**Student name:** Peter Ehmann

**Student ID:** 216313918

**Course:** EECS3311 Sec. B Lab 01

**TA:** Naeiji Alireza

**Display and Sort Shapes**

## Part I: Introduction

This project challenges the programmer to create software that will generate a list of 6 shapes of types **Circle**, **Rectangle**, and **Square** that are randomly generated and will be sorted using a specific sorting algorithm. The goal of this project is to design and implement a solution to this.

The main challenge to this assignment will be the overall design of the class structure of the project. Having multiple classes dealing with different methods can get messy so it is important to draw up a clear plan of what each class will be responsible for before any coding is done.

OOP principles such as **inheritance**, **abstraction**, as well as **encapsulation** will be heavily stressed and utilized. For example, inheritance will be used to link the shape class to each of the individual shape classes of Circle, Square, and Rectangle. Since they are all shapes, those classes will extend the shape class thus creating an **is-a** relationship which is the OOD principle known as inheritance. Abstraction will be used when implementing our getArea() method, since the area equations for circles are different from squares and rectangles, and thus we would need to override the parent class’s getArea() method. Encapsulation will be used throughout the classes in order to keep certain variables hidden from classes so as to not cause confusion between specific class variables.

This report will consist of two UML diagrams of possible class structures that one could use to complete this assignment, followed by an in-depth explanation of the sorting algorithm used, some information about the coding environment used, a video on how to open the project, as well as some concluding remarks.

## 

## 

## Part II: Design of the solution

The two design principles I have used in my software are the **Singleton Pattern** as well as the **Factory Pattern**. When initializing my frame called “myFrame”, the next line in my Main method adds a singleton instance of my ShapePanel class to myFrame. The ShapePanel class is important as it is responsible for holding important fields such as shapeList, as well as the fact that it allows me to use paintComponent to draw shapes. Since the constructor of this class is private, the class can never be instantiated as an object, thus demonstrating the singleton design pattern. Furthermore, the second design pattern I have used in my design is the factory pattern. This design pattern comes in handy when I was implementing my shape classes: Shape, Circle, Square, and Rectangle. This technique allowed me to declare all of my defining shape variables like x, y, height, width, and colour in the same class without having to duplicate code. Since Circles, Squares, and Rectangles all share those same variables, it didn’t make sense to give all of those classes the same fields. Another way that this design pattern helped out was by allowing me to override certain methods such as draw() as well as getArea(). Since circles have a different area equation from a square or rectangle, overriding the parent method in the child classes allowed me to place specific instructions to calculate area and to draw the shapes in their respective classes.

My secondary design is one that is less optimized than my first and it suffers from what is successful in the first class diagram. The main change here is that the shape classes are handling the creation of the shapes themselves instead of allowing it to be routed through the shape class. This means that instead of declaring shape variables that are common to all in one class, each shape class is responsible for its own variables and it becomes way more redundant (as you can see by the lengths of the classes in the diagram). This is an example of poor class design and it is why we make designs in the first place.

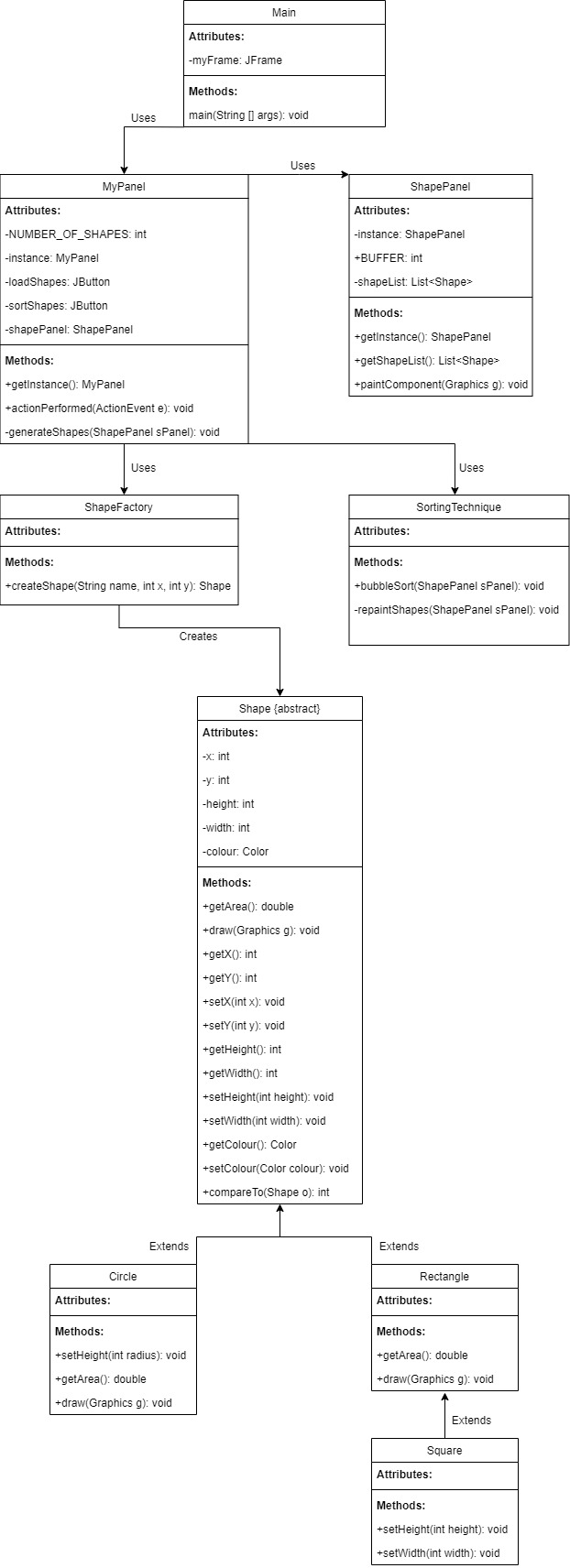
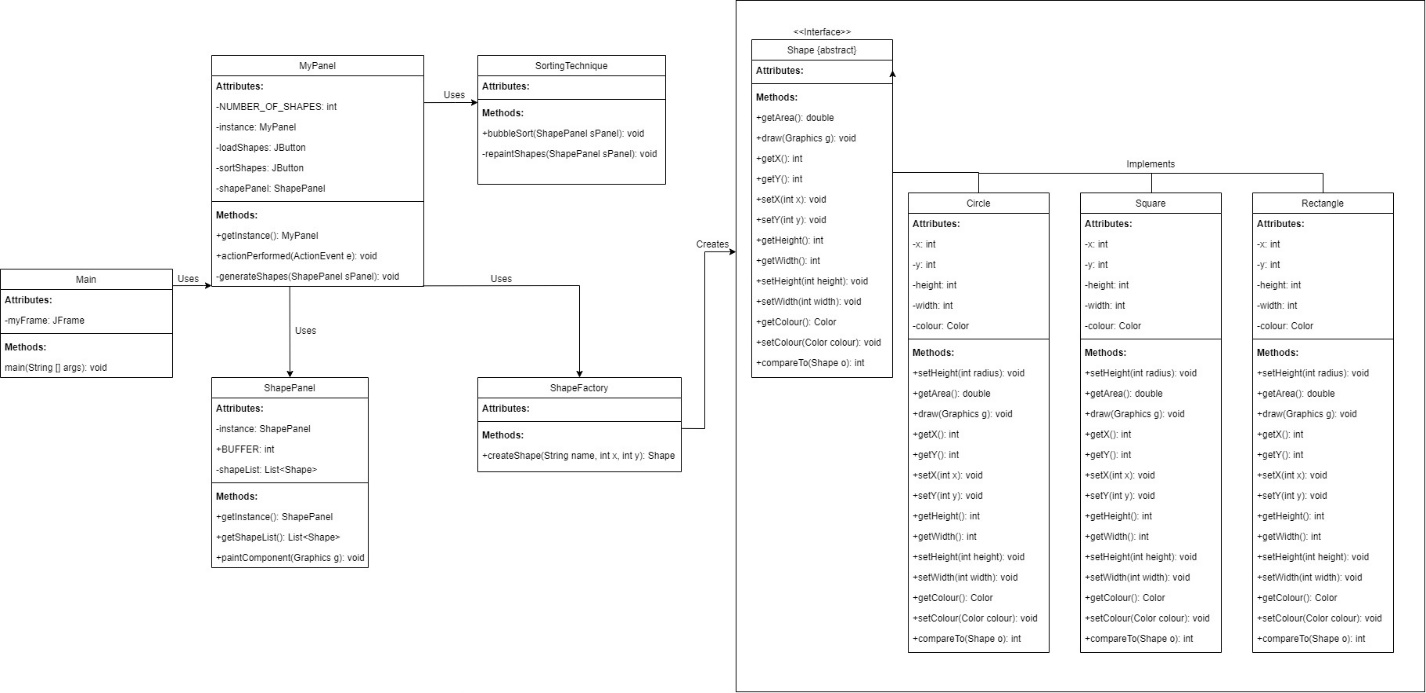
 Figure 1: Primary Class Diagram 

Figure 2: Secondary Class Diagram

## Part III: Implementation of the solution

The sorting technique I used to sort the ArrayList of shape objects is known as the **Bubble Sort** method of sorting. This technique utilizes two for loops that iteratively search the list and compare adjacent shape area values. It does this by using the **Comparable Interface** found in the abstract shape class.From our comparison, we see if the shape’s area at element i is greater than element i + 1. If this is the case, we create a temporary variable to store the shape of element i while setting the shape of element i + 1 to the initial position of the shape of element i on the list. Once a swap either happens or does not (if the area is not greater), the iterator moves on and compares the next adjacent pair which compares i + 1 with i + 2 and so on until the swaps are complete. After the loops are finished running, the shapes have been sorted by area of least to greatest in ascending order.

Below will be a list of my classes as well as some information about them and how they were implemented composing my solution:

**public class Main:** My main class is used to house the frame in which my buttons will be added to and also acts as the canvas where my shapes will be painted to and displayed to the user.

**public class MyPanel extends JPanel implements ActionListener**: The MyPanel class is responsible for creating the panel in which both buttons are added to as well as the Shape Panel where the shapes will be stored. Since this class implements an actionlistener, there is a method named **actionPerformed(ActionEvent e)** that handles whether the load or sort button is pressed and directs the code to do as the buttons are meant to do. If the sort shapes button is pressed, the code redirect to our SortingTechnique class, but if the load shapes button is pressed the method named **generateShapes(ShapePanel sPanel)** is called and the shapes are generated randomly according to the **createShape(name, x, y)** method located in the ShapeFactory class.

**public class ShapeFactory**: The createShape class is namely responsible for the creation of our shapes! Considering the name that is passed as a parameter, this class takes the necessary information required to build shapes and passes it to the specific shape classes for creation. After the shape is created, an output statement is recorded to the console that reads, for example: **“A Circle was created with an area of 4185”**. Six of these messages are generated (due to our six shapes being created) and the shape created is returned to the generateShapes(ShapePanel sPanel) method in the MyPanel class and it is added to our shapePanel. The panel is then repainted.

**public class SortingTechnique**: Namely, the sortingTechnique class is responsible for sorting the ArrayList of shape objects using the bubble sort technique explained in the first answer of part III of this report. This class also houses the repaintShapes(ShapePanel sPanel) method that is responsible for repainting the shapes once they’ve been sorted.

**public class ShapePanel extends JPanel**: This class contains the all important ArrayList named shapeList of type <Shape>. It has an accessor method called getShapeList that is used quite often in the SortingTechnique class to retrieve the current list of shapes. And finally, the vital paintComponent(Graphics g) method which is used to paint all of the shapes within shapeList.

**public abstract class Shape implements Comparable<Shape>**: This class is our main hub class for all of our child classes (Circle, Rectangle). This class contains variables pertaining to the creation of shapes such as x, y, height, width, and colour. This class is declared abstract due to the fact that the classes Circle, Rectangle, and Square have different ways of calculating area and drawing from one another. These methods are abstract and are overridden to perform their specific functions in the child classes. The compareTo(Shape o) method is also found in this class.

**Circle, Rectangle, and Square classes**: These classes are where the shapes are instantiated. The constructors for these classes have parameters that contain x, y, radius (or) width and height, and colour. They also contain overridden methods like getArea() and draw(Graphics g) that specify how to calculate area and draw the specific shape of the class.

Tools:

IntelliJ IDEA 2021.2.2 (Community Edition)

Build #IC-212.5284.40, built on September 13, 2021

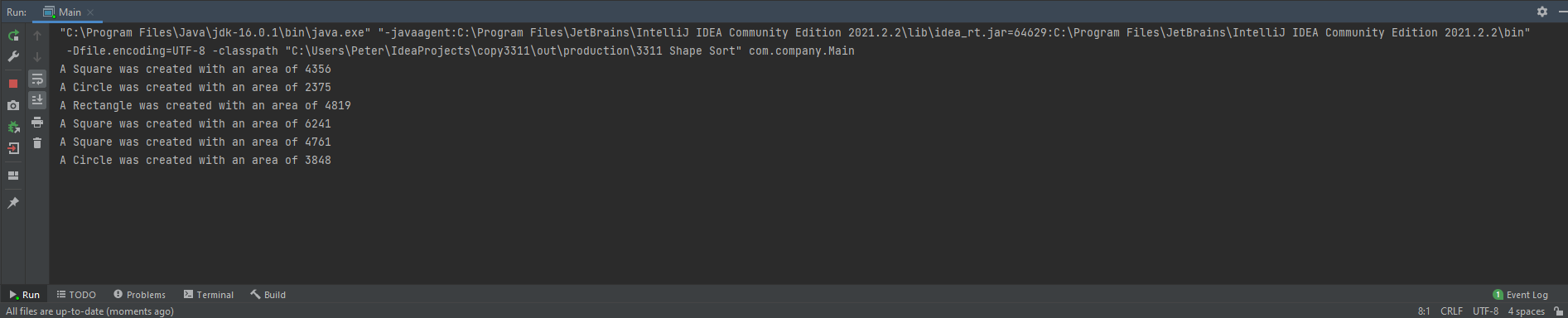
Runtime version: 11.0.12+7-b1504.28 amd64

VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.

Windows 10 10.0

Kotlin: 212-1.5.10-release-IJ5284.40

Snapshot:



Video:



Instructions:

1. In Eclipse go to File → Import → Git → Projects from Git (With Smart Import) → Clone URI
2. Under the green Code: copy and paste the URI from the github repository to the URI section at the top of the Location box. <https://github.com/Peterthen00b/hello-world.git>
3. For user, enter in your GitHub credentials for User but for Password you’ll need a personal access token found on GitHub: Click on your profile → Settings → Developer Settings → Personal Access Tokens → Generate New Token → Copy that long string of characters and paste that into the Password field in eclipse in the Authentication box.
4. Uncheck “main” in the branches box (as it is unnecessary) and then click next.
5. Click “Browse” and direct it to an empty directory as to store the files on your computer then hit next.
6. Click finish.

## Part IV: Conclusion

Creating the initial frame as well as the individual shape classes. I knew I would need to create a factory class for shapes that would dictate which individual shapes I was making and that went smoothly. I had a decent idea as per what was required as I had written a similar software back in highschool.

After finalizing most of my code I ran into a little problem where my paintComponent method found in my ShapePanel class was not painting but it was later resolved and I found that it was in fact a panel issue. I probably should have drawn out my class diagrams to get a big picture of how the classes would interact with each other before getting started.

I learned that I need to do more planning before attempting to code. Having a diagram or written plan would be beneficial for easier method implementation. The modularity of solving many small problems instead of larger ones would do me good.

My recommendations to someone starting this software project would be to 1: Draw up a plan, 2: Break up the requirements into many smaller problems, and 3: Modularity is your best friend, using small methods that do specific things can save you from writing redundant code.