Dongwon Lee 215805260

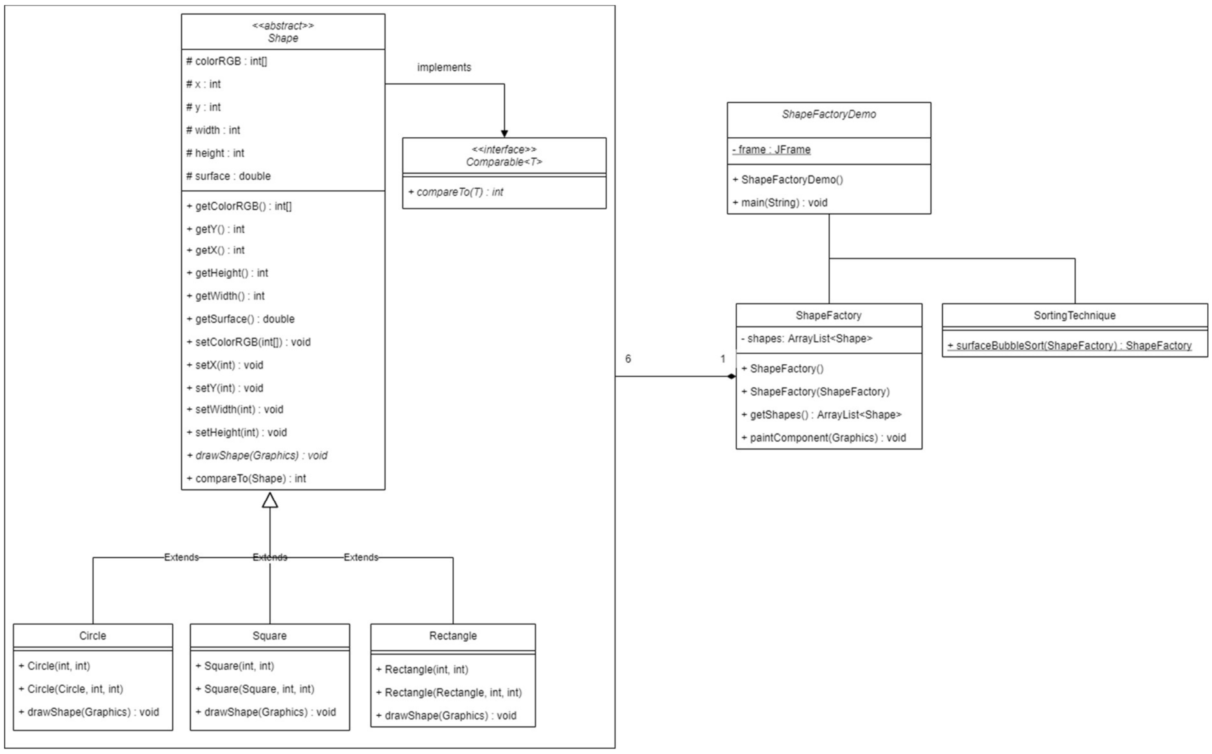
TA Alireza Naeiji

EECS 3311 section B

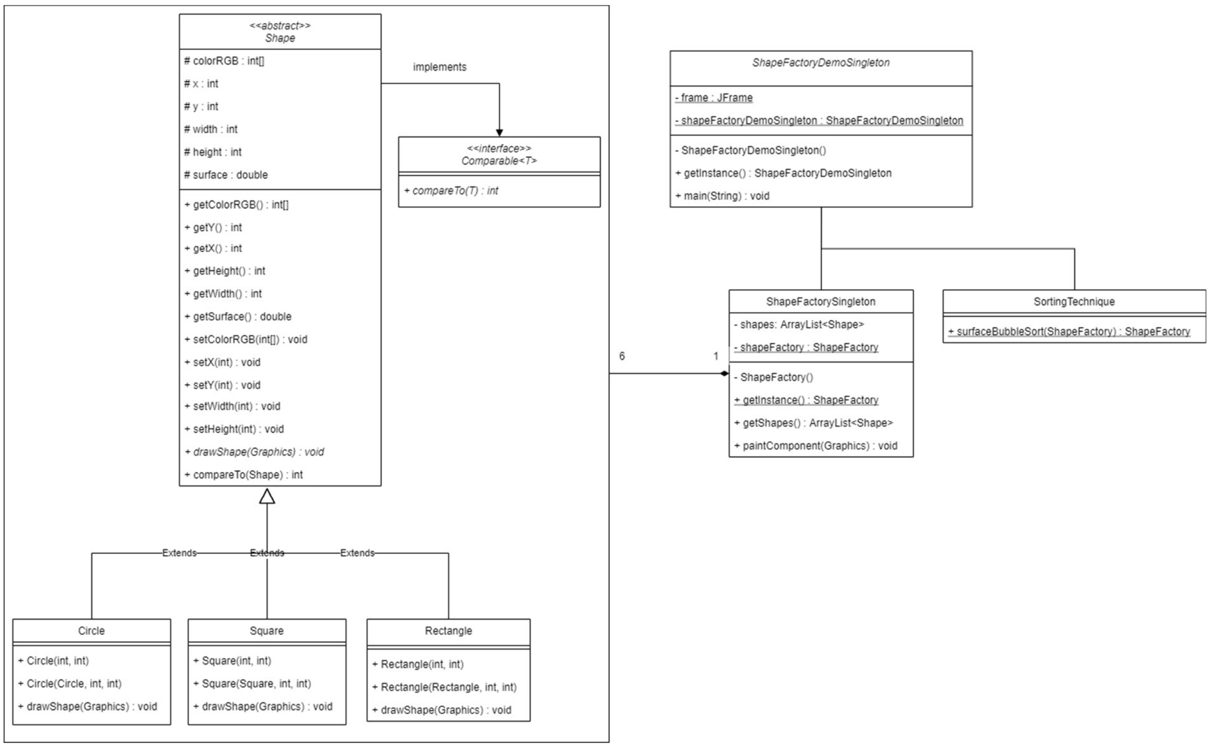
10 October 2021

Shapes Creation and Sorting Program by Object-Oriented Design

This project is to create a program that uses object-oriented design to create 6 random shapes of the square, rectangle, and circle, sorts them according to the area of the surface, and displays those shapes as a GUI in Java. The challenge in this project is first to deal with 6 shapes of the square, rectangle, and circle through OO design, secondly to implement a sorting function, and thirdly to represent all of these in a GUI. The OO designs to be used in this project are a factory pattern design and a Singleton pattern design and OO design principles to be used are abstraction, capsulation, inheritance, and polymorphism. With Shape class as an abstract class, Circle, Square, and Rectangle classes as child classes that inherit and implement it, ShapeFactory class that handles 6 Shape classes, and ShapeFactoryDemo class that expresses it with GUI, factory pattern is the most efficient OO design in this project. And with singleton pattern, resource usage can be reduced by maintaining only one instance each. This project report consists of the design of the solution, the implementation of the solution, and the conclusion. In the design part, two UML class diagrams of two different design patterns will be covered, and in the implementation part, how and how to implement it in code will be covered. The conclusion part will deal with the review after the project has been completed.

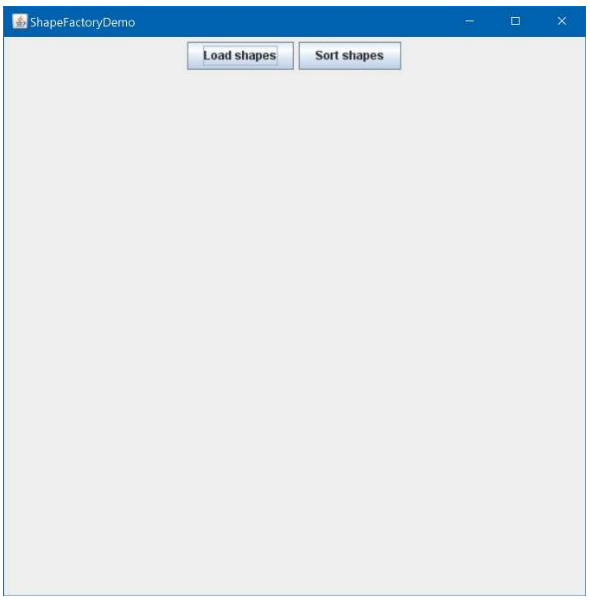


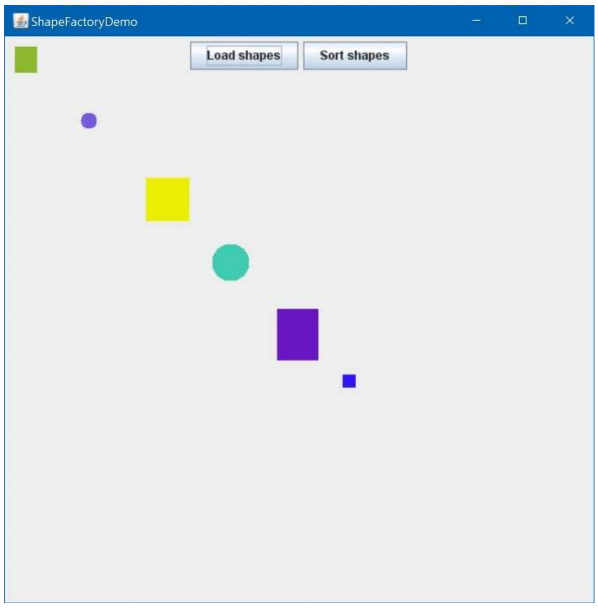
As a design of the solution, I made a UML class diagram with a factory design. Shape, Circle, Square, and Rectangle are parent-child relationships, and ShapeFactory is a factory class that creates and handles 6 Shapes. And SortingTechnique is a class that sorts 6 Shapes created in ShapeFactory according to the surface, and ShapeFactoryDemo is a class that expresses functions in GUI. In this class diagram, four OO design principles are applied. As an abstraction principle, Circle, Square, and Rectangle are abstracted into Shape. As an encapsulation principle, the fields of each class are made private and only the essentials are known by implementing getter methods that can obtain them. As a principle of inheritance, with Shape as the parent class and Circle, Square, and Rectangle as child classes, common fields or methods are put in Shape, and individual ones are implemented in child classes to achieve inheritance relationship. Polymorphism is achieved by making the drawShape method in the Shape class as an abstract class and in the Circle, Square, and Rectangle classes, drawShape method is implemented to call methods that draw their own shape. I also made a UML class diagram with the other OO design. It is a class diagram that uses the Singleton pattern. Basically, based on the factory pattern class diagram above, ShapeFactroy class and ShapeFactoryDemo class are made into Singleton class so that only one instance can exist each. And since the Singleton class cannot use a copy constructor, that part was also modified. The advantage of this second class diagram over the first class diagram is that resource usage can be reduced by maintaining only one instance of ShapeFactory and ShapeFactoryDemo.



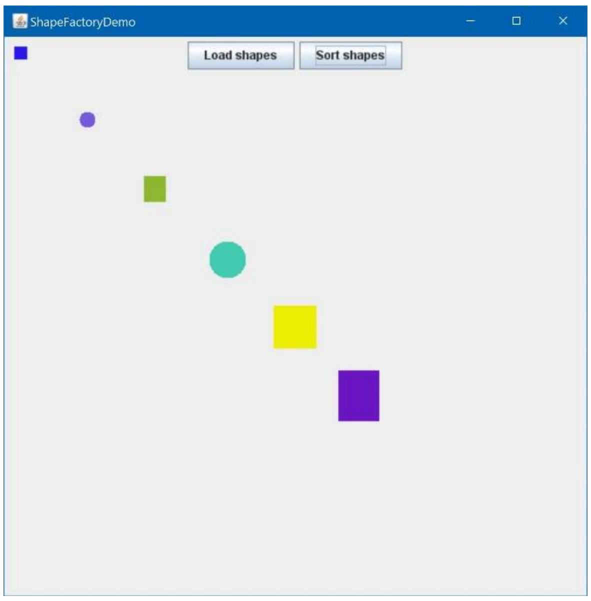
<<abstract>>
Shape
# colorRGB : int[]
# x : int
# y : int
# width : int
# height : int
# surface : double
+ getColorRGB() : int[]
+ getY() : int
+ getX() : int
+ getHeight() : int
+ getWidth() : int
+ getSurface() : double
+ setColorRGB(int[]) : void
+ setX(int) : void
+ setY(int) : void
+ setWidth(int) : void
+ setHeight(int) : void
+ drawShape(Graphics) : void
+ compareTo(Shape) : int
implements
<<interface>>
Comparable<T>
+ compareTo(T) : int
Extends
Extends
Extends
Circle
+ Circle(int, int)
+ Circle(Circle, int, int)
+ drawShape(Graphics) : void
Square
+ Square(int, int)
+ Square(Square, int, int)
+ drawShape(Graphics) : void
Rectangle
+ Rectangle(int, int)
+ Rectangle(Rectangle, int, int)
+ drawShape(Graphics) : void
6
1
ShapeFactoryDemoSingleton
- frame : JFrame
- shapeFactoryDemoSingleton : ShapeFactoryDemoSingleton
- ShapeFactoryDemoSingleton()
+ getInstance() : ShapeFactoryDemoSingleton
+ main(String) : void
ShapeFactorySingleton
- shapes: ArrayList<Shape>
- shapeFactory : ShapeFactory
- ShapeFactory()
+ getInstance() : ShapeFactory
+ getShapes() : ArrayList<Shape>
+ paintComponent(Graphics) : void
SortingTechnique
+ surfaceBubbleSort(ShapeFactory) : ShapeFactory

The sorting algorithm used in this project is bubble sort. It uses bubble sort for loop to repeatedly check 2 adjacent shapes and uses the compareTo method of the Shape class to compare their surfaces. If they need to be swapped, Collections.swap method is used to swap them. The time complexity of bubble sort is O(n2), which is slow, but in this project, only 6 shapes were used, so bubble sort was enough. The code of the project was implemented with the first class diagram, that is, the factory pattern class diagram. Overall, Shape is an abstract class, Circle, Square, and Rectangle classes inherit and implement it, ShapeFactory handles 6 Shape instances, and GUI is implemented with ShapeFactoryDemo. The Shape class is an abstract class that implements the Comparable<Shape> interface. It includes fields such as colorRGB, x, y, width, height, and surface, and methods such as getter/setter, compareTo, and drawShape that are commonly used by Circle, Square, and Rectangle. Getter/setter and compareTo are implemented because the three child classes have a common implementation, and drawShape is an abstract method, so Circle, Square, and Rectangle can implement their own shape. Circle, Square, and Rectangle classes receive x and y as arguments in common to their constructors, assign them to their own x and y, generate color, height, and width randomly, and calculate the surface accordingly. Also, a copy constructor for the Sorting part, receives the instance of each class, x, and y as arguments, and copies the field of the instance, and assigns the x, y to their own. Also, the drawShape method draws each shape with the color, x, y, height, and width randomly created in the constructor on the Graphics instance that is received as an argument. The Comparable class is an interface that has a compareTo method. ShapeFactory class manages 6 shapes and is a Factory class that inherits JPanel. It has ArrayList<Shapes> as a field, and in the constructor, 6 of Circle, Square, and Rectangle are randomly created and added to the ArrayList by increasing x and y as arguments. The copy constructor for the Sort part takes an instance of ShapeFactory as an argument and recreates the sorted ShapeFactory. As a class that inherits from JPanel, it overrides the paintComponent method and calls drawShape in the Shape class for every 6 shapes. The SortingTechnique class is a class that sorts shapes according to their surface. It defines the surfaceBubbleSort method as static so that it can be called from anywhere, and receives a ShapeFactory instance as an argument, and sorts the shapes in it in ascending order according to the surface using bubble sort and Shape's compareTo method and return the instance. ShapeFactoryDemo class is a class that implements GUI and has a Load shapes button and a sort shapes button. These buttons include ActionListners that randomly create 6 shapes using ShapeFactory, sort them, and then draw them again. In this project, Eclipse 2020-06 (4.16.0) version and JDK version 1.8 were used. Below are the snapshots of the result of running the code.





Initial screen When 'Load shapes' button is pressed



When 'Sort shapes' button is pressed

As a comment, after pressing the button once, the two buttons become invisible on the screen, which can be made visible again by hovering the mouse over the button. Other than that, if the Load shapes button is pressed, 6 shapes appear randomly, and if the button is pressed again, the previous shapes are deleted and new shapes appear. Also, if the Sort shapes button is pressed, the 6 shapes currently on the screen are sorted and displayed again. If the Sort shapes button is pressed again, nothing happens because they are already sorted. The video of running the code is uploaded to the Github repository.

The part that did go well in this project was that Shape is the parent and abstract class, and Circle, Square, and Rectangle are implemented as children and concrete classes to create an inheritance structure. Also, it worked well to create a ShapeFactory class that has a dynamic array of Shapes with ArrayList. It creates and adds shapes at random, and then has a method to handle each shape with ArrayList. The part that did not go well in this project was the GUI implementation part. The difficult part was that the buttons were located at the top center, but the shapes had to be implemented with absolute positioning that specifies the positions one by one. I instantiated a JPanel to contain buttons, but it took a long time to solve it because this JPanel prevented the shape from being drawn or the shape was created next to the button instead of the designated location. The most successful method was separately adding JPanel for button and JPanel for shape to JFrame. This way, the buttons, and shapes were displayed properly, and each time the Load button was pressed, new shapes were updated, and when the Sort button was pressed, the existing shapes were updated to the sorted shapes. There are two things I learned from this project. The first was creating an efficient inheritance structure based on what the child classes have in common with Interface or abstract class as the parent. The second is how to implement GUI through Java Swing. I learned a lot about the parent-child relationship between JFrame and JPanel, and how to create an actionListner for a button. If I can recommend three easy ways to do this project for those who are new to this project, first, start by studying the GUI properly, secondly understand the relationship between classes, and thirdly, make UML and then start. As mentioned earlier, the most time-consuming part of this project was the GUI implementation. The project will become easy if the components and relationships of Java Swing's JFrame, JPanel, Component, and their methods are understood. The second is to figure out the relationship between the classes to be implemented in the project. For example, Shape, Circle, Square, and Rectangle are inheritance relationships, and if these relationships can be understood from the beginning, it will be very easy to implement by minimizing overlapping parts. The third is to make UML and then start. If the project is started after making the relationship between classes and the classes mentioned above, and the fields and methods of the classes in UML, the implementation becomes easy. I spent a lot of time on this project, but the knowledge I gained surely improved my OO design skill and it was a helpful project.