This centralizes the execution and management of commands, maintaining a history and enabling undo/redo functionality.

4. Decoupling Actions from Execution:

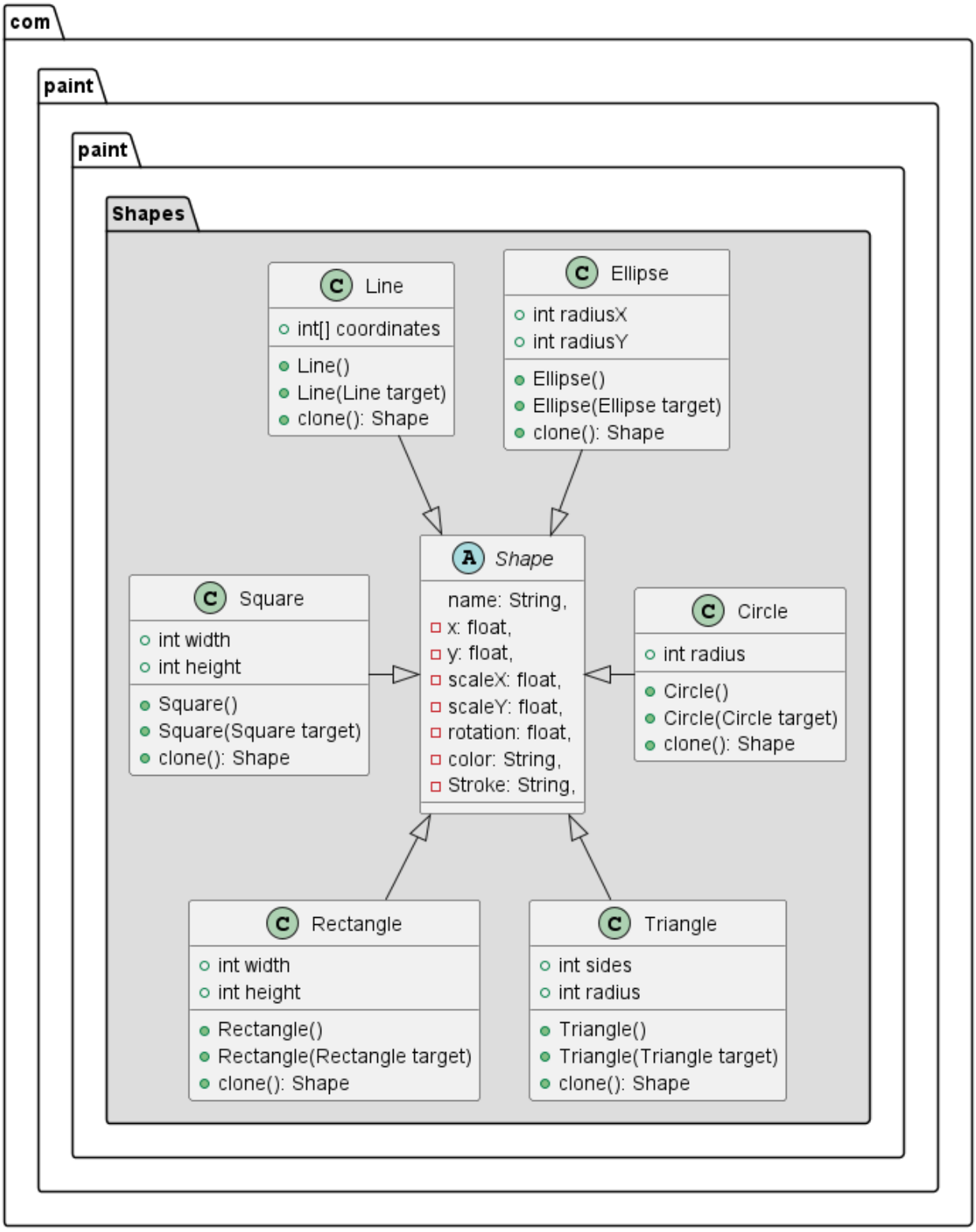
- The command pattern clearly separates the definition of actions from their execution.
- This allows the same event (command) to be executed in different contexts and by different triggers without affecting its internal logic.
- This promotes reusability and simplifies code organization.

5. Supporting Undo/Redo:

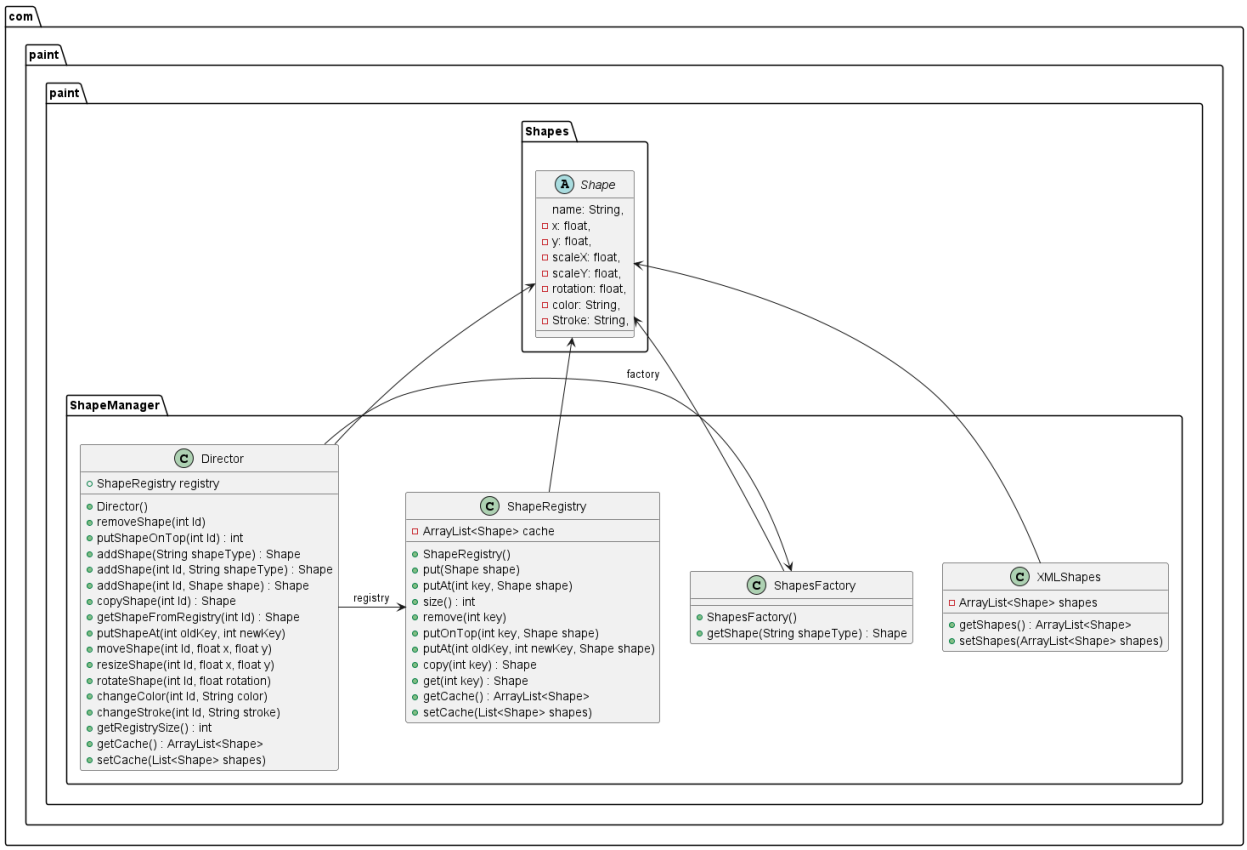
- The `EventsHandler` maintains a history of events and allows undoing and redoing them.
- By tracking the actions and their inverses, the system can revert past changes and restore previous states.
- This enhances user experience and provides a safety net for accidental actions.

3 UML Class Diagrams

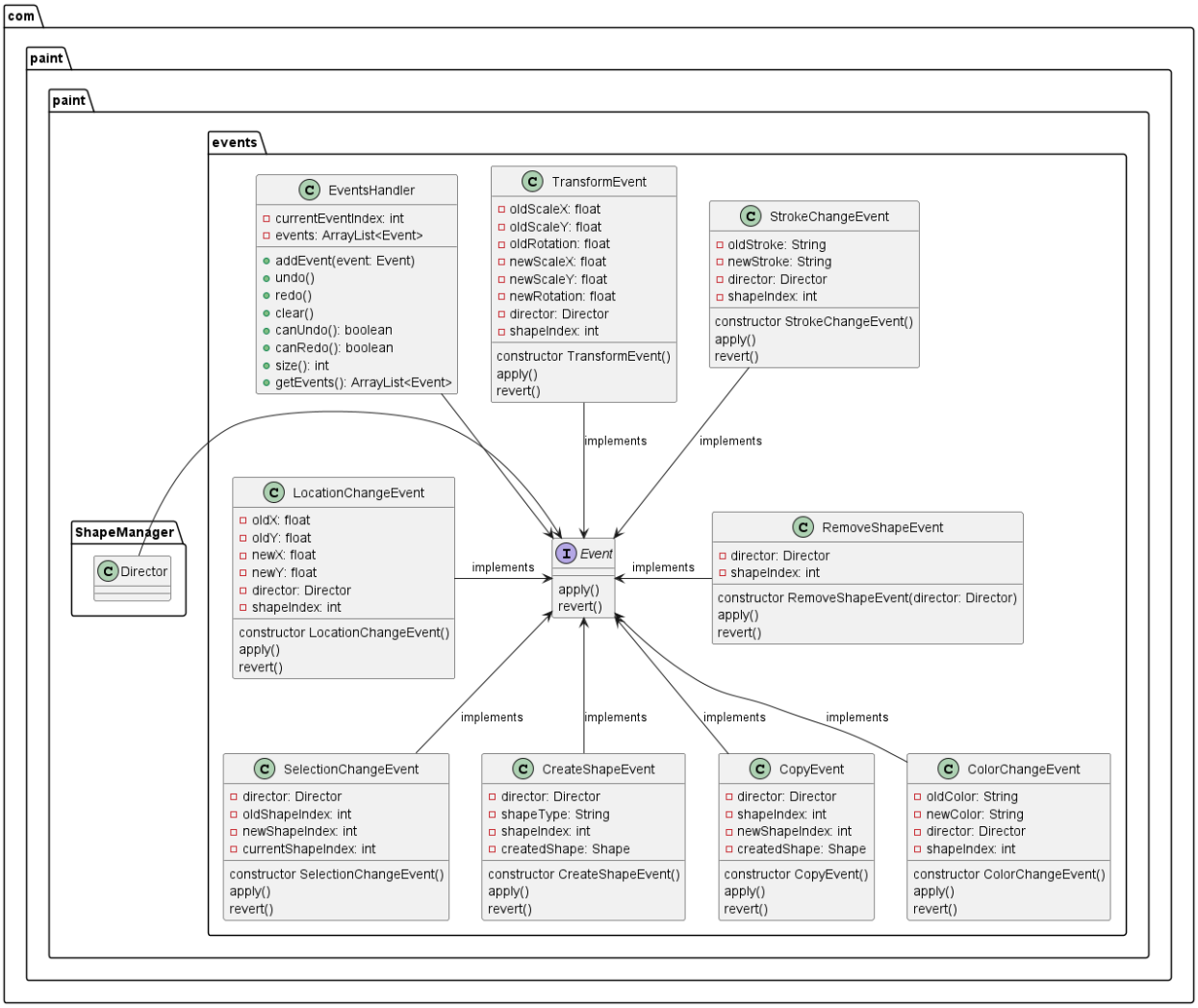
3.1 Shapes Package Diagram



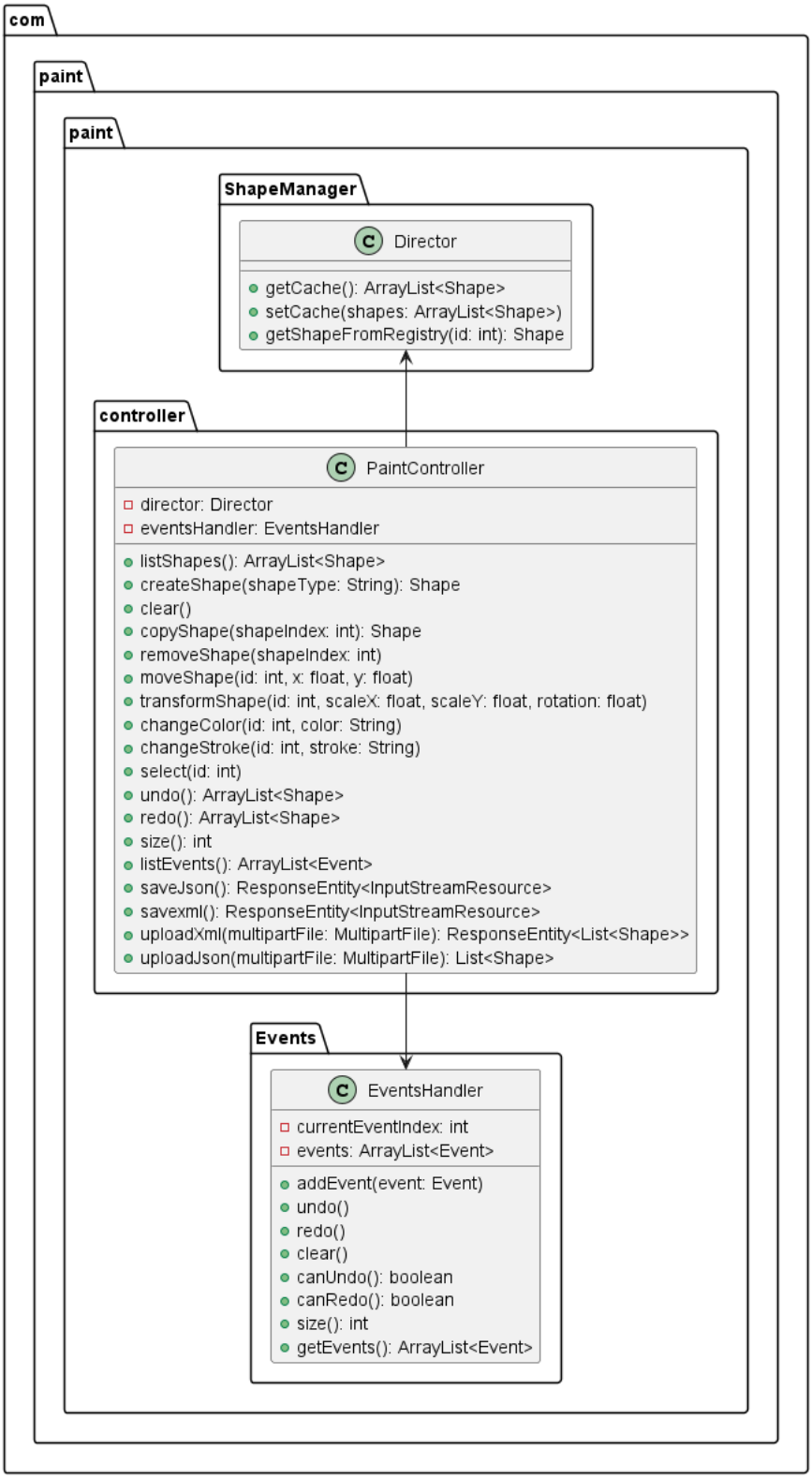
3.2 ShapeManager Package Diagram



3.3 Events Package Diagram



3.4 Controller Package Diagram



4 User Guide

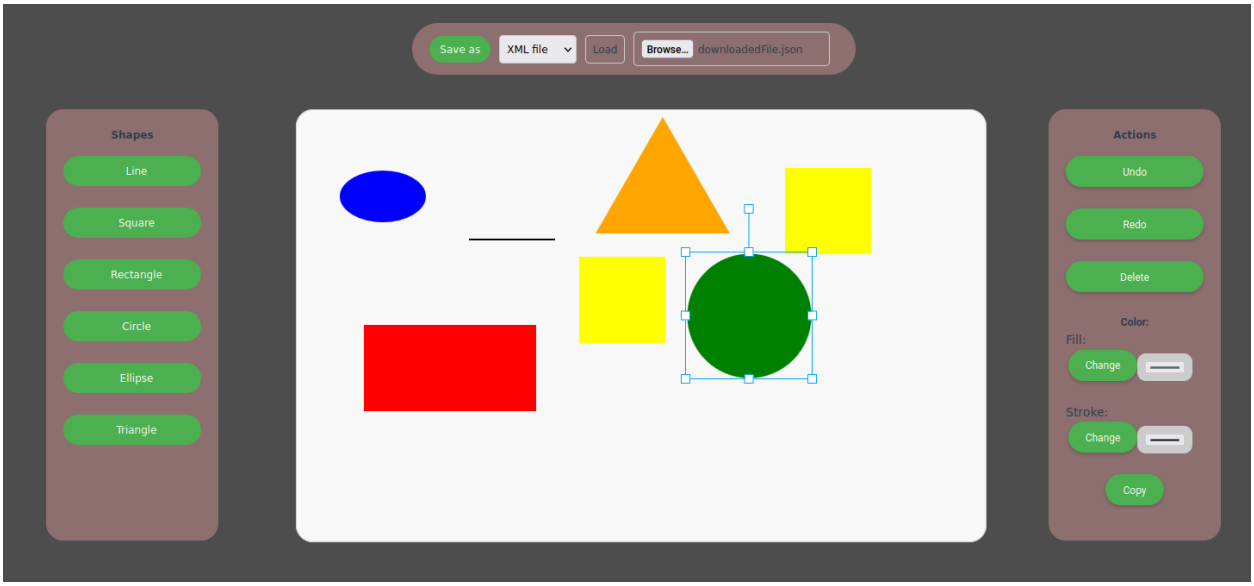


Figure 1: Paint app screenshot

4.1 Drawing Shapes

On the left sidebar, you'll find six drawable shapes: Line, Square, Rectangle, Circle, Ellipse, and Triangle. To add a shape to your canvas, simply drag it from the sidebar to your desired location.

4.2 Moving Shapes

You can easily move any shape on the canvas by dragging it to a new position.

4.3 Layering Shapes

Select a shape and click on it to bring it to the top layer, ensuring it appears in front of other shapes.

4.4 Transformations

Upon selecting a shape, you can modify its properties:

- Horizontal and Vertical Scale
- Aspect Ratio Resize
- Rotation

4.5 Undo and Redo

The right sidebar allows you to undo and redo your actions, providing a seamless editing experience.

4.6 Editing Colors

After selecting a shape, customize its appearance by changing:

- Fill Color
- Stroke Color

4.7 Copying Shapes

Easily duplicate a shape by copying and pasting it.

4.8 Save and Load

Use the top sidebar to save your drawings in either JSON or XML format. You can also load drawings from existing XML or JSON files.

Enjoy your artistic journey with our painting app! If you have any questions or feedback, feel free to reach out to us. Happy creating!