Bryce Cooke, 217346354, EECS3311 A, Software\_project\_EECS3311, Kazi Mridul

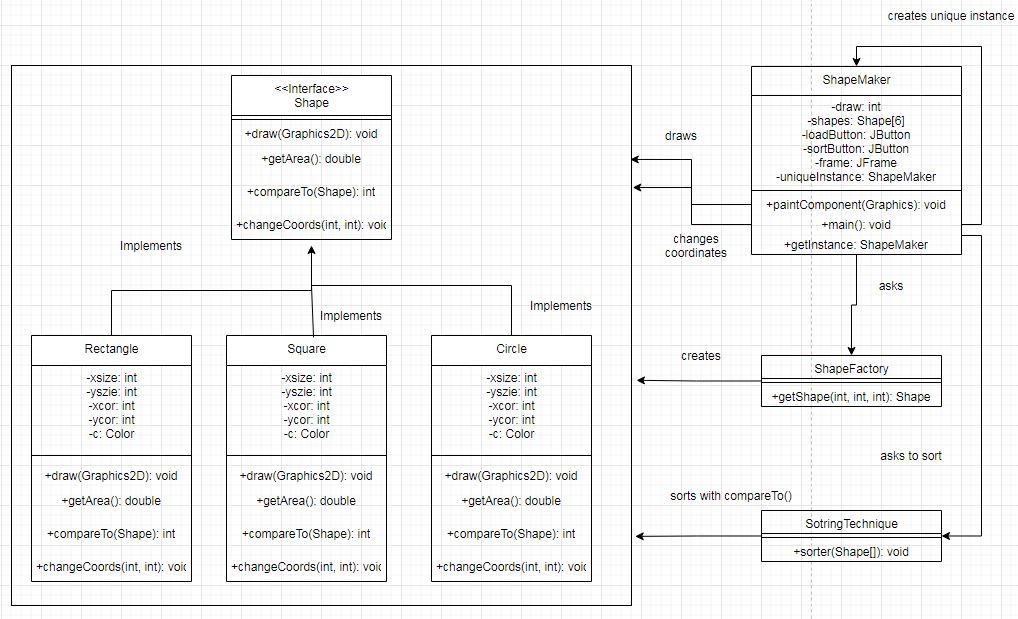
The software project is about creating shapes and sorting them. When the “Load Shapes” button is clicked, six shapes should be created. When the “Sort Shapes” button is clicked, the shapes should be sorted in order of surface area. Shapes can either be rectangle, square, or circle. Shapes should have a colour and must not overlap each other.

The challenges with this project are that I am not familiar with the packages used to create shapes and buttons as I have never used them before.

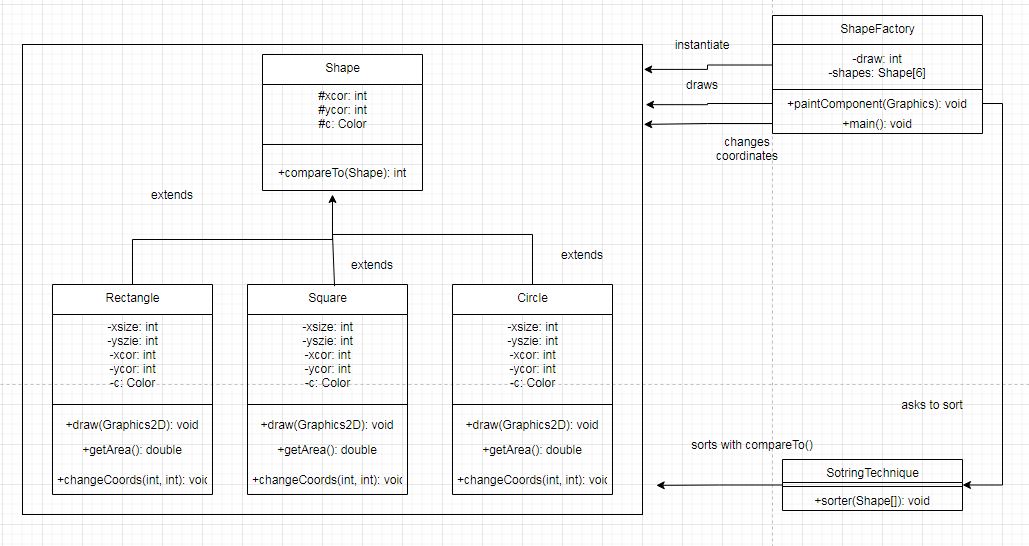
I will use OOD to make shapes, since each shape will be an object. OOD principles I will use include abstraction, polymorphism, and encapsulation; each class will hide unnecessary details from others, the draw and getArea methods work differently for different shapes, and shape objects will be able to compare area with each other. I will use the factory design pattern to make shapes and the singleton pattern to make a unique ShapeMaker instance.

This report will be structured to describe the factory pattern and the sorting technique I used.

Part II



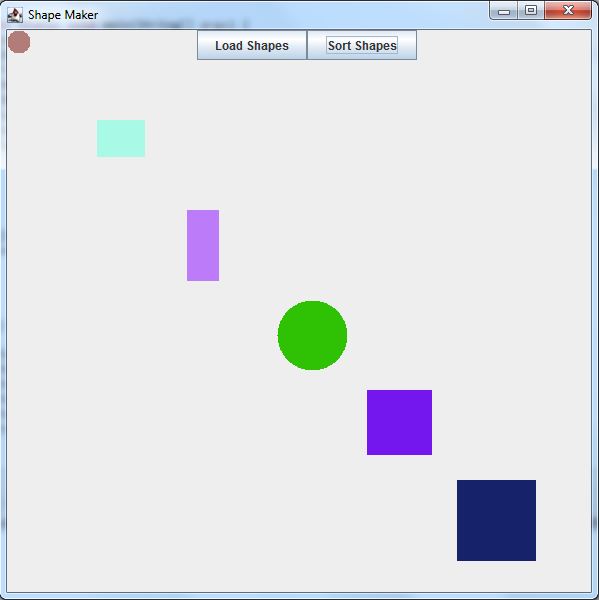
In my design, I used the principle of abstraction; the fields of each shape object are private and can only be used through methods. These methods being draw, getArea, compareTo, and changeCoords. The fields of the ShapeMaker class are also private; the shapes[] array is only given to another class when the SortingTechnique is invoked. I also used the principle of encapsulation, since shape objects can interact with each other via the compareTo method. Finally, I used polymorphism: the draw() method works differently when used on a rectangle or a circle.



The above diagram is my alternate design, where shape is a class instead of an interface. This design is worse than my actual one because each time a shape is drawn, it has to be cast to its specific type (rectangle, square) in order to be drawn. This is undesirable compared to the first design, which uses an interface to negate the need for type casting.

Part III

The algorithm I used was insertion sort. The algorithm checks each element from left to right, and puts it in the position of the first element that is smaller than it and to the left, before shifting the necessary elements. I implemented the first class diagram. The interface shape contains 4 methods which the classes Rectangle, Square, and Circle implement. The main class, ShapeMaker uses ShapeFactory to instantiate the correct type of shape and SortingTechnique to sort an array of them. To make this project, I used Eclipse IDE 2020-12.



Part IV

During the project, what went well was implementing the factory design pattern. A helpful diagram made it easy to code the correct classes and interface. What went wrong was that it took me a while to get the JFrame to work properly. I didn’t realize that after the shapes were sorted, the frame had to be “repainted” in order for the shapes to update. It also took some research to figure out how to make the buttons do something. I had to research “action listeners”. Another setback was the time it took to figure out how to implement the insertion sort algorithm in java. While doing this project, I learned how convenient interfaces are for making methods polymorphic. Three recommendations I would make to ease the project: link how to make action events for buttons, link how to repaint JFrame, and specify which parameters fillRect() are the coordinates and the dimensions so we don’t have to figure out ourselves.