Name	ID
Mohamed Abdelrahman Anwar	20011634
Hussein Mohamed Mansour	20010499
Amr Ahmed Abd El Azim	20011037
Hussien Khaled Hussien Elsaid	20010494

Programming 2
Assignment 3 Report

Problem Statement:

-Design an object-oriented model for geometric shapes and apply the OOP concepts of inheritance and polymorphism to your design

-Design and implement a GUI that allows the following functionalities for the user on all the shapes: Draw, Color, Resize, Move, Copy, and Delete.

-Implement your application such that it would allow the user to undo or redo any action performed and the cursor should be used to select the location of a shape while drawing it, or moving it to another location.

-Provide an option in the UI to save the drawing in XML (encoding: ISO-8859-1) and JSON file.

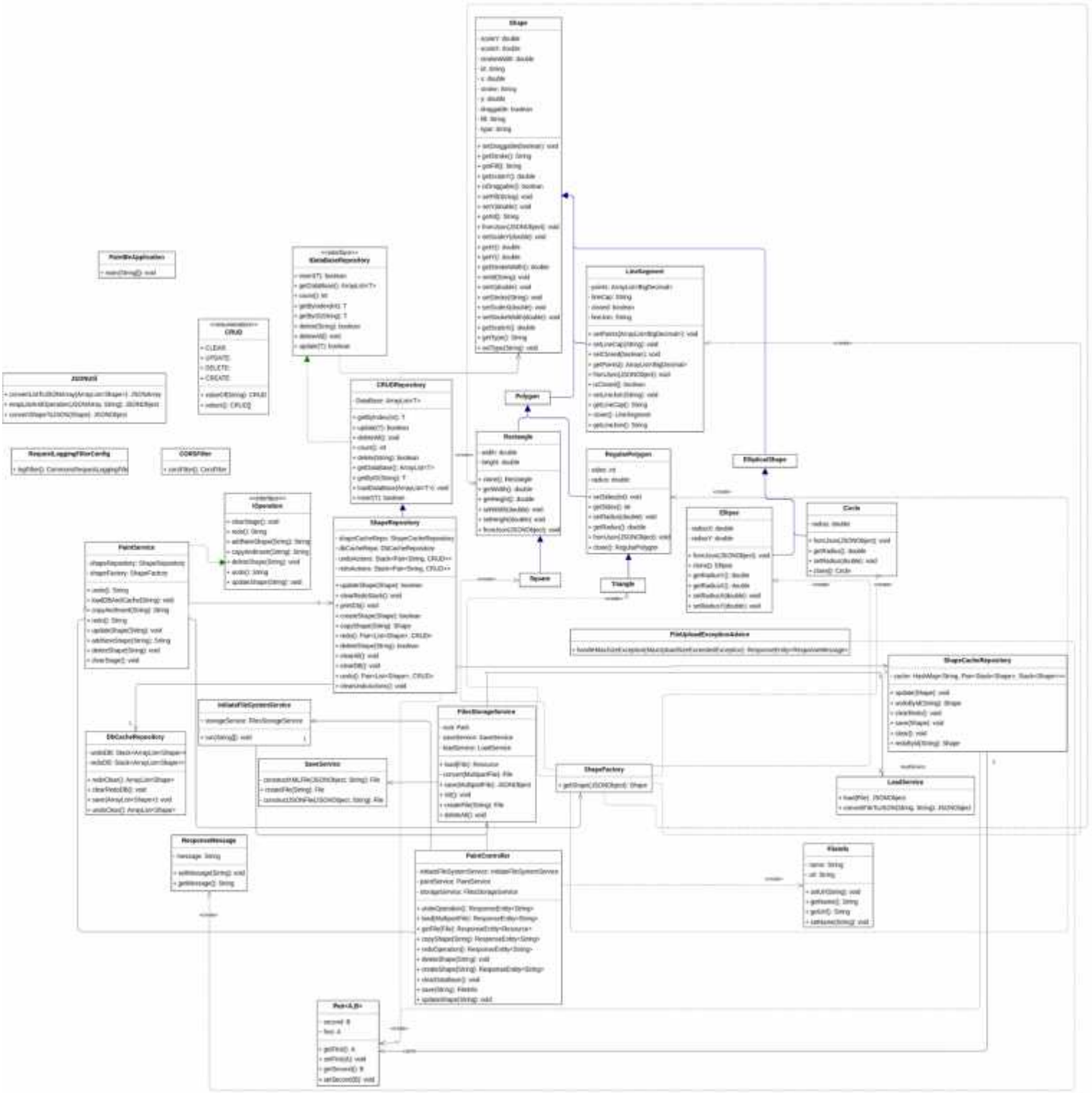
How-To-Document:

- Download the zip file containing the front and back folders.
- Open the back end folder using an ide, then run the project.
- If you face an error about the port used you can go to src/main/java/resources/application.resources and then change the server port, then run the project again.
- Open the front end folder using an ide.
- Make sure you have installed nodeJs on your computer, If not open the terminal and type “npm install”
- After installing you can serve your project on a local host by typing on the cmd (ng serve –open) Make sure you are on the project directory, Note: if you have changed the port in the back end make sure to update it in the shapeService file in the front folder.
- The site will be open and Paint application is ready to use

Github Repo links:

FrontEnd: https://github.com/husseinkhaled733/Web_Paint_Front
BackEnd: https://github.com/husseinkhaled733/Web_Paint_Back

UML Class Diagram:

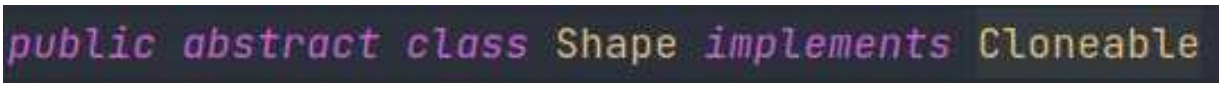


Design Patterns Used:

Factory: Created a Shapefactory class which is responsible for creating the required shape object which we will get the required type and attributes from the json sent from angular.

```
@Component
public class ShapeFactory {
    public Shape getShape(JSONObject object) {
        String shapeType = object.getJSONObject( key: "attrs").getString( key: "type");
        Shape shape = switch (shapeType) {
            case "Square" -> new Square();
            case "Rectangle" -> new Rectangle();
            case "RegularPolygon" -> new RegularPolygon();
            case "Triangle" -> new Triangle();
            case "Ellipse" -> new Ellipse();
            case "Circle" -> new Circle();
            case "LineSegment" -> new LineSegment();
            default -> null;
        };
        if (shape != null) shape.fromJson(object.getJSONObject( key: "attrs"));

        return shape;
    }
}
```

MarkerInterface: used the Clonable marker interface which the model implements to enforce the prototype pattern.



```
public abstract class Shape implements Cloneable
```

Prototype: By using the cloneable Marker Interface We enforce the prototype design pattern which makes the object cloneable without creating a new object or using the word new.

```java
@Override
public Shape clone() {
    try {
        return (Shape) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

- In case of objects which have reference types like (LineSegment) we need to add more steps to that method to make a deep copy of that object (since Object's class **clone()** method creates a swallow copy of that object which isn't suitable for our use case.

```java
@Override
public LineSegment clone() {
    LineSegment line = (LineSegment) super.clone();
    line.setPoints(new ArrayList<>(this.getPoints()));
    return line;
}
```

- Had to create a new object for the attribute points because it was a reference type , but there is another

approach with the **Serialization.Utils()** method which spring offers out of the box for deep cloning objects.

Singleton:

```
@Component
@Scope("singleton")
public class ShapeCacheRepository
```

```
@Component
@Scope("singleton")
public class DbCacheRepository
```

```
@Component
@Scope("singleton")
public class ShapeRepository extends CRUDRepository<Shape>
```

@Scope(“singleton”) provided by spring enforces one object from a class **ShapeCashRepository** and **DbCashRepository** are used to manage the cached data all over the program which is why we need only one instance during the runtime of the program , **ShapeRepository** which manages these classes including the parent class **CRUDRepository** which is the class managing the database and we only need one instance of it so we used singleton for it too.

Delegates and Interfaces:

Interfaces:

- **IOperation**: Holds the main operations of the program like (createShape , updateShape , undo , redo , save , load , …).
  ```
  public interface IOperation
  ```
- **ICRUDRepository**: Holds the main dataBase operations (CRUD : Create , Read , Update , Delete) and some more helping methods.

Delegates:

ShapeRepository acts as a delegate for executing the main operations of the **IOperation** interface

```
@Component
public class PaintService implements IOperation{

    private ShapeRepository shapeRepository;
    private final ShapeFactory shapeFactory;
```

Examples:

- ```
  @Override
  public String addNewShape(String json) {
      Shape shape = shapeFactory.getShape(new JSONObject(json));
      shapeRepository.createShape(shape);
  ```
  operation **addNewShape** or createShape is called from the **PaintController** which then the **PaintService** class delegates the operation to **ShapeRepository** class.
- ```
  @Override
  public void updateShape(String json) {
      Shape shape = shapeFactory.getShape(new JSONObject(json));
      shape.setId(new JSONObject(json).getJSONObject( key: "attrs").getString( key: "id"));
      shapeRepository.updateShape(shape);
  ```

Same as before for **updateShape** operation which is delegated to **ShapeRepository**.

Note: ShapeFactory can be considered a delegate for the PaintService which creates new shapes for it.

Design Decisions:

```
public interface ICRUDRepository<T extends Shape> {

    1 implementation
    boolean update(T obj);

    1 implementation
    ArrayList<T> getDataBase();

    1 implementation
    T getByID(String id);

    1 implementation
    T getByIndex(int index);

    1 implementation
    int count();

    1 implementation
    boolean delete(String id);

    1 implementation
    boolean insert(T shape);

    1 implementation
    void deleteAll();

}
```

We chose an ArrayList as data structure for our Database and we have our Driver which manages the Database,so

we can easily change it and this database saves all objects (shapes) in the front-end.

```java
@Component
@Scope("singleton")
public class DbCacheRepository {

    private final static Stack<ArrayList<Shape>> undoDB = new Stack<>();
    private final static Stack<ArrayList<Shape>> redoDB = new Stack<>();

    public void save(ArrayList<Shape> old){
        ArrayList<Shape> newDB = new ArrayList<>();
        old.forEach(shape -> newDB.add(shape.clone()));
        undoDB.push(newDB);
    }

    public ArrayList<Shape> undoClear(){
        redoDB.push(undoDB.pop());
        if (redoDB.isEmpty()) return null;
        return redoDB.peek();
    }

    public ArrayList<Shape> redoClear(){
        undoDB.push(redoDB.pop());
        if (undoDB.isEmpty()) return null;
        return undoDB.peek();
    }

    public void clearRedoDB() { redoDB.clear(); }

}
```

We choose Stack<ArrayList> for undo and redo which saves our previous database when the user clears it and DbCache Repository class to manage them.

```java
@Component
@Scope("singleton")
public class ShapeCacheRepository {

    // First for undo
    private final static HashMap<String, Pair<Stack<Shape>, Stack<Shape>>> cache = new HashMap<>();

    public void save(Shape shape) {
        if (!cache.containsKey(shape.getId())){
            Stack<Shape> undo = new Stack<>();
            Stack<Shape> redo = new Stack<>();
            undo.push(shape);
            Pair<Stack<Shape>,Stack<Shape>> history = new Pair<>(undo,redo);
            cache.put(shape.getId(), history);
        }
    }

    public void update(Shape shape) {
        Pair<Stack<Shape>,Stack<Shape>> history = cache.get(shape.getId());
        if (history == null){
            save(shape);
        }else {
            history.getFirst().push(shape); //undo
        }
    }
```

```java
public Shape undoById(String id) {
    Pair<Stack<Shape>,Stack<Shape>> history = cache.get(id);
    if (history != null) {
        Stack<Shape> undo = history.getFirst();
        Stack<Shape> redo = history.getSecond();
        redo.push(undo.pop());
        if (undo.isEmpty()) return null;
        return undo.peek();
    }else {
        return null;
    }
}

public Shape redoById(String id) {
    Pair<Stack<Shape>,Stack<Shape>> history = cache.get(id);
    if (history != null) {
        Stack<Shape> undo = history.getFirst();
        Stack<Shape> redo = history.getSecond();
        undo.push(redo.pop());
        return undo.peek();
    }else {
        return null;
    }
}

public void clear() {
    cache.clear();
}

public void clearRedo(){
    for (String key : cache.keySet()) cache.get(key).getSecond().clear();
}
```

We choose for save the previous state of every shape a

HashMap<String, Pair<Stack<Shape>, Stack<Shape>>> data structure .
The key is the ID of every shapeObject and the value is two stacks: the first of them is for undo and the second for the redo; these stacks save every change in the shapeObject and the **ShapeCacheRepository** manages its functions.

```
public enum CRUD {
    CREATE( name: "create"), UPDATE( name: "update"), DELETE( name: "delete"), CLEAR( name: "clear");

    CRUD(String name) {}
}
```

The **CRUD** is the enum of the operations which we made on the Shape Object.

```
@Component
@Scope("singleton")
public class ShapeRepository extends CRUDRepository<Shape> {

    private ShapeCacheRepository shapeCacheRepo;
    private DbCacheRepository dbCacheRepo;

    private final static Stack<Pair<String, CRUD>> undoActions = new Stack<>();
    private final static Stack<Pair<String, CRUD>> redoActions = new Stack<>();

    @Autowired
    public ShapeRepository(ShapeCacheRepository cacheRepository, DbCacheRepository dbCacheRepository) {
        this.shapeCacheRepo = cacheRepository;
        this.dbCacheRepo = dbCacheRepository;
    }

    public boolean createShape(Shape shape) {
        undoActions.push(new Pair<>(shape.getId(), CRUD.CREATE));
        shapeCacheRepo.save(shape.clone());
        return super.insert(shape);
    }


    public boolean updateShape(Shape shape) {
        undoActions.push(new Pair<>(shape.getId(), CRUD.UPDATE));
        shapeCacheRepo.update(shape.clone());
        return super.update(shape);
    }

    public void clearDB() {
        undoActions.push(new Pair<>("clear", CRUD.CLEAR));
        dbCacheRepo.save(getDataBase());
        deleteAll();
    }
```

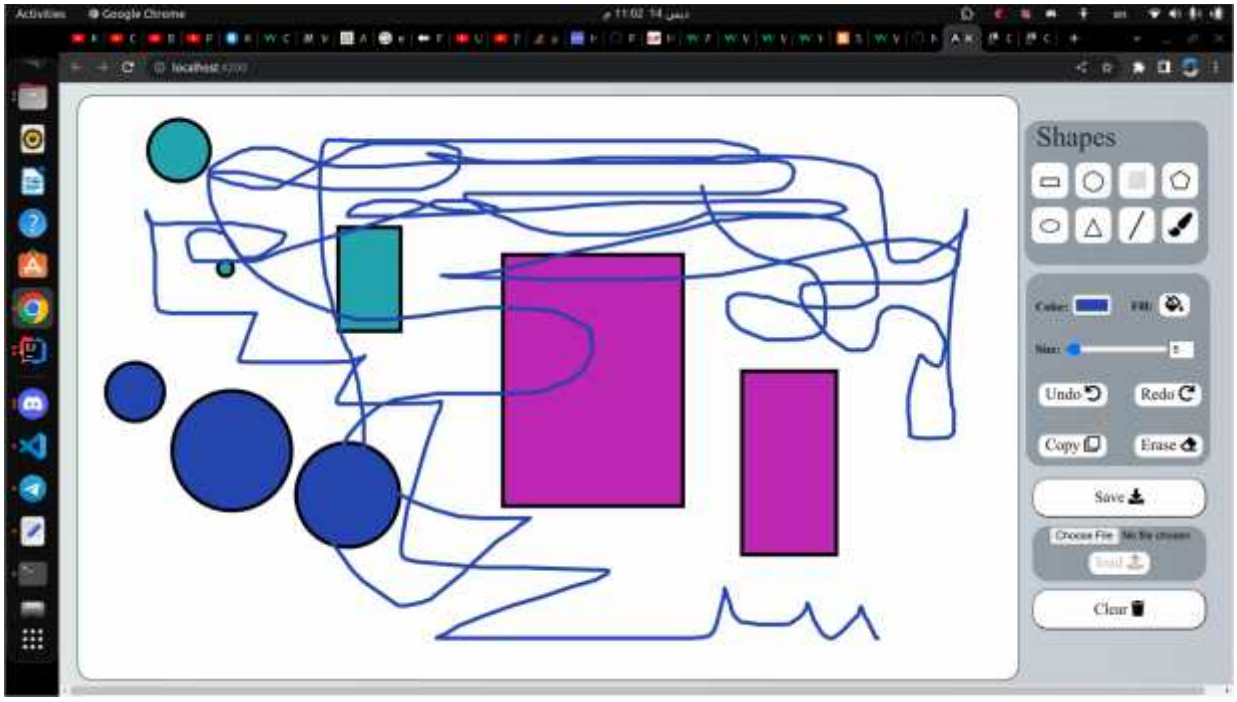
```java
public boolean deleteShape(String id) {
    undoActions.push(new Pair<>(id, CRUD.DELETE));
    shapeCacheRepo.update(getByID(id).clone());
    return super.delete(id);
}

public Shape copyShape(String id) {
    Shape shape = getByID(id).clone();
    shape.setId(UUID.randomUUID().toString());
    return shape;
}


public Pair<List<Shape>, CRUD> undo() {
    if (undoActions.isEmpty()) return null;
    Pair<String, CRUD> lastAction = undoActions.pop();
    redoActions.push(lastAction);
    String id = lastAction.getFirst();
    CRUD action = lastAction.getSecond();

    System.out.println(id);

    switch (action) {
        case CREATE -> {
            Shape shape = getByID(id);
            super.delete(id);
            shapeCacheRepo.undoById(id);
            return new Pair<>(List.of(shape), CRUD.DELETE);
        }
        case UPDATE -> {
            Shape shape = shapeCacheRepo.undoById(id);
            super.update(shape.clone());
            return new Pair<>(List.of(shape), CRUD.UPDATE);
        }
        case DELETE -> {
            Shape shape = shapeCacheRepo.undoById(id);
            super.insert(shape.clone());
            return new Pair<>(List.of(shape), CRUD.CREATE);
        }
        case CLEAR -> {
            ArrayList<Shape> cacheDb = dbCacheRepo.undoClear();
            super.loadDataBase(cacheDb);
            return new Pair<>(cacheDb, CRUD.CREATE);
        }
    }
    return null;
}
```

```java
public Pair<List<Shape>, CRUD> redo() {
    if (redoActions.isEmpty()) return null;
    Pair<String, CRUD> lastAction = redoActions.pop();
    undoActions.push(lastAction);
    String id = lastAction.getFirst();
    CRUD action = lastAction.getSecond();
    switch (action) {
        case CREATE -> {
            Shape shape = shapeCacheRepo.redoById(id);
            super.insert(shape.clone());
            return new Pair<>(List.of(shape), CRUD.CREATE);
        }
        case UPDATE -> {
            Shape shape = shapeCacheRepo.redoById(id);
            super.update(shape.clone());
            return new Pair<>(List.of(shape), CRUD.UPDATE);
        }
        case DELETE -> {
            Shape shape = shapeCacheRepo.redoById(id);
            super.delete(shape.getId());
            return new Pair<>(List.of(shape), CRUD.DELETE);
        }
        case CLEAR -> {
            var db :ArrayList<Shape> = dbCacheRepo.redoClear();
            super.deleteAll();
            return new Pair<>(null,CRUD.CLEAR);
        }
    }
    return null;
}

public void clearRedoStack(){
    redoActions.clear();
    dbCacheRepo.clearRedoDB();
    shapeCacheRepo.clearRedo();
}

public void clearUndoActions() { undoActions.clear(); }

public void clearAll(){
    deleteAll();
    shapeCacheRepo.clear();
    undoActions.clear();
    redoActions.clear();
}
```

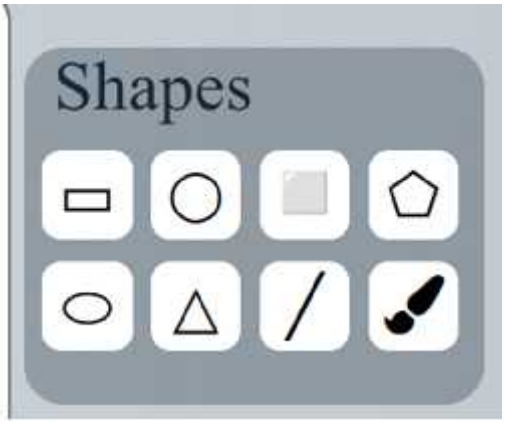
We have two **Stack**<Pair<String, CRUD>> one for undoActions and the second for redoActions, This stacks saves every operation in the order they were executed, The String represents the id of the object and the CRUD enum contains each operation made on the shape with that id.

The ShapeRepository class manages all the functions which we made on the shapeObject and calls the other caches class.

UI Snapshots:



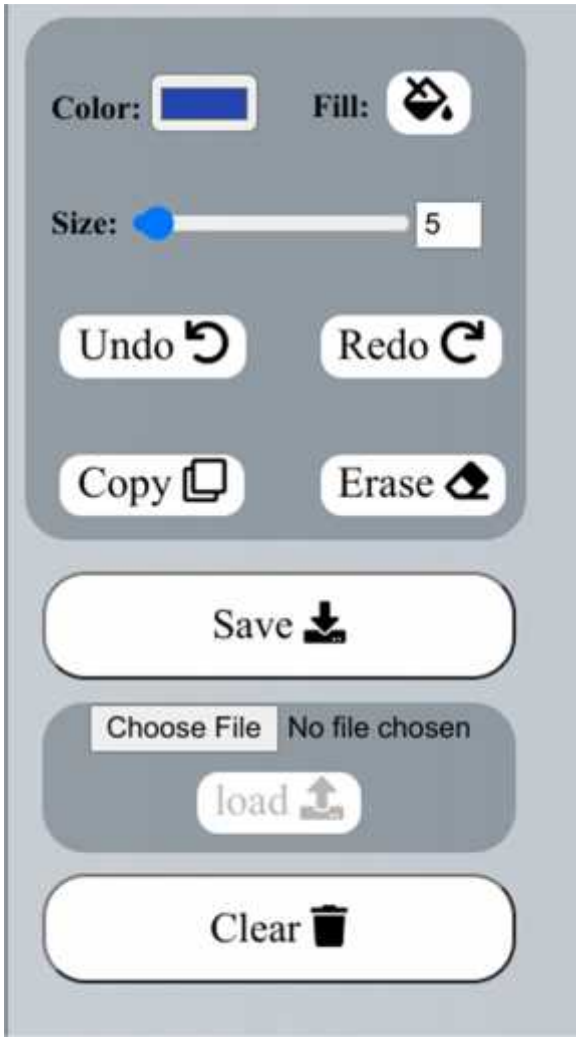
User Guide:



Our web page is classified into two sections: **board and options**, and everything is drawn on the board.

Shape Section:
If the user wants to draw any shape he selects the shape he wants and goes on the board and he can drag the shape and leave the mouse.

Any shape selected by its button would be hovered and the user can draw this shape while it’s hovering.
If the user wants to deselect the shape he clicks on it again.
If no button is selected the user could select any shape on the board and do the operations on them.
If a shape is selected,
The user could click on **Copy** and the shape will be copied
The user could click on **erase** and the shape will be deleted.



Any time the user could **undo** or **redo** by clicking on the buttons.

The user could **save** his folder by first choosing the type of the file and then he could type a name or not and then the file would be downloaded to the downloads on his computer.

The user could **load** a file from his computer by choosing a file from his computer first then clicking on load then the board will be loaded with the drawings he saved before.

The user could clear the board if he clicked on the clear button.