EECS 3311
Lab Project 1 (Lab 3)

Friday, October 8, 2021

Andrew Hocking

215752835

TA: Naeiji Alireza

Part I: Introduction

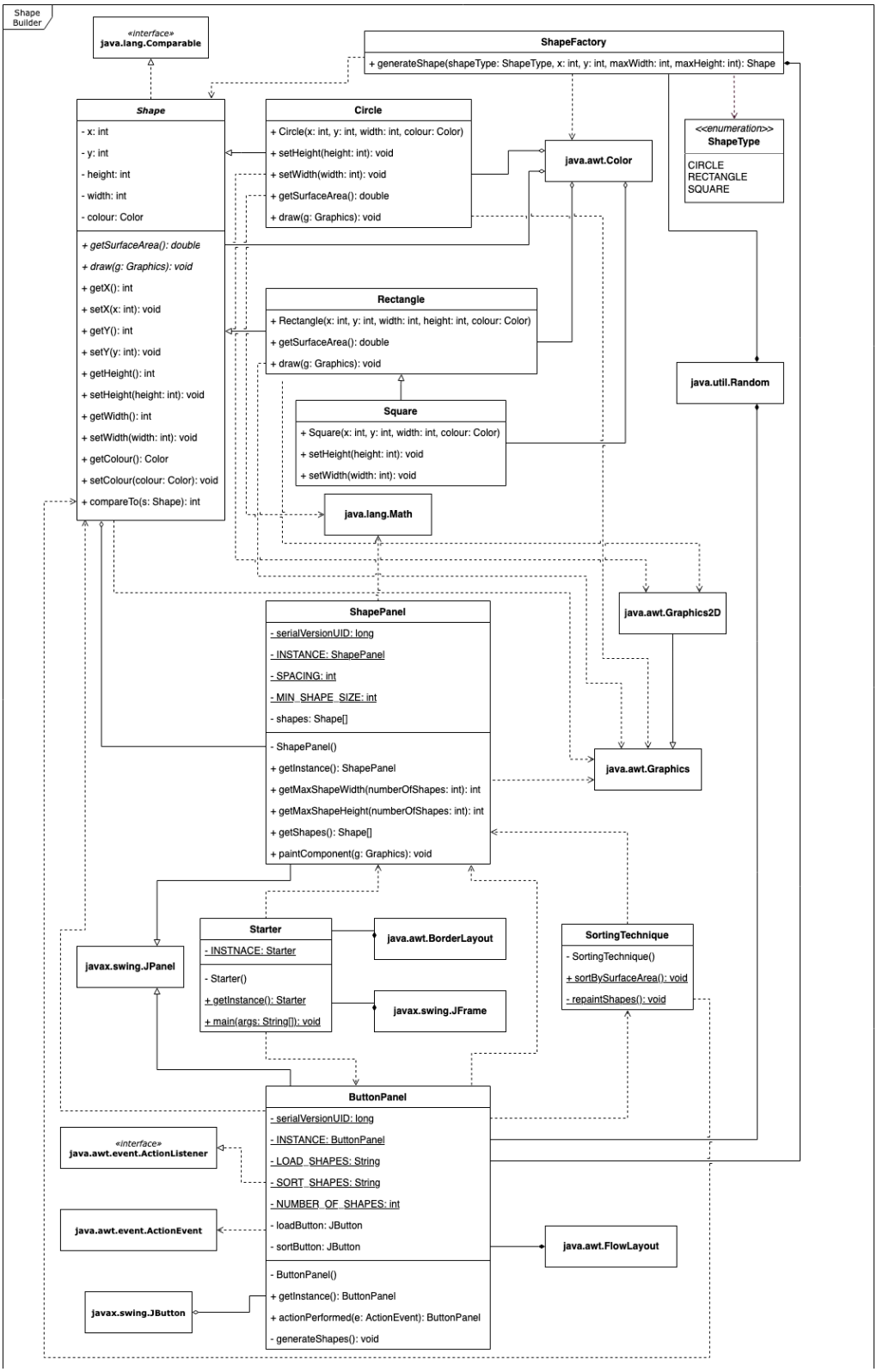
The goal of this project is to give students an idea of how to actually use and implement OOD, OOD principles, and design patterns in their code, by creating a small, simple application that utilizes these concepts.

This may come as a slight challenge, as in the past I have typically spent very little (if any) time planning what my approaches would look like; I usually dive right into solving the problem and sorting out any readability or understandability issues later on.

In this assignment, I will use all four of the main OOD principles as defined in lecture: Abstraction, Encapsulation, Polymorphism, and Inheritance. Abstraction will be used in places such as the ShapeFactory and SortingTechnique classes. Encapsulation will be used in places such as each Shape object's private attributes. Polymorphism will be used to treat all shapes as Shape objects when sorting, etc. regardless of if they are Rectangle, Square, or Circle objects. Finally, Inheritance will be used in a similar manner, where all the methods in the Shape class will be available for its child classes (Rectangle, Square, Circle) to use and/or override. Several different design patterns will be used as well, which will be mentioned in Part II below.

The given assignment is split into four parts. Accordingly, I have split this report into four parts as well. The answers to the questions in the assignment are in roughly the same order in this report as they are in the assignment, but a few may be moved around slightly to allow for more natural sentence structure and logical organization.

Part II: Design of the Solution



Class Diagram 1

ShapePanel, ButtonPanel, and Starter are all “singleton” classes, since they should only ever be used once in this application, so there is no need to create multiples of them. Their single instances can be obtained by using each of the classes’ respective getInstance method.

The abstract Shape class houses private variables for the shape’s coordinates, width, height, and colour, along with getters and setters for each, for proper data encapsulation. It also has an undefined getSurfaceArea method and an undefined draw method, which calculate the surface area of the shape and draw the shape on the screen, respectively. The Shape class conforms to the Comparable interface, so it also has a compareTo class, which compares the shapes based on surface area.

The Rectangle and Circle classes extend from the Shape class. They have definitions for the undefined methods. The Square class extends the Rectangle class. The Square and Circle classes both also override the getters and setters for width and height, since for both of them, the width and height should always remain equal to each other.

The ShapeType enum simply has 3 values: RECTANGLE, SQUARE, and CIRCLE, which obviously correspond to the three defined types of Shapes above.

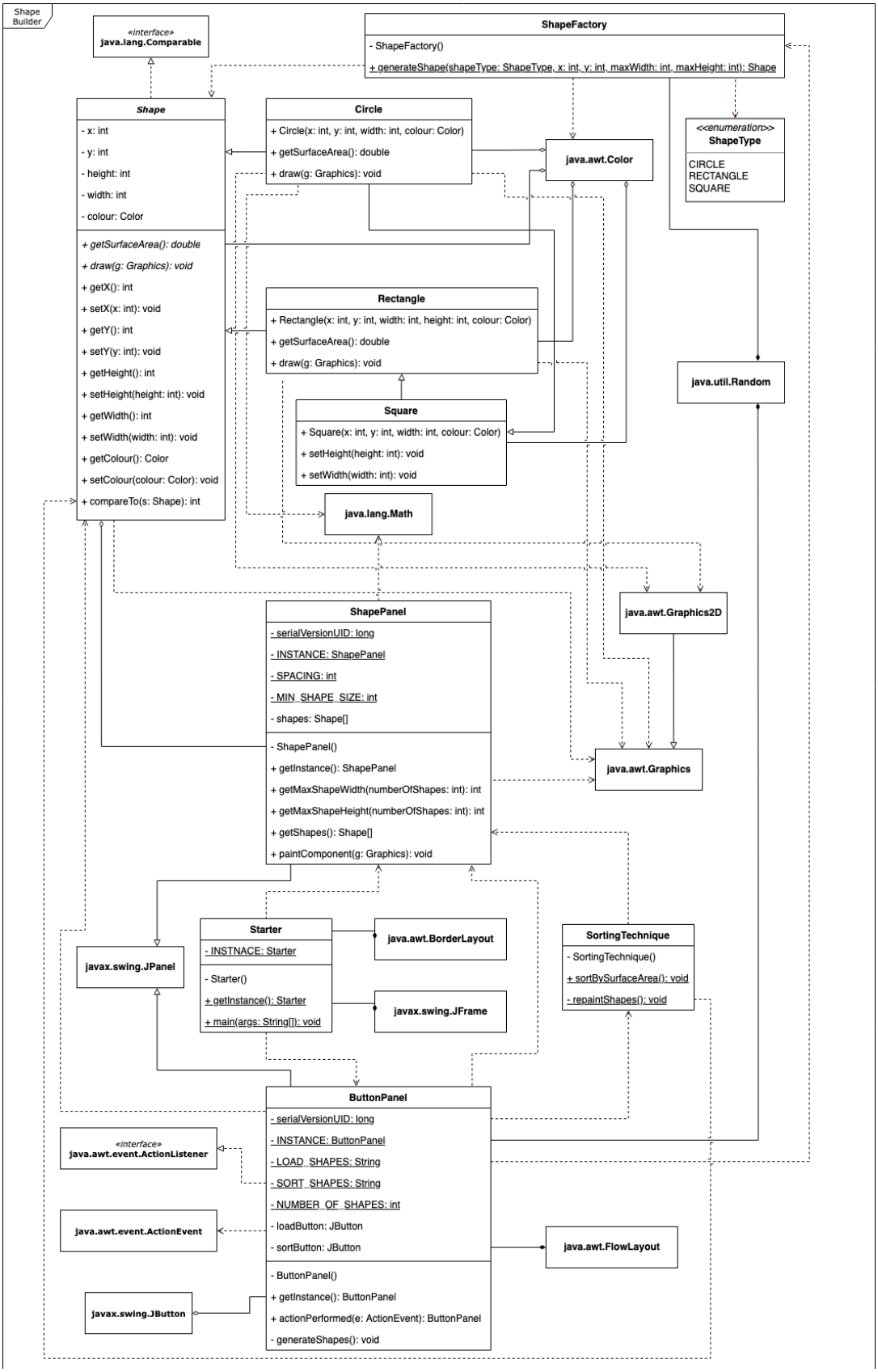
ShapePanel is an extension of JPanel, and it is where the shapes will appear on screen. It holds some public, static constants for the minimum size of and spacing between shapes, as well as a private array of Shape objects, which is accessible by using the getShapes method. The ShapePanel class also has methods to determine the maximum width and height any new shape is allowed to be when they are created, based on how large the ShapePanel is. The class also overrides the paintComponent method, which paints each of the shapes in the shapes array onto the screen.

ButtonPanel is an extension of JPanel, and it houses the two buttons on screen. It uses the “observer” design pattern. In its constructor, it creates the two JButtons and places them at the top of the screen. The overridden actionPerfomed method dictates the two buttons’ actions: the “Load shapes” button will use the generateShapes method to create and draw the shapes, and enable the “Sort shapes” button. The “Sort shapes” button will sort the shapes by surface area using the SortingTechnique class, and disable the “Sort shapes” button. The generateShapes method gets the ShapePanel instance and clears its shapes array. It will then generate a new set of shapes using the encapsulated ShapeFactory instance.

ShapeFactory is a “factory” class with only one method: generateShape. It creates a Shape of the type provided, and returns the new Shape.

SortingTechnique is a “utility” class with one public method, sortBySurfaceArea, and one private method, repaintShapes. These do as one would expect.

Finally (ironically), Starter is a “bridge” class that runs the application. In its main method, it creates a JFrame, places the ButtonPanel and ShapePanel inside of it, and displays the window.



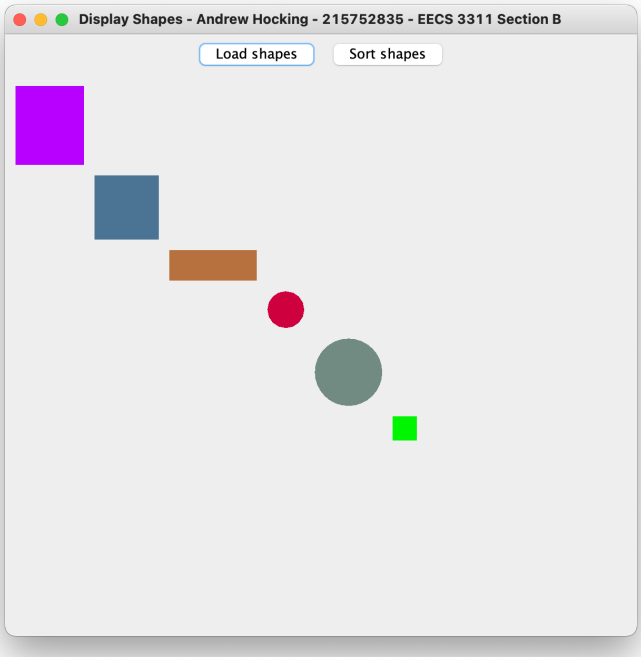
Class Diagram 2

My second diagram is very similar to my first, as the basis of what this application needs to do remains the same. However, the major changes in the second diagram are that:

- ShapeFactory is now a (non-instantiable) utility class with just the generateShape method.
- Circle extends Square, since its overridden getters and setters are identical.
- ShapePanel and ButtonPanel are no longer singletons.

This diagram was actually what my first design of the app looked like, before I decided to change how I was doing things and created the other diagram to replace it. I believe the first diagram above is better than this one, since it better adheres to established design patterns, and makes more logical sense (e.g. Circle should not extend Square, even if it creates some code duplication, because those are very different shapes). It also does not require passing around a single ShapePanel object to a bunch of different classes and methods as a parameter.

Part III: Implementation of the Solution



Snapshot of the application running

I used Class Diagram 1 for the implementation of my application, for the reasons stated above in Part II. I initially attempted to implement Class Diagram 2, but I found it was clunky and was not in the spirit of the assignment.

The ins and outs of each of the classes are also mostly stated above in Part II, with the exception of the intricacies of the sorting algorithm, which uses Bubble Sort to order the shapes by surface area. Bubble Sort compares two adjacent elements, and if they are in the incorrect order, it will swap them. IT does this for each adjacent pair in the array until it reaches the end. At that point the final element will be in the correct position, and it starts again from the beginning, continuing as it did in the first pass until it reaches the element before the end of the array, which will end u in its final position. This repeats over and over until every element is eventually in its rightful place. This algorithm is stable, since it does not swap any elements if they are equal.

The Java files are all in the “shapebuilder” package (located at /src/shapebuilder, on the GitHub repo). The application is run by executing the main method of the Starter class. Please see the “Shapes App Demo.mov” video file for a demonstration of the application. I used Eclipse, Version 2019-12 (4.14.0) to write and run the code, using JDK 1.8 (Java 8), on macOS 11.2.

Part IV: Conclusion

In terms of what went well, the coding part was quite easy. The application itself is not too complicated, and my previous experience with Java Swing UI programming made it fairly simple. Demoing it was also quite simple, and only took one attempt.

Nothing really went wrong with this project, but actually paying attention to design patterns etc. made this project take longer than previous coding projects. I usually just dive right into the coding, but this project required forethought. I tried jumping in with my Class Diagram 2 above, without giving it too much thought, and as I said, I ended up changing a lot in the end to get the application to line up more with what the assignment entailed.

While doing this assignment, I learned that I naturally use some established design patterns already, but I do not follow their guidelines strictly on my own. I also learned about some new design patterns that I ended up using in this project, such as the “observer” and “bridge” patterns.

If I had to give advice to future students who will do this assignment, I would recommend that they:

1. Create a basic UML class diagram first, then start working on the project, and then go back and amend your class diagram for any inaccuracies. I tried at first to just go for a UML diagram right off the bat, but it led to a lot of overthinking and confusion about what needed to be connected what, and how they should connect, and what Java-included classes I would be using, and it made it much harder for me to actually do. I eventually just moved to a basic diagram, and built it up as I went, which made it more understandable overall.
2. Find resources for Java Swing UI. I am familiar with it now, but when I first started working with it a couple years ago, I found it very difficult. Utilize any included example code, and if that does not help enough, find a good YouTube tutorial for how to use it.
3. Start early! The coding part of this assignment went by fairly quickly, probably 3 or 4 hours. But the rest of the assignment took a lot longer for me to complete. This is not a do-it-the-night-before assignment, and I say that as someone who has done 99% of my assignments the night before. I am glad I gave myself a head start on this one.