Marlon Lee EECS 3311 Lab 3
217 129 230 Section A TA: Kazi Mridul

Part 1(Introduction):

For this software project, the main objectives were to create a Java GUI application that displays and sorts shapes with the use of two buttons that the user can click and interact with. The load button instantiates and creates six randomly generated shapes that can either be a circle, square or a rectangle with a randomly given height and width. A circle is defined with a framing rectangle where its width and height have equal values. A rectangle is defined by an area in the coordinate space with a height and width of different values. A square is defined as a rectangle but with equal height and width. Each shape is given a random colour and is displayed on the interface, positioned with its set upper left x and y coordinates. The sort button sorts the six shapes' area accordingly from smallest area to largest area (positioned from the top left corner to the bottom right corner). The goals for this project were to ultimately be familiar with the design patterns and provide a chance to apply and implement such knowledge such as the object oriented design principles.

The challenges associated with this project was properly displaying and generating the random shapes. At first, I didn't really know how to approach the problem of generating a random type of shape as well as randomly generated height and width values. To solve this problem, I implemented some helper functions to return a random type of shape which was either a rectangle, circle or square. In my shape factory class, I had two methods to assign a random colour and also to assign a random width and height value that was between a set minimum and maximum value. This also helped in ensuring that the shapes would not overlap or intersect each other.

The concepts that were used in this project were object oriented design principles such as encapsulation, inheritance, polymorphism and abstraction. As well, I used the factory design pattern and the singleton design pattern.

As I conclude the introduction of this software project: explaining what the project is about, outlining the objectives and goals, explaining some of the challenges, and explaining the concepts and design patterns that were used, I will now proceed to explain how the rest of the report would be structured. In the next section, I provide a UML class diagram of this system to show the design workflow and design process that were done to complete the project's goals and objectives. I also provide an additional UML class diagram as an alternative design and comparison to the original design shown in the UML class diagram. In part three, I describe the implementation stage of the process, by explaining the sorting algorithm used to sort the shapes as well as the tools used to help implement this project. Finally, in the concluding section, I highlight what went well and what didn't go so well and also reflect upon what I have learned after completing this project. In my concluding statements, I offer three recommendations that would be helpful in completing the project.

Part 2 (Design):

As shown in the UML class diagram, I used object oriented design principles in my design such as inheritance, abstraction, encapsulation, and polymorphism.

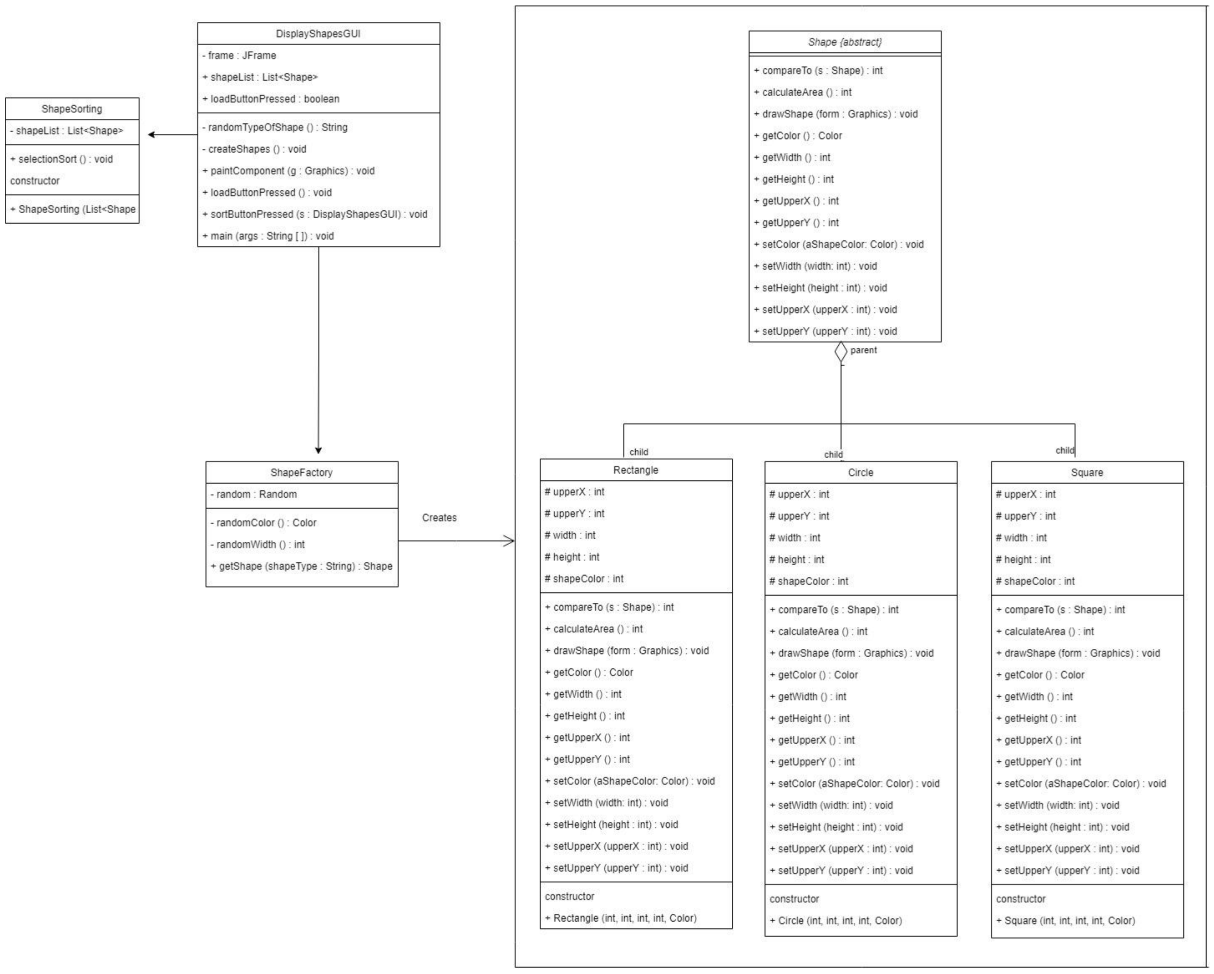

ShapeSorting
- shapeList : List<Shape>
+ selectionSort () : void
constructor
+ ShapeSorting (List<Shape
DisplayShapesGUI
- frame : JFrame
+ shapeList : List<Shape>
+ loadButtonPressed : boolean
- randomTypeOfShape () : String
- createShapes () : void
+ paintComponent (g : Graphics) : void
+ loadButtonPressed () : void
+ sortButtonPressed (s : DisplayShapesGUI) : void
+ main (args : String [ ]) : void
ShapeFactory
- random : Random
- randomColor () : Color
- randomWidth () : int
+ getShape (shapeType : String) : Shape
Creates
Shape {abstract}
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
parent
child
child
child
Rectangle
# upperX : int
# upperY : int
# width : int
# height : int
# shapeColor : int
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
constructor
+ Rectangle (int, int, int, int, Color)
Circle
# upperX : int
# upperY : int
# width : int
# height : int
# shapeColor : int
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
constructor
+ Circle (int, int, int, int, Color)
Square
# upperX : int
# upperY : int
# width : int
# height : int
# shapeColor : int
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
constructor
+ Square (int, int, int, int, Color)

Figure 2.1: UML class diagram

In my design, I have included the factory design pattern as well as the singleton design pattern. The factory design pattern is included as I have a Shape factory class that has a getShape method, which returns a particular child instance of a shape. In this way, the object is created without exposing the knowledge of creation to the client.

I used inheritance with the following classes: Rectangle, Square, and Circle all extend Shape which is the parent class that is abstract. In addition, the main class called DisplayShapesGUI extends JPanel which is its parent class. Encapsulation is used as evident by the getter and setter methods in Rectangle, Square, Circle and Shape class. Abstraction is used with the abstract class Shape and its abstract methods. This allows the child classes to provide its own implementation of the abstract methods. Polymorphism is achieved when creating the shape objects. In the createShapes method in the DisplayShapesGUI class, the shapes are initialized with type Shape but are assigned with either a Circle, Rectangle, or a Square instance via the getShape method from the shapeFactory class.

An alternative design is shown with the following UML class diagram.

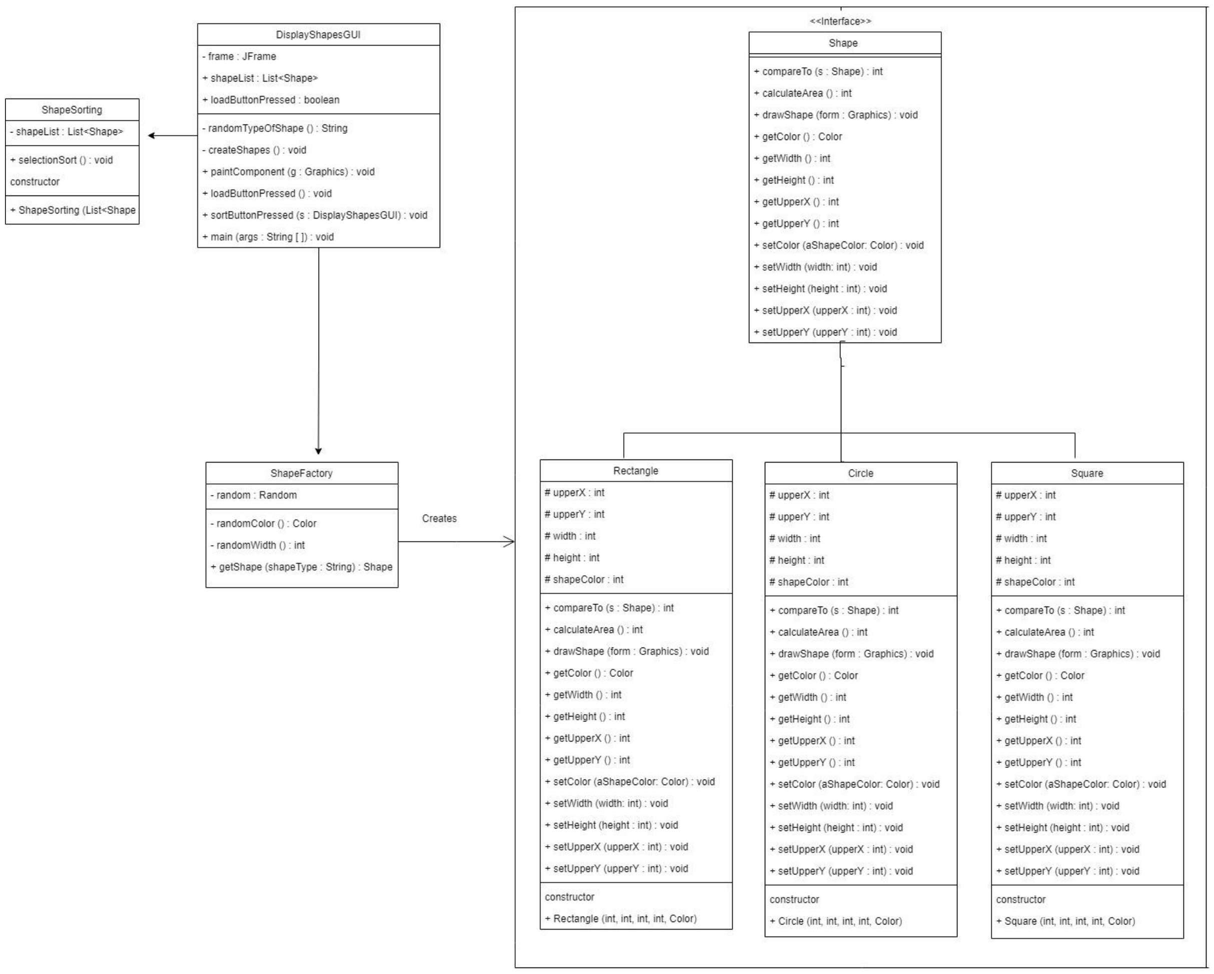

ShapeSorting
- shapeList : List<Shape>
+ selectionSort () : void
constructor
+ ShapeSorting (List<Shape
DisplayShapesGUI
- frame : JFrame
+ shapeList : List<Shape>
+ loadButtonPressed : boolean
- randomTypeOfShape () : String
- createShapes () : void
+ paintComponent (g : Graphics) : void
+ loadButtonPressed () : void
+ sortButtonPressed (s : DisplayShapesGUI) : void
+ main (args : String [ ]) : void
ShapeFactory
- random : Random
- randomColor () : Color
- randomWidth () : int
+ getShape (shapeType : String) : Shape
Creates
<<Interface>>
Shape
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
Rectangle
# upperX : int
# upperY : int
# width : int
# height : int
# shapeColor : int
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
constructor
+ Rectangle (int, int, int, int, Color)
Circle
# upperX : int
# upperY : int
# width : int
# height : int
# shapeColor : int
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
constructor
+ Circle (int, int, int, int, Color)
Square
# upperX : int
# upperY : int
# width : int
# height : int
# shapeColor : int
+ compareTo (s : Shape) : int
+ calculateArea () : int
+ drawShape (form : Graphics) : void
+ getColor () : Color
+ getWidth () : int
+ getHeight () : int
+ getUpperX () : int
+ getUpperY () : int
+ setColor (aShapeColor: Color) : void
+ setWidth (width: int) : void
+ setHeight (height : int) : void
+ setUpperX (upperX : int) : void
+ setUpperY (upperY : int) : void
constructor
+ Square (int, int, int, int, Color)

Figure 2.2: An alternative design

I think that the first class diagram yields a better design because an abstract class provides functionality between closely related objects which is true in this case with shapes and the child objects being types of shapes. In contrast, an interface is better used to state functionality but not to implement it.

Part 3 (Implementation):

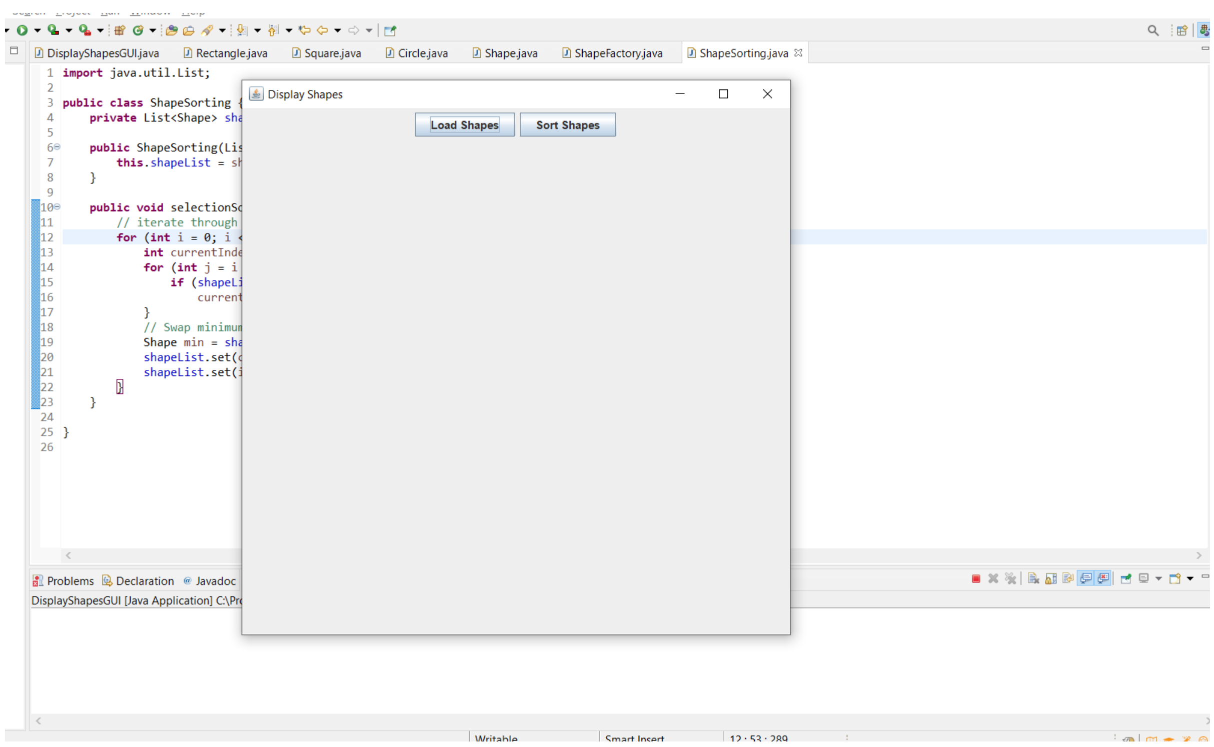
The sorting technique I used to sort the shapes was the selection sort algorithm. Basically for this algorithm, I iterate through the shape array list, and I search for the shape with the least area by using the shape's compareTo method. The element that has the least area is then swapped with the element with the current index being used to iterate through the list. In this way, there are effectively two parts in selection sort where the input list is broken down into a sorted list and an unsorted list. By the end, the sorted list should be sorted from the shape with the least area to the greatest area.

For the implementation, I implemented the first UML class diagram (see Part Two: Design). For the main class which I named DisplayShapesGUI, in the main method: I initialize a 600 by 600 JFrame where I add a JPanel with the two buttons (load and sort buttons) added onto the panel. Additionally, I only add an action listener for the load button but the sort button does not do anything when clicked. However, when the user selects load, the method loadButtonPressed is called. In this method, the boolean field (called loadButtonPressed) is set to true and I instantiate a DisplayShapesGUI instance which extends JPanel. I clear the frame by removing the current panel and I add the instantiated DisplayShapesGUI panel to the frame where I also add the two buttons to the panel once again. This time, I add action listeners to both buttons so that load and sort has functionality when clicked.

In the paintComponent method where the shapes are drawn and displayed, I check if the boolean field is true, which means the load button was clicked, and so I call the method to create the shapes. In this method, an instance of the shape factory is instantiated, and I instantiate six shapes with the shape factory getShape() method of randomly selected shape types with the help of a helper function (randomTypeOfShape). Next, I set the upper x and upper y coordinates of each shape to achieve a diagonal like pattern. Then, I initialize the shapeList field which is an Array List of type Shape for the shapes to be added to. Thus, when the paintComponent calls this method, the shapeList (which is initially empty) is initialized where the array list can then be iterated through and each shape calls its drawShape method for the shape to be displayed on the interface.

In regards to the sorting of the shapes, after shapes are displayed onto the screen and the sort button is clicked, the sortButtonPressed method is called and its parameters are the current JPanel which is the DisplayShapeGUI instance that was created when the load button was selected. From there, an instance of the ShapeSorting is instantiated which is the class that handles the sorting of the shapes. This instance calls its selectionSort method to sort the array list of shapes from the shape with the least area to the greatest area. Next, I reset the positions of the shapes by setting the upper x and upper y coordinates of each shape so that the new sorted shapes would be displayed in the correct order. Also, the boolean field is set to false so that in the paintComponent method, it does not call the method that will create a new set of random shapes. The panel is revalidated and repainted to display the updated positions of the shapes.

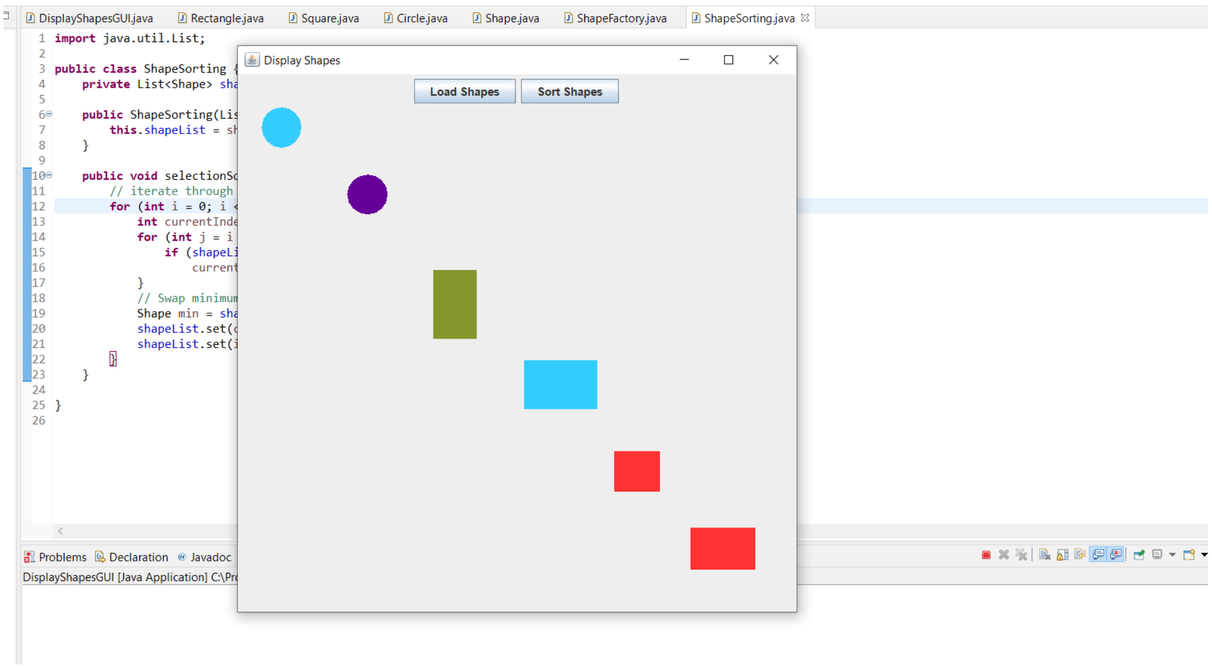
The tools I used during the implementation were the Eclipse IDE (Version 4.16.0) to write the code and run it, as well as JDK 12.0.1.0.



```
import java.util.List;

public class ShapeSorting {
    private List<Shape> sha

    public ShapeSorting(Lis
        this.shapeList = sh
    }

    public void selectionSo
        // iterate through
        for (int i = 0; i <
            int currentInde
            for (int j = i
                if (shapeLi
                    current
            }
            // Swap minimum
            Shape min = sha
            shapeList.set(c
            shapeList.set(i
        }
    }

}
```

DisplayShapesGUI [Java Application] C:\Pr

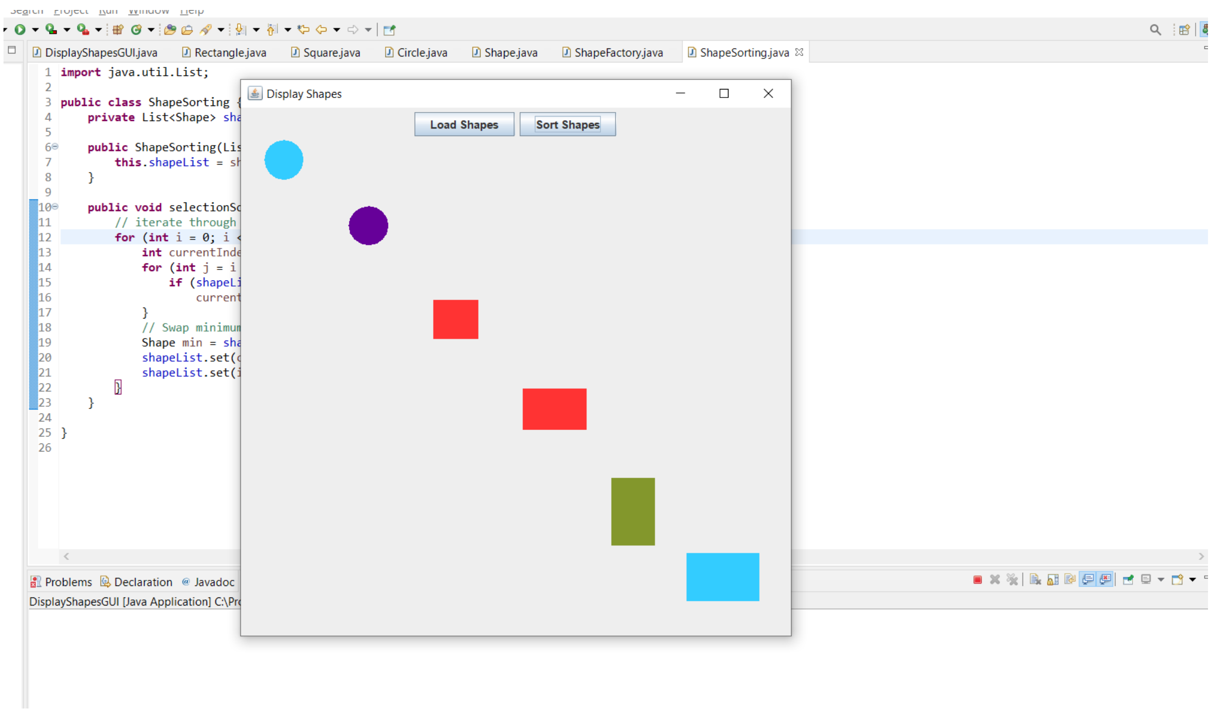
Figure 3.1: GUI interface that appears when a user first runs the application.



```
import java.util.List;

public class ShapeSorting {
    private List<Shape> sha

    public ShapeSorting(Lis
        this.shapeList = sh
    }

    public void selectionSo
        // iterate through
        for (int i = 0; i <
            int currentInde
            for (int j = i
                if (shapeLi
                    current
            }
            // Swap minimum
            Shape min = sha
            shapeList.set(c
            shapeList.set(i
        }
    }

}
```

DisplayShapesGUI [Java Application] C:\Pr

Figure 3.2: Six randomly generated shapes are displayed when "Load Shapes" is selected.



```
import java.util.List;

public class ShapeSorting {
    private List<Shape> sha

    public ShapeSorting(Lis
        this.shapeList = sh
    }

    public void selectionSo
        // iterate through
        for (int i = 0; i <
            int currentInde
            for (int j = i
                if (shapeLi
                    current
            }
            // Swap minimum
            Shape min = sha
            shapeList.set(c
            shapeList.set(i
        }
    }

}
```

DisplayShapesGUI [Java Application] C:\Pr

Figure 3.3: The shapes are sorted by their area when "Sort Shapes" button is selected

Marlon Lee EECS 3311 Lab 3
217 129 230 Section A TA: Kazi Mridul

Part 4 (Conclusion):

Reflecting on this software project, the parts that went well was getting the GUI interface working such as displaying the JPanel with the two buttons. I found some degree of success in displaying the shapes with the load button. Also, I didn't really encounter many issues with implementing my sorting algorithm to sort the shapes.

Conversely, the component that took the longest for me to resolve was displaying the shapes after sorting them with the sort button. For example, after clicking the load button to generate the six shapes and then after clicking the sorting button, a new set of six shapes were generated. It turned out that in the paintComponent() method, I was calling the method to instantiate the shapes for both the load and sort buttons. Therefore, I had a boolean field that would be assigned true in the method called when the load button was selected. As well, this boolean field would be assigned false in the method being called when the sort button was selected. Thus, in the paintComponent() method, if the boolean flag is true, then that means the load button was selected and the method to create a new set of shapes could be called.

Ultimately, I felt that I learned much from the software project, especially about java swing and object oriented design principles. Before doing this project, I did not have much experience with Java swing but now after completion, I feel that I have grasped a lot of the basic concepts such as creating a JFrame, JPanel, and JButtons as well as using action listeners. In regards to the design principles and concepts, I feel that I learned a lot about using object oriented design principles such as inheritance, abstraction, encapsulation and polymorphism.

In conclusion, I want to end off with offering three recommendations that would be useful and help ease the process in completing this project. Firstly, I think that the design process is very important and so it should be done at the start instead of just jumping in to code right away. This would also help in ensuring that you understand the requirements and objectives for this project. Secondly, I think that reading and researching about using Java Swing beforehand, would be very beneficial especially if you have not had much experience with Java Swing. Understanding how to create a JFrame, JPanel, JButtons and how to display objects on the screen with the paintComponent was crucial to properly display and paint the shapes for this project. Finally, I think that effectively documenting your code was very helpful in easily identifying errors and bugs in the code. A lot of the time, I had trouble finding the source of a bug somewhere in my code and so using commenting methods and variables helped remind me of what parts of the method was actually doing.