EECS 3311 B - Lab 03
Design Report

Amir Mohamad - 217 387 762

TA - Alireza Naeiji

Oct 10, 2021

Contents

1 Introduction

SHAPESORT is a Java program that simply sorts a set of shapes based on their surface area using concepts of Object Oriented Design (OOD) such as polymorphism, inheritence, encapsulation, and abstraction. During this project, I didn't really face many challanges since I planned the project well and had a solid OOD and requirements foundation before implementation. I first created use cases for SHAPESORT and defined a domain model on pencil and paper. Then finally, from my analysis, I designed the specific classes and relationships. The only issue that I faced was with using the `java.awt` package when drawing the shapes onto a `JPanel`. This was easily overcome in a few minutes by reading the Java API documentation. SHAPESORT uses several design patterns to achieve certain functionality and create an efficient design. For example, I use the factory and singleton design patterns to help abstract drawing the shapes onto a `JPanel`. The purpose of using these design patterns is to make sure I could have maintainability, reusability, and efficiency with the design of project. To help organize the project and create separation of concerns, I used the Model View Presenter (MVP) architecture to implement the back end and front end of SHAPESORT. In this design document, we will cover the design of this system with resect to the front end and the backend. We will overview the general design and then analyze the specific implementation.

2 System Design

2.1 Overview

SHAPESORT's design is structured using the MVP architecture. The system is divided into several packages to implement this architecture. This way, we can maintain a maintainable design while incorporating our OOD principles. The system has 3 main packages: `shapeSort.model`, `shapeSort.gui`, `shapeSort.util`. The `shapeSort.model` package is used to contain the backend model used in SHAPESORT. The `shapeSort.gui` package is used to contain all the UI related code. Finally, `shapeSort.util` is used to contain all the utility classes used by SHAPESORT. Using these packages we can manage a cross-package dependencies easily. The main components of the system across all packages are `ShapeFactory`, `Shape`, `View`, `Presenter`, and `SortingTechnique`. The other componenets are left out of the general system design because they just supplement the general relationship between the main components.

2.1.1 UML notation

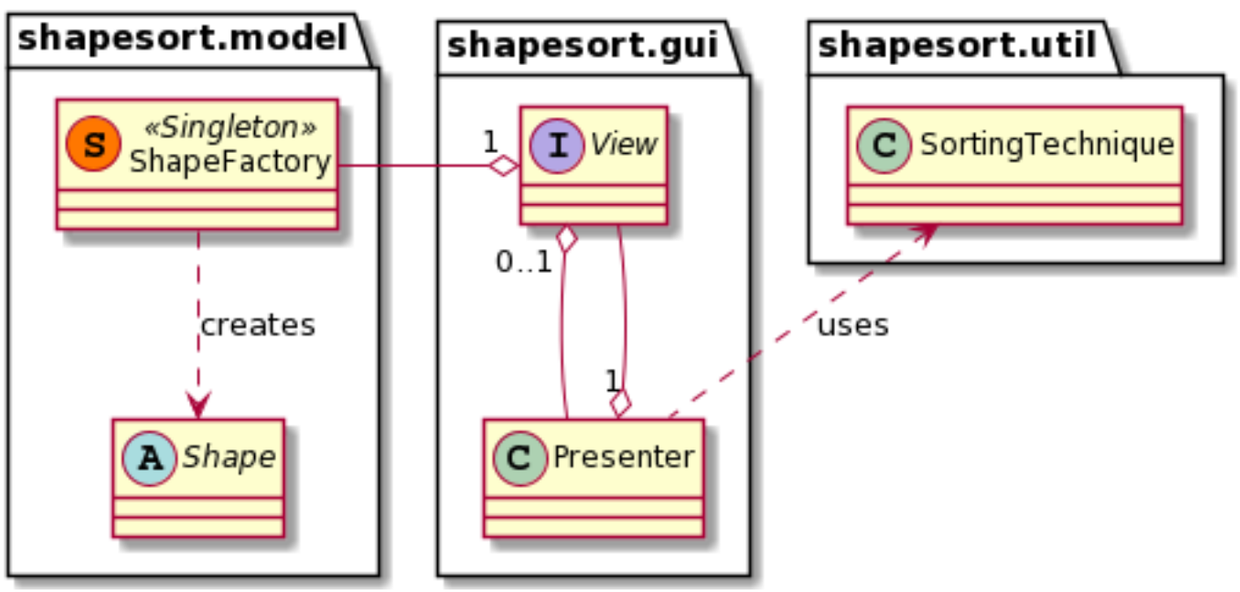

Figure 1: The general system design of SHAPESORT

3 Front end

3.1 Overview

The front end uses a modular approach to help create different types of views. We designed `SHAPESORT` to be easily extendable in its ability to add new views. It's as simple as adding a new `ViewType` and the corresponding view which extends from `AbstractView`, which itself implements a `View`. For example, in `SHAPESORT` we have just one view which is `MainView`. This main view has a `Canvas` which extends `JPanel` to achieve the functionality to draw shapes. The implementation is made much simpler and organized due to the MVP architecture. It created separation between the different components of `SHAPESORT`'s system.

3.2 Class Diagram

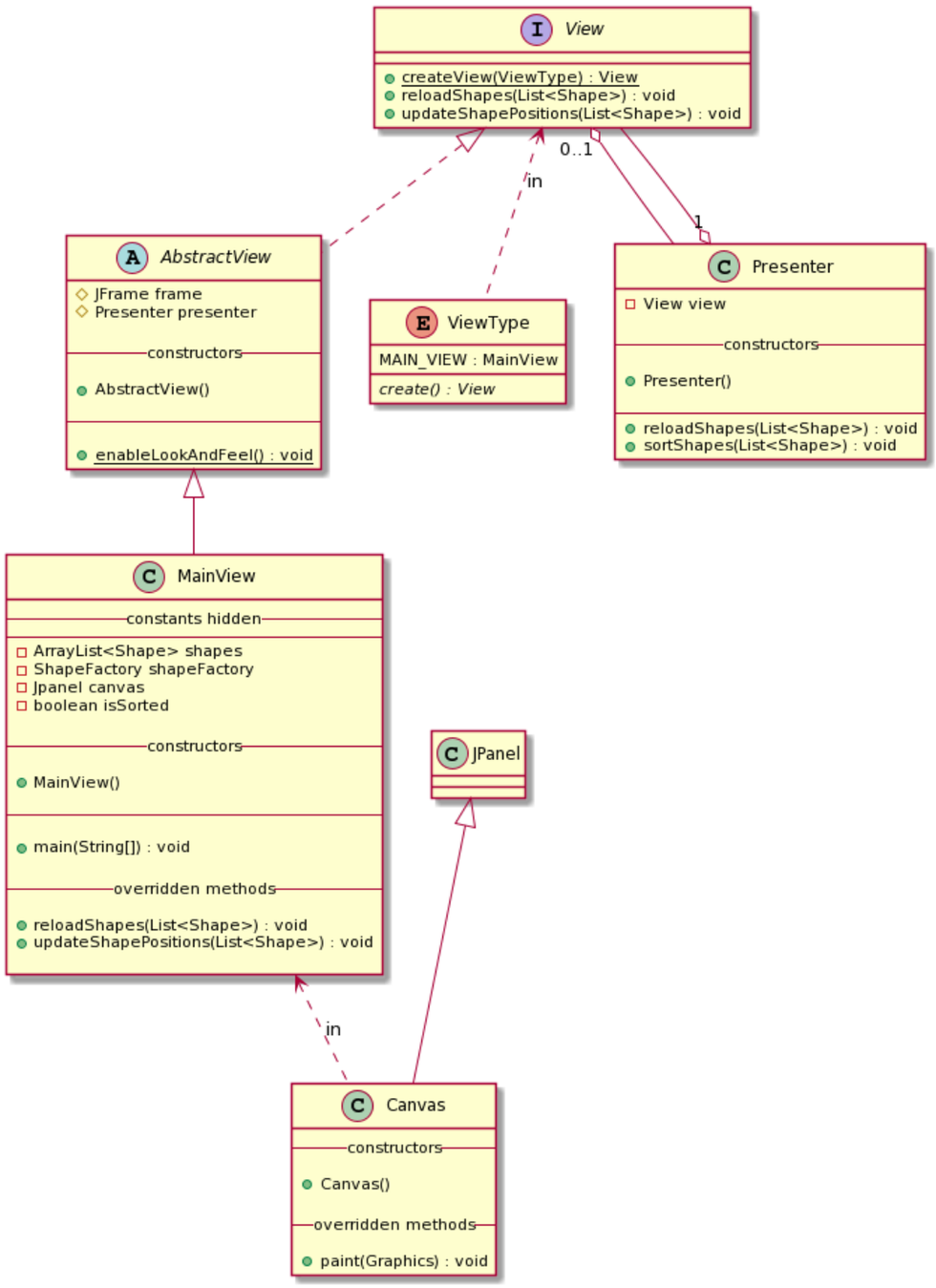

Figure 2: The class diagram of `SHAPESORT`'s front end model

3.3 View

The `View` is a template for `SHAPESORT`'s core functionality (sorting and drawing shapes). We can create various types of view that we wish through `View.create(ViewType)`, where `ViewType` is an enum with all of `SHAPESORT`'s views.

3.4 AbstractView

The `AbstractView` is an abstraction of a view in `SHAPESORT`. It contains the `JFrame` of the respective view and a Presenter that links the front end with the back end model. `SHAPESORT` has a dependency on the Flat Look and Feel. The abstract view enables this look and feel to create a unified theme across all views.

3.5 Presenter

The `Presenter` is the link between the back end model and the view. For example, the `Presenter` is responsible for reloading the shapes present in a view and sorting the shapes via `SortingTechnique`.

3.6 MainView

The `MainView` is the main and currently only view in `SHAPESORT`. This view is simple. It has two buttons; A button to load shapes and a button to sort the shapes. Since `MainView` is the main entry point to `SHAPESORT`, we have a main method which displays the frame once the view is finished configuring its styles and looks.

3.7 ViewType

The `ViewType` is simply an `Enum` which lets use define and create multiple views as we want. For example, this program can be extended to have a view where the shapes are displayed and sorted differently. This opens up various possibilities.

3.8 Canvas

The `Canvas` is just an extention of a `JPanel`. Its only purpose is for shapes to be drawn onto it. This is done by overridding the `paint(Graphics)` method and accordingly drawing the shapes with their position, and other properties.

4 Back end

4.1 Overview

The back end uses all the OOD principles such as polymorphism, inheritence, encapsulation, and abstraction. Inheritence is used to enable `Shape` to take many different forms (ie. circle, rectangle, etc.). This helps solidify the principle of polymorphism. Each type of shape subclass uses encapsulation to help protect the subclass's `Object` properties. This enables that shapes can't be mutable without the develoepr explicitly modifiying the object. Of course, we use abstraction to help hide the specific implementation details when we use the shapes and the different components of `SHAPESORT`. For example, when we use the sorting method from `SortingTechnique` we can hide the specific deatils on how we sort the shapes. The developer can just use it knowing we will get the correct functionality. In addition, the two main design patterns used in `SHAPESORT` are factory and singleton patterns combined into the `ShapeFactory` class.

4.2 Class Diagram

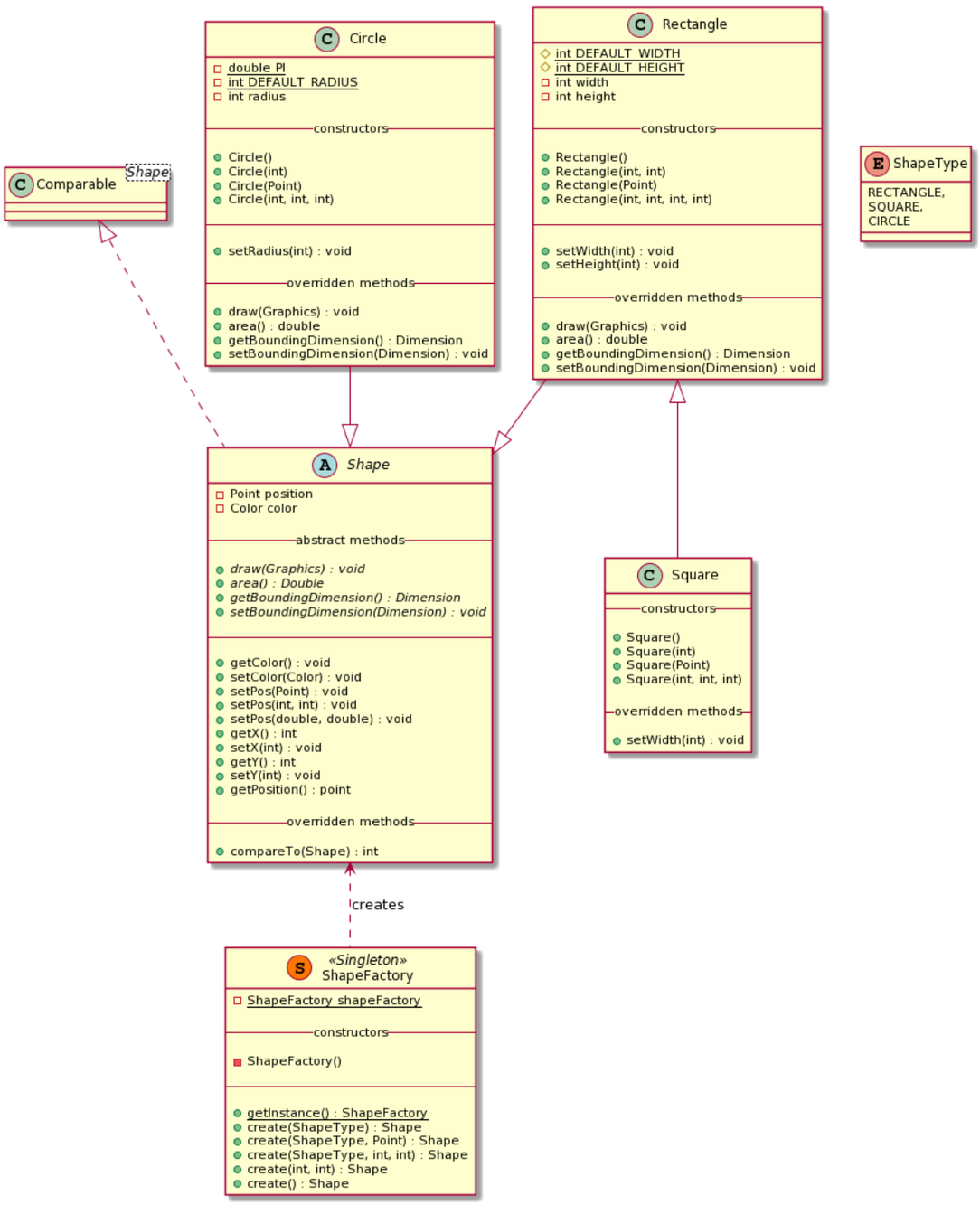

Figure 3: The class diagram of `SHAPESORT`'s shape model

4.3 Shape

The `Shape` is an abstract representation of a shape. It only has two pecies of information available to its children: the position of a shape and its color. We use the `java.awt.Point` from the Java API to keep track of a shapes position and `java.awt.Color` for the shapes color respectively. We also use encapsulation to help us provide an API to modify any shape. Shapes also contain two behaviours in our design. The ability to compute their own area using the `area()` method and to draw themselves based on the `draw()` method which takes the graphics context of some component. Another important thing to note is that `Shape` implements `Comparable` to help sort shapes based on their surface area.

4.4 Rectangle

The `Rectangle` extends `Shape` so it has all the properties of a shape along with properties of a rectangle. For example, the width and height. Same as the `Shape` class we use encapsulation to help modify the two properties of a rectangle.

4.5 Square

The `Square` extends `Rectangle` so it has all the properties of a rectangle along with properties of a square. It's important to know that during our analysis we noted a squrae is just a subset of rectangles. The `Square` is a simple calss to modify. We set it's properties using a `super()` call to the `Rectangle` class. We make sure that in the `Rectangle` class we make the `setHeight()` method to private. To follow up we override `Rectangle`'s `setWidth()` to set the only property a square has.

4.6 Circle

The `Circle` extends `Shape` so it has all the properties of a shape along with properties of a circle. For example, the radius. Same as the `Shape` class we use encapsulation to help modify the two properties of a circle.

4.7 ShapeType

The `ShapeType` is an `Enum` which simply has all the supported shapes in `SHAPESORT`. This makes it much more convienent to implement functionality for other types of shapes (ie. triangle).

4.8 ShapeFactory

The `ShapeFactory` is a singleton class that uses the facotry design pattern. In `SHAPESORT` we we only have one instance of a `ShapeFactory` to help create different types of shapes based on `ShapeType`.

4.9 Utility

The following are utility classes used to compute a single function in their respective area. These classes don't specifically have any OOD relationships besides regular dependency relationships.

4.9.1 ColorUtility

This utility class's main function is to generate random `java.awt.Color` instances for our shapes. It has potential to be extended to provide further functionality.

4.9.2 SwingUtility

This utility class's main function is to help modify swing components in an effective and efficient way. This helps to reduce code duplication, maintainability, and productivity.

4.9.3 SortingTechnique

This utility class's sole function is to use various methods to sort a list of shapes based on different properties. In this project we currently only sort based on the area by default. We can provide a custom comparator to define the desired sorting property. For now by calling `sort(List<Shape>)` this class will use Quicksort to sort the shapes based on a default or custom comparator.

5 Alternative Design

This alternative design has a few changes. Firstly, the shape model is changed by having `Square` extend `Shape` just like all the other shapes, with its own properties defined independent of `Rectangle`. This is not a better option since it increases code duplication and doesn't utilize the abstraction of a square. Secondly, the `ShapeFactory` class is no longer a singleton class. The idea beheind this is to make sure we can have mutliple factories for multiple views. In this sense its is better for the design. Lastly, the `MainView` class is turned into a singleton since there should only be one instance for the main view. This is much better in terms of design since we only need one instance of each view. To make this design better than the first, we will implement all the changes mentioned above except the first change (the `Square` class). I implemented the first class diagram for this project.

5.1 Class Diagram

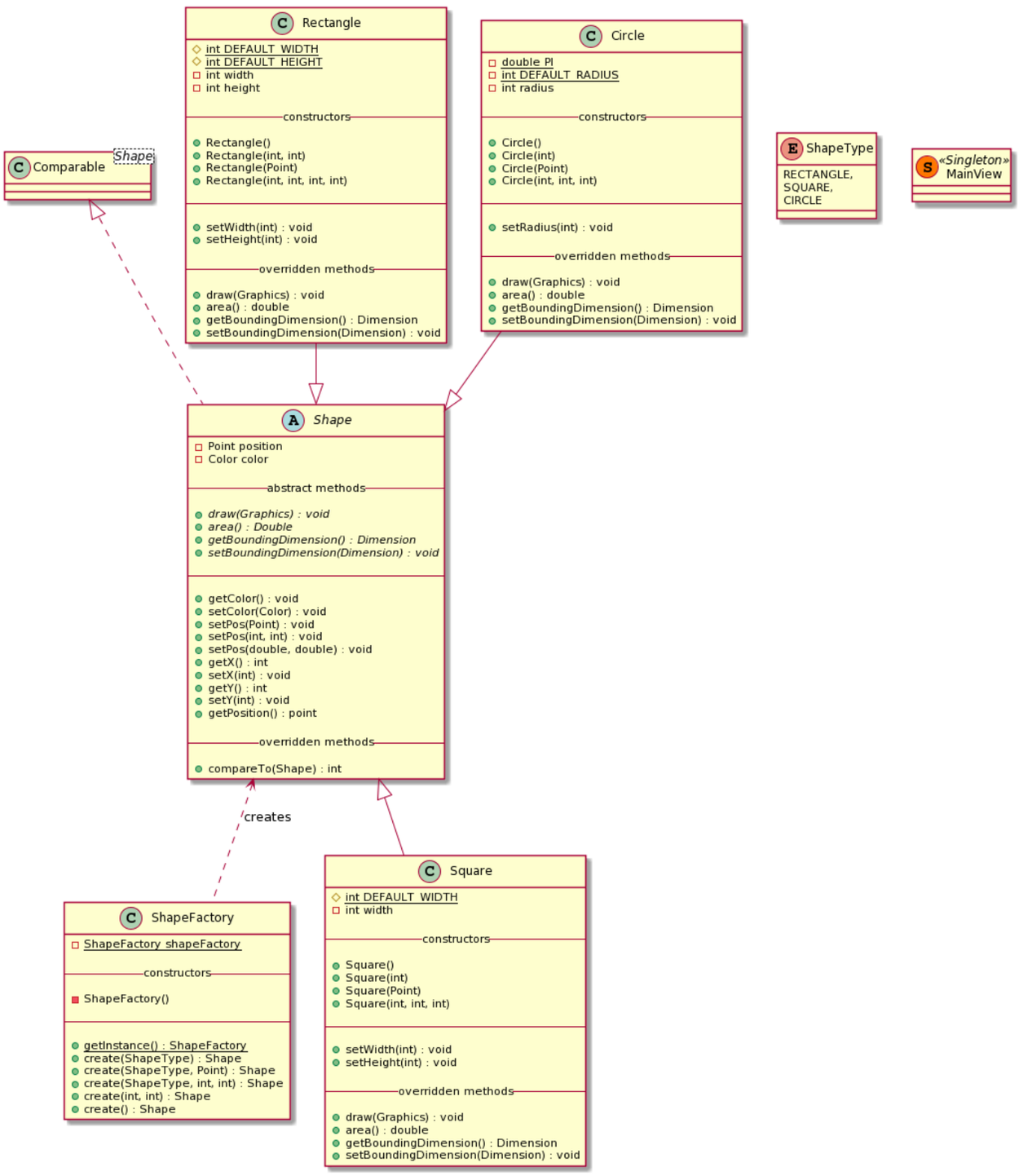

Figure 4: The alternate class diagram of `SHAPESORT`'s shape model + `MainView`

6 Implementation

6.1 Sorting Algorithm

The algorithm used to sort the shapes based on the surface area is Quicksort. The key to help implement this algorithm is to take advantage of `Shape` being comparable. We use the `compareTo()` method to help define order when partitioning in the algorithm. Another crutial step is to make sure that we swap the shapes positions when we swap the `Shape` objcts within the internal array.

6.1.1 Pseudocode Code

```
1  fun quickSort(shapes, int left, int right)
2      if (left >= right)
3          return
4
5      Shape pivot = shapes[(left + right) / 2]
6      int index = partition(shapes, left, right, pivot)
7      quickSort(shapes, left, index - 1)
8      quickSort(shapes, index, right)
9
10 fun partition(shapes, int left, int right, Shape pivot)
11     while (left <= right)
12         while (shapes[left] < pivot)
13             left++
14
15         while (shapes[right] > pivot)
16             right--
17
18         if (left <= right)
19             if (shapes[right].area() != shapes[left].area())
20                 swap(left, right, shapes)
21             left++
22             right--
23
24     return left
25
26 fun swap(int left, int right, shapes)
27     Shape leftShape = (Shape) shapes[left]
28     Shape rightShape = (Shape) shapes[right]
29     // swap shape positions also
30
31     shapes[left] = shapes[right]
32     shapes[right] = leftShape
```

6.2 Class Implementation

`SHAPESORT`'s first step in creation after the analysis/requirements and design steps was initializing a java gradle project using `gradle init` task. After this I began the implementation of each class in IntelliJ usin their rich set of features available for java. I created the packages to implement the base MVP architecture. After the inital setup was done I began working on specific class implementation using the iterative scrum wokrlfow. Of course I was alone so I was all the roles in the process. After implementation I tested everything and repeated the process until the final submission. I compiled all the java classes using java 16 with the help of IntelliJ and the `gradle run` task. In regards to gradle, I used it to manage the look and feel dependencies, create coverage reports for future use, and and easy way to run the application using the application `gradle run` task. Since everything was designed before implementation it was easy to implement the project using these tools. Ofcourse in the iterative procces requirements changed on each iteration I did to improve the design of the original implementation.

6.3 Execution

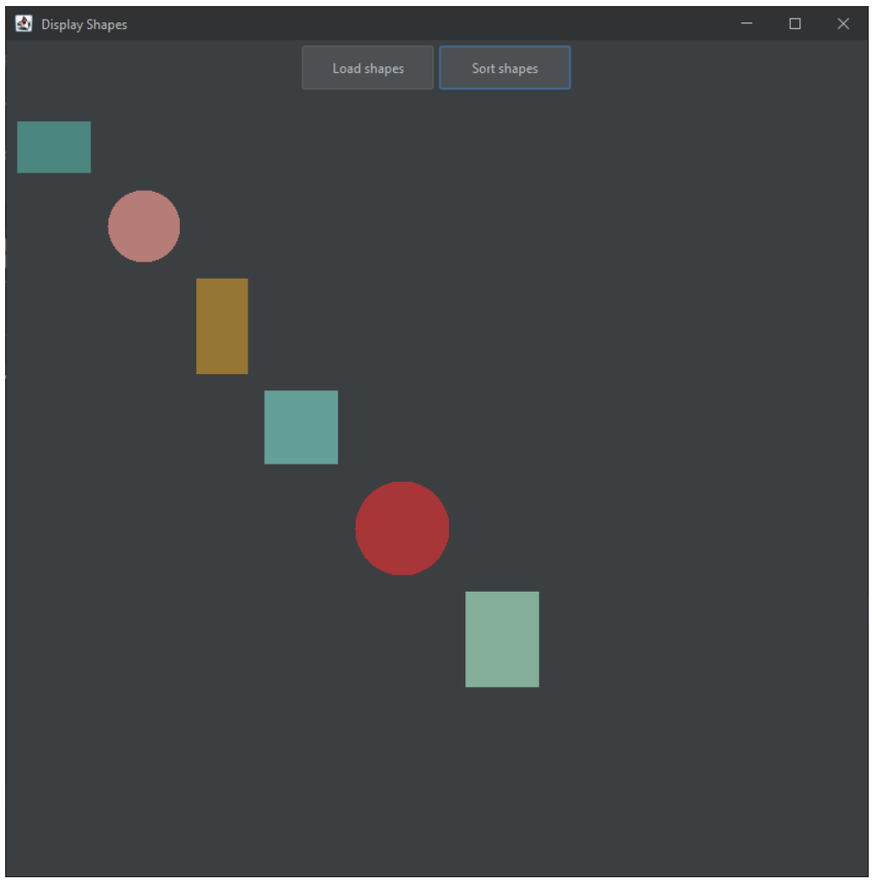

Figure 5: `SHAPESORT`'s main interface

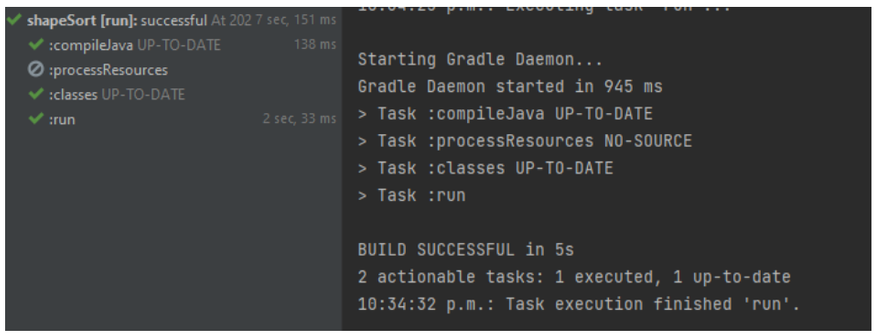

Figure 6: The success execution status from the `gradle` run task

7 Tools

7.1 Source Code

IDE: IntelliJ IDEA 2021.2.1, to implement Java classes
Documentation: vscode, LaTeX, for documentation (dessign report, etc.)

7.2 Build

Build: Gradle 7.2 (to manage dependencies and build)
Java Version: openjdk version "16.0.1" 2021-04-20

8 Conclusion

Overall, This project was successful because I followed through with the four step OO design/analysis process to complete `SHAPESORT`. The only thing that might have been percived as going wrong initially is not understanding how to use some classes from the `java.awt` package. This was a simple fix with a few minutes of reading documentation. During this project I learned how to implement a project while actively thinking of design patterns in the process. The recommendations I would give someone to help complete this project effectively is to learn how to read Java's API documentation. This is curutial to help understand how to use certain classes to achieve the desired functionality. Also planning out the project helps to keep organized at each stage of development.