Painter web application

A fully designed usable web application using **Spring** and **VUE JS**

Github link:https://github.com/YoussifKhaled/Paint

Contributors:

1. Aly El-Din Mohamed El Sayed **21010835**
2. Mohamed Mahfouz Mohamed **21011210**
3. Esmail Mahmoud Hassan **21010272**
4. Youssif Khaled Ahmed **21011655**

Running process

The attached Pom.xml, Main.Js, vue.config.js, and index.html files serve the purpose of instantiating the required libraries for our project.
After Setting up the project using the Uploaded Source Code:

You should Use the Commands “npm install primevue” and “npm install konva”.

- It is advised to Modify the Pom.xml file to match the versions used on your machine.
- You should Use the Commands “npm install primevue” and “npm install konva”.

Design patterns

Factory method:

Usage

Create an object without exposing the creation logic to the client and refer to newly created object using a common interface, It’s used to create one family of products, It’s used to separate the responsibility of creating objects of other classes to another entity and it’s a tool to implement **Dependency Injection.**

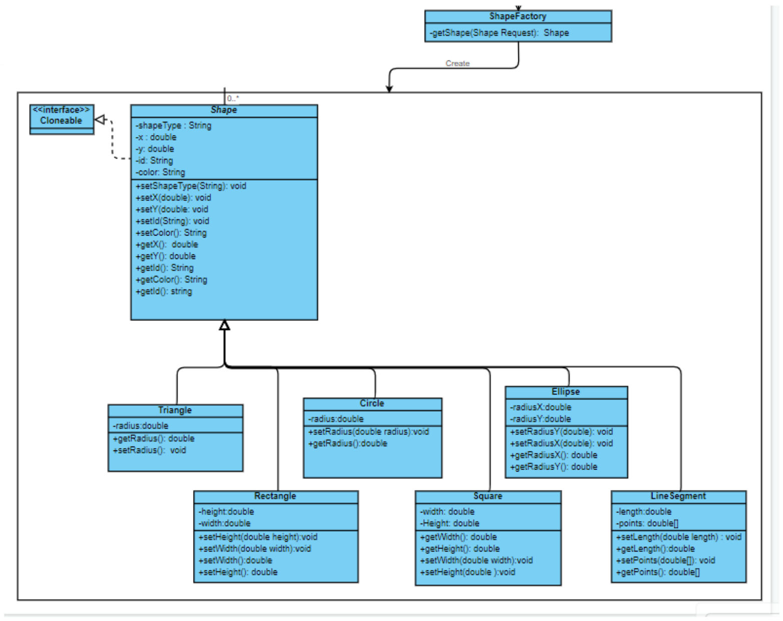
How It’s used in Painter application?

It’s used to create any instance of a concrete class to create the requested shape (Circle, Rectangle, Square, Ellipse, Triangle, Line segment) with customized or default attributes.

Consequences of using factory method?

- Encapsulation to what varies⇒ only one class is change when we add further shapes
- Program through an interface not to an implementation⇒ All calls are done to interface

UML



Code

```
@Service
public class ShapeFactory {
    public Shape getShape(ShapeRequest shapeRequest){
        return switch (shapeRequest.getType()) {
            case "circle" -> new Circle(shapeRequest);
            case "ellipse" -> new Ellipse(shapeRequest);
            case "rectangle" -> new Rectangle(shapeRequest);
            case "triangle" -> new Triangle(shapeRequest);
            case "line" -> new LineSegment(shapeRequest);
            case "square" -> new Square(shapeRequest);
            default -> null;
        };
    }
}
```

Prototype

Usage

Prototype design pattern is a creational design pattern that deals with the creation of objects, emphasizing the creation of objects by copying an existing object, known as the prototype. The main idea behind this pattern is to create new objects by copying an existing object, rather than creating them from scratch.

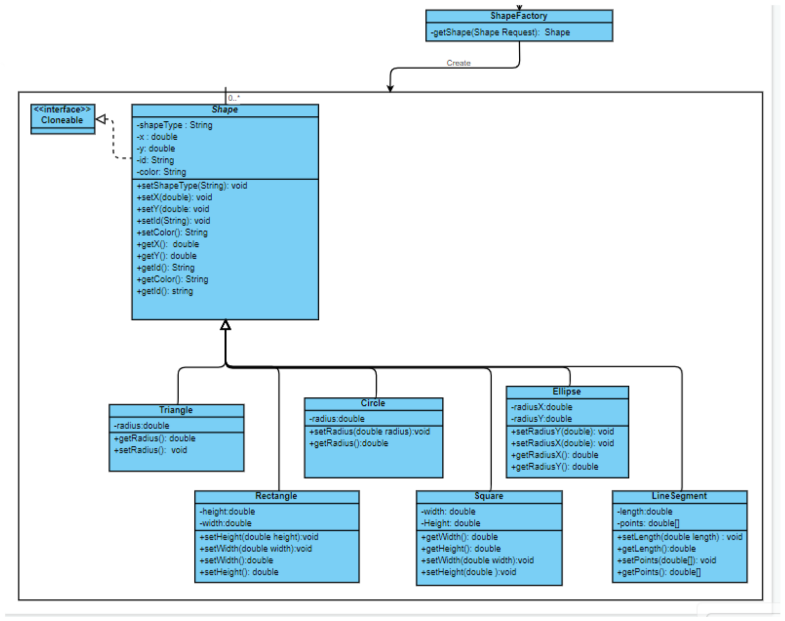
How It’s used in Painter application?

Painter application provides an option to the user to make a copy of a created shape, hence the application uses **Prototype** design pattern to implement this feature in the **Spring** application so as to have new shape with the same attributes of another shape so client could manipulate it independently of the pre-created shape.

Consequences of using prototype?

- Applying the design principle of separation of concerns, where the system is able to create objects without knowing their exact class, how they are created, or what data they represent.
- Classes to be instantiated are not known by the system until runtime, when they are acquired on the fly by techniques such as dynamic linkage.

UML



Code

```
//Shape class in Controller package
public abstract class Shape implements Cloneable{
    //Shape attributes
    private String id;
    protected double x, y;
    protected String color;
    private String type;
    //Default constructor
    public Shape(){}
    //Customized constructor
    public Shape(ShapeRequest shapeRequest){
        this.x = shapeRequest.getX();
        this.y = shapeRequest.getY();
        this.type = shapeRequest.getType();
        this.id = shapeRequest.getId();
        this.color=shapeRequest.getFill();
    }
    //Copying a shape
    public Object Clone(){
        try{
            return super.clone();
        }
        catch (CloneNotSupportedException e){
            throw new InternalError();
        }
    }
}

//Cloning service in Service package
public Shape getClone(String id){
        Shape clonedShape;
        for(Shape shape:shapes){
            if(shape.getId().equals(id)){
                clonedShape=(Shape)shape.Clone();                                  //Cloning the requested shape
                clonedShape.setId(Long.toString(System.currentTimeMillis()));      //Setting a new ID to the cloned shape
                clonedShape.setX(shape.getX()+15);                                 //Setting new shifted coordinates to the cloned shape
                clonedShape.setY(shape.getY()+15);                                 //...
                addShape(clonedShape);                                             //Adding the cloned shape to the list of shapes
                return (Shape) clonedShape.Clone();                                //Returning a copy of the cloned shape
            }
        }
        return null;
    }
//Getting the copy of the requested shape
@GetMapping("/copy/{id}")
public Shape copyShape(@PathVariable String id){
    return shapeService.getClone(id);
}
```

Singleton

Usage

The Singleton design pattern is a creational design pattern that ensures a class has only one instance and provides a global point to this instance. This pattern is useful when exactly one object is needed to coordinate actions across the system.

How It’s used in Painter application?

Painter application provides to the clients saving XML and JSON files to be able to load the saved shapes again and modify them.

The entity (Service class) which is responsible to implement this feature is “DataBase” class, where an instance of it is created only one time during the program.

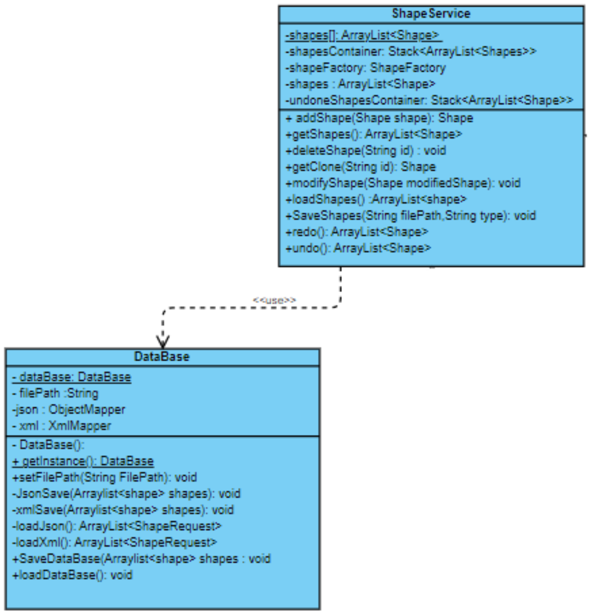
Using this service enables the client to set his/her file path (the file to which the client wants to save) and save designs, then load any file of the saved files.

The intent of using this design pattern to implement this feature is that an instance of it is created only once.

Consequences of using Singleton

- Provides an object creation one time only during the program, so It avoids unnecessary object creation, hence If an instance doesn't exist, it creates one; otherwise, it returns the existing instance.
- Lazy initiallization

UML


ShapeService
-shapes[]: ArrayList<Shape>
-shapesContainer: Stack<ArrayList<Shapes>>
-shapeFactory: ShapeFactory
-shapes : ArrayList<Shape>
-undoneShapesContainer: Stack<ArrayList<Shape>>
+ addShape(Shape shape): Shape
+getShapes(): ArrayList<Shape>
+deleteShape(String id) : void
+getClone(String id): Shape
+modifyShape(Shape modifiedShape): void
+loadShapes() :ArrayList<shape>
+SaveShapes(String filePath,String type): void
+redo(): ArrayList<Shape>
+undo(): ArrayList<Shape>
<<use>>
DataBase
- dataBase: DataBase
- filePath :String
-json : ObjectMapper
- xml : XmlMapper
- DataBase():
+ getInstance(): DataBase
+setFilePath(String FilePath): void
-JsonSave(Arraylist<shape> shapes): void
-xmlSave(Arraylist<shape> shapes): void
-loadJson(): ArrayList<ShapeRequest>
-loadXml(): ArrayList<ShapeRequest>
+SaveDataBase(Arraylist<shape> shapes : void
+loadDataBase(): void

Code

```
private static DataBase dataBase;
private DataBase(){
    xml = new XmlMapper();
    json = new ObjectMapper();
 }
public static DataBase getInstance() {
        if(dataBase == null)
            dataBase = new DataBase();
        return dataBase;
 }
```

Design process

Service package

It’s the package which is designed to implement services (classes) which the client benefits from.

ShapeFactory class

Class in which the factory method design pattern is implemented to serve the program creation of shapes

DataBase class

Intent

Class is designed to implement loading and saving data in either **XML** or **JSON** files.

It includes all functions and helper functions which do so:

- setFilePath: Function to set up the file path in which shapes are saved
- JsonSave: Function to save design in a JSON file
- xmlSave: Function to save design in a XML file
- loadXMl:Function to load design in a XML file

ShapeService class

Class is designed to provide multi webservices to the client such as:
add, delete, modify, get, get clone, save, load, undo, redo, clear all

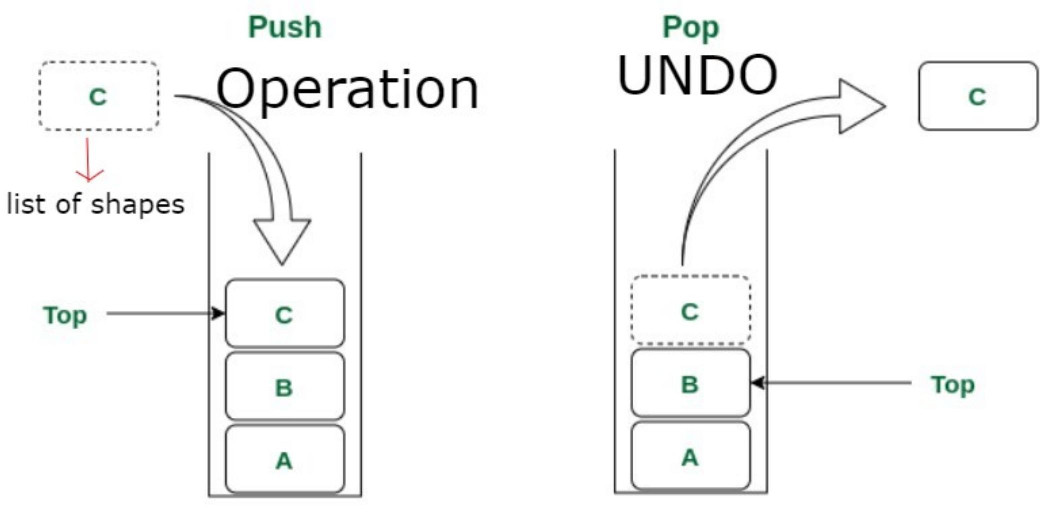
Undo and redo algorithm

Used data structure: Stack

Algorithm:

two stacks are created, one for storing every single action (create, modify, clone, etc..)

and the other stores undo operations so that we could redo



Cases handled:

- User could not Redo without undo
- The flow of undo operations is break when any action is taken (Ex: when you undo then create a new shape or modify any attribute, you could not redo because no the flow of undo is broken
- Client could undo all operation until empty canvas, then redo all of them againS

Model package

ShapeRequest class

Class content

Class includes all attributes of all concrete classes do exist, in addition to setters and getters

Intent

It’s designed to ease the communication between the VUE application and Spring application, where JSON data is transferred and received as “shapeRequest” class.

Shape class

It’s an abstract super class includes the main attributes of any shape (coordinates, color, etc..)

Concrete classes

6 classes, each of them represents the shape would be designed and each of them inherit from **Shape** class.

Shape controller class

It’s the entity which contains the methods where the frontend app requests.

Design purposes

- **Refresh**

  When Client refreshes the web page, the created shapes still exist as they are.

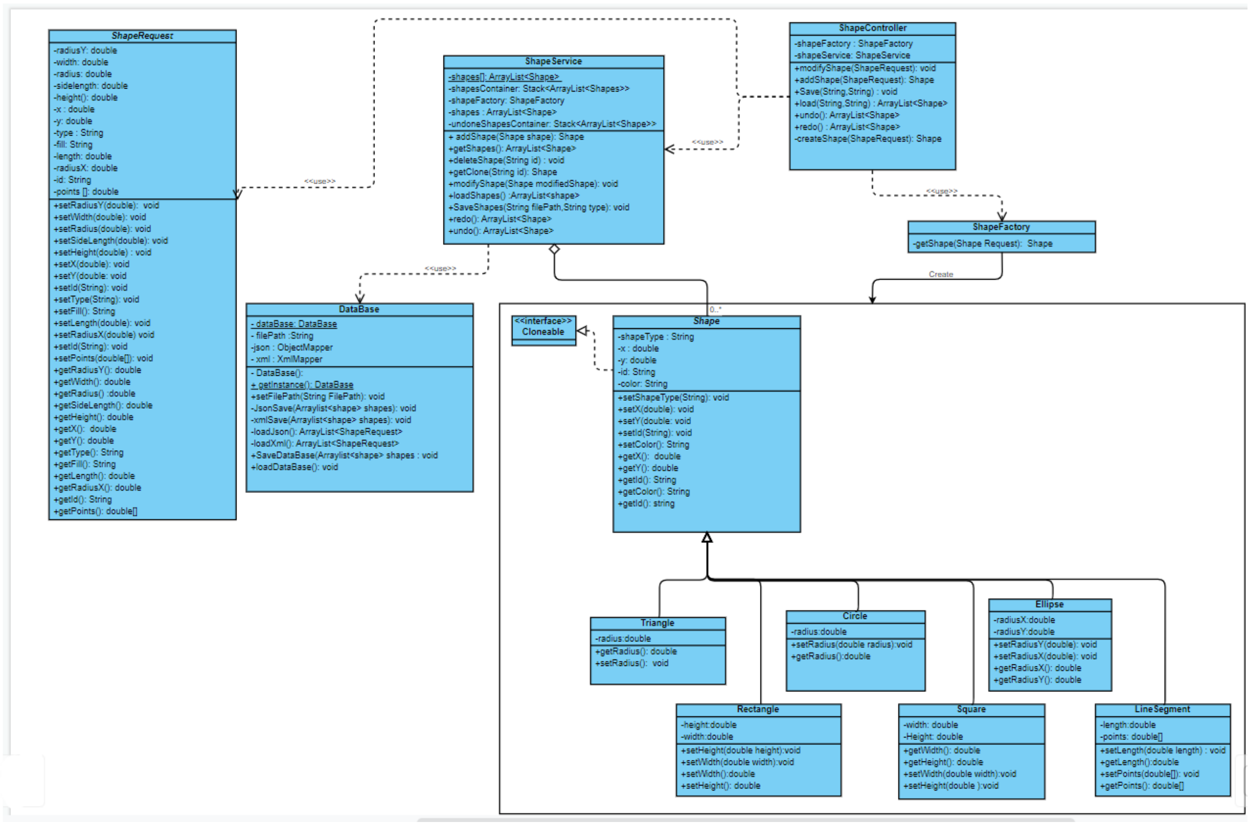
- **Clear**

  Client could easily clear all created shapes and start from scratch providing the undo option to undo the clear action itself.

- **Delete**

  Client could easily delete any created shape providing undo action

UML


ShapeRequest
-radiusY: double
-width: double
-radius: double
-sidelength: double
-height(): double
-x : double
-y: double
-type : String
-fill: String
-length: double
-radiusX: double
-id: String
-points []: double
+setRadiusY(double): void
+setWidth(double): void
+setRadius(double): void
+setSideLength(double): void
+setHeight(double) : void
+setX(double): void
+setY(double: void
+setId(String): void
+setType(String): void
+setFill(): String
+setLength(double): void
+setRadiusX(double) void
+setId(String): void
+setPoints(double[]): void
+getRadiusY(): double
+getWidth(): double
+getRadius() :double
+getSideLength(): double
+getHeight(): double
+getX(): double
+getY(): double
+getType(): String
+getFill(): String
+getLength(): double
+getRadiusX(): double
+getId(): String
+getPoints(): double[]
<<use>>
ShapeService
-shapes[]: ArrayList<Shape>
-shapesContainer: Stack<ArrayList<Shapes>>
-shapeFactory: ShapeFactory
-shapes : ArrayList<Shape>
-undoneShapesContainer: Stack<ArrayList<Shape>>
+ addShape(Shape shape): Shape
+getShapes(): ArrayList<Shape>
+deleteShape(String id) : void
+getClone(String id): Shape
+modifyShape(Shape modifiedShape): void
+loadShapes() :ArrayList<shape>
+SaveShapes(String filePath,String type): void
+redo(): ArrayList<Shape>
+undo(): ArrayList<Shape>
<<use>>
ShapeController
-shapeFactory : ShapeFactory
-shapeService: ShapeService
+modifyShape(ShapeRequest): void
+addShape(ShapeRequest): Shape
+Save(String,String) : void
+load(String,String) : ArrayList<Shape>
+undo(): ArrayList<Shape>
+redo() : ArrayList<Shape>
-createShape(ShapeRequest): Shape
<<use>>
ShapeFactory
-getShape(Shape Request): Shape
Create
<<use>>
DataBase
- dataBase: DataBase
- filePath :String
-json : ObjectMapper
- xml : XmlMapper
- DataBase():
+ getInstance(): DataBase
+setFilePath(String FilePath): void
-JsonSave(Arraylist<shape> shapes): void
-xmlSave(Arraylist<shape> shapes): void
-loadJson(): ArrayList<ShapeRequest>
-loadXml(): ArrayList<ShapeRequest>
+SaveDataBase(Arraylist<shape> shapes : void
+loadDataBase(): void
0..*
<<interface>>
Cloneable
Shape
-shapeType : String
-x : double
-y: double
-id: String
-color: String
+setShapeType(String): void
+setX(double): void
+setY(double: void
+setId(String): void
+setColor(): String
+getX(): double
+getY(): double
+getId(): String
+getColor(): String
+getId(): string
Triangle
-radius:double
+getRadius(): double
+setRadius(): void
Circle
-radius:double
+setRadius(double radius):void
+getRadius():double
Ellipse
-radiusX:double
-radiusY:double
+setRadiusY(double): void
+setRadiusX(double): void
+getRadiusX(): double
+getRadiusY(): double
Rectangle
-height:double
-width:double
+setHeight(double height):void
+setWidth(double width):void
+setWidth():double
+setHeight(): double
Square
-width: double
-Height: double
+getWidth(): double
+getHeight(): double
+setWidth(double width):void
+setHeight(double):void
LineSegment
-length:double
-points: double[]
+setLength(double length) : void
+getLength():double
+setPoints(double[]): void
+getPoints(): double[]