James Correia

217 250 331

EECS3311

Section B

Labs Week 3: First software project

TA: Naeiji Alireza

Part I: Introduction

This software project is primarily serves as a small demonstration of software planning and design principles. It has a sufficient level of complexity to demonstrate a variety of relationships between objects which object-oriented languages like Java support. This project will be a success if, the shape generator and sorter work according to the specification, and does so by correctly using OO principles.

There will be some challenges in this project, most notably, learning to use the Java Swing library will be necessary as I have very little experience with it. Furthermore, the lack of knowledge of Swing will complicate planning, as misunderstanding the documentation may cause issues. For instance, confusion with the Swing library may lead to my plan containing impossible or unrealistic object relationships.

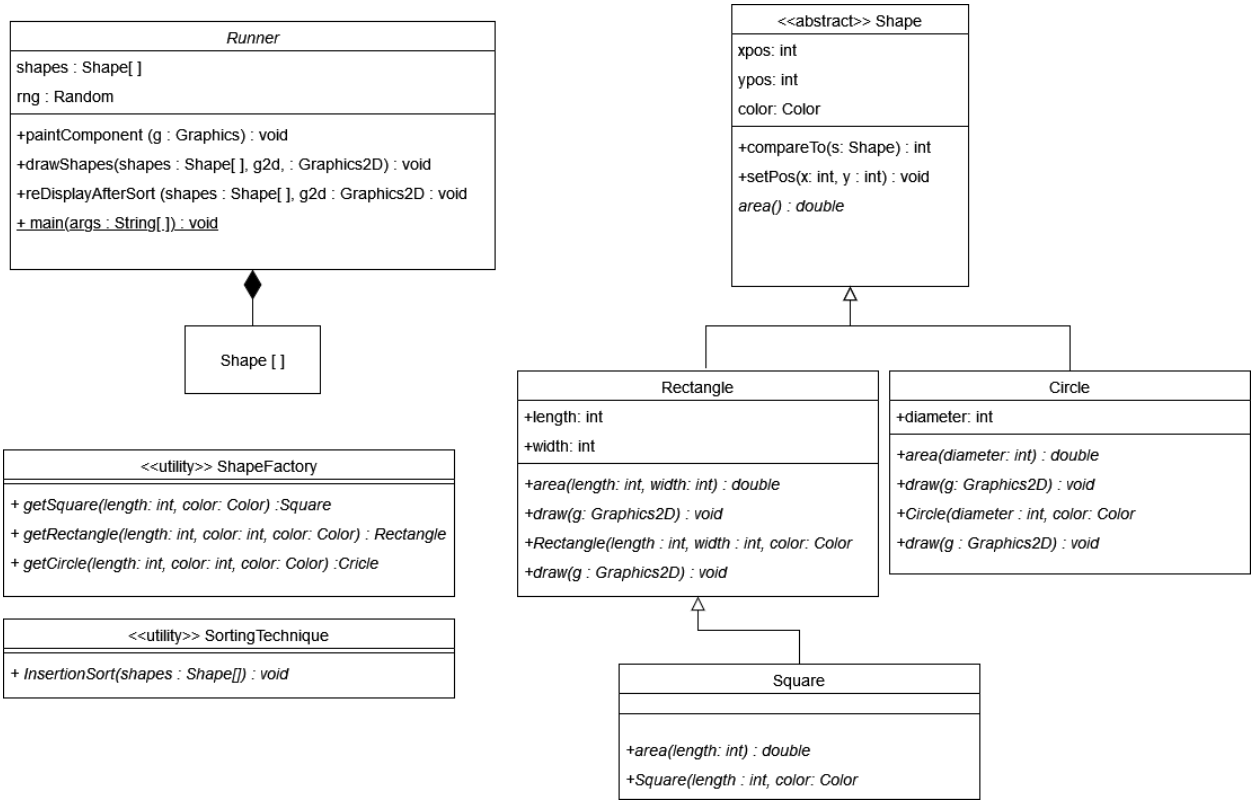
At the outset, there are a few clear ways Object Oriented Design principles will clearly be helpful in modeling the task. For instance, it will be necessary to calculate the area of several shapes, all of which require different arithmetic to do so. This is a great opportunity to use abstraction, so that other classes can request the area of a shape without concern or knowledge of how the task is completed. This will simplify the process of sorting the shapes, as the sorter does not need any awareness of geometry to do so.

Furthermore, many shapes are very different from one another, but all shapes have some things in common. This is a perfect scenario to use inheritance to reuse code. For instance, all shapes have a position, so that can be defined in the parent Shape class, while a specific need such as a circle's diameter can be defined in the child class, without needing to restate any of the fundamentals found in Shape.

With such a variety of shapes to construct, I also find it valuable to create a class following the "factory" design pattern. This way, I can define all constructor behaviors in a single place, without having direct calls to constructors.

Considering these concepts, I will structure the following report keeping these principles in focus. The report will consist of diagrams with along with explanations. They will consider these aforementioned OO concepts most closely, with others taking the periphery.

Part II: Design of The Solution:

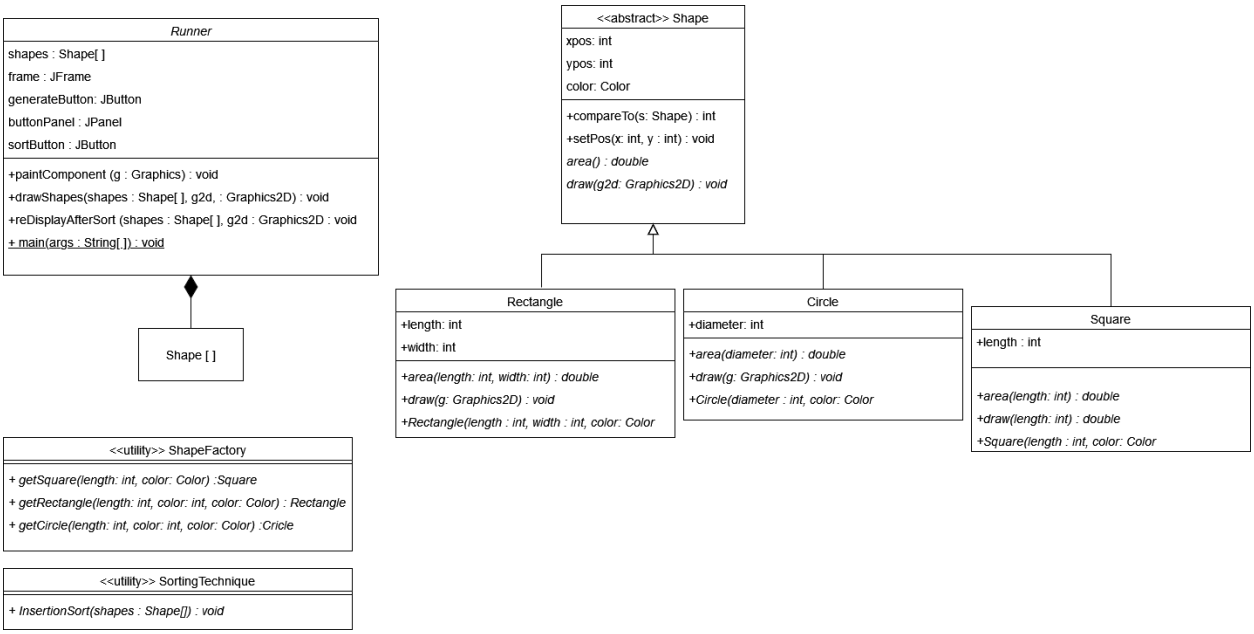


I have employed several OO design principles in this class diagram. Firstly, I have made great use of polymorphism in the design of the software. The draw() method in particular is a great example. draw() is defined as an abstract method in Shape, but in its children Rectangle and Circle there exist implementations. Furthermore, Square, a child of Rectangle, does not have an additional implementation, as the software can draw a square as though it were a rectangle, and it makes no difference to the result. This is a great use of OO principles, as it allows wide functionality across many objects without reuse of code.

I have also included the Object-Oriented concept of inheritance in my design. Shapes differ in many ways, but they also have many things in common. For this reason, my class of Shapes includes three parameters which are inherited by all its children. the X, and Y coordinates, as well as the color of the shape. These will be relevant to all shapes and therefore can be declared a single time in the shape class, and never restated in any “descendant” classes

Additionally, I have used a factory class which contains a list of methods which return class instances, this is a variant of the factory design pattern. I have also included a utility class, an OO design pattern. The SortingTechnique class follows the utility design pattern and contains a single utility that allows the user to use insertion sort on an array of shapes.

However, this is far from the only way the program could be structured, take for instance the following alternative UML diagram:



This diagram differs from the above in a couple of ways. Firstly, Square was reclassified as a direct child of Shape, rather than a “grandchild” through Rectangle as it was in the above example. Consequently, this means draw must be defined separately in Square, rather than inheriting its functionality through Rectangle. Secondly, many of the Java Swing library elements were added as attributes on the runner class, rather than being local variables in main().

I feel this design is inferior to the first, as it deviates from OO principles in many ways, and is generally a more complicated design.

To begin with, the case of Square being a child of Shape: while it is true that Squares could be considered distinct from rectangles, in OO it is good practise to make classes model the real world. Squares, in the mathematical sense are a subclass of rectangles. Some rectangles are squares, but not vice versa. By reflecting this reality in the design, the code follows the OO concept of hierarchies more closely. As well, as mentioned above, the separation of Square from Rectangle causes the draw function to require an additional implementation, rather than inheriting its implementation. OO prefers to reuse code when possible from these sorts of relationships, so this is inferior.

Another difference is the choice to include Java Swing library components as attributes on Runner. This is an unsatisfying solution as well. Firstly, from a perspective of maintaining hierarchies, it is not sensible to disentangle the complex relationship of hierarchy levels which exists in Java Swing in favor of representing them at a single level in the class attributes. Buttons cannot meaningfully exist without a panel to be placed on, and panels too without a frame. In short, these items do not all belong to the Runner, but rather to each other in differing ways,

which should not be changed. Instead, these relations should be defined in the main method, to allow their hierarchies to remain intact.

Part III: Implementation of The Solution:

The sorting technique I opted to use in my solution was an implementation of insertion sort. Insertion sort, in simplified terms, works as follows: Beginning with the second element of the list, it is compared with all preceding elements successively until it finds an element which is smaller than it. Once this occurs, the element is inserted at that position. Room for it already exists, because while iterating the array was being shifted.

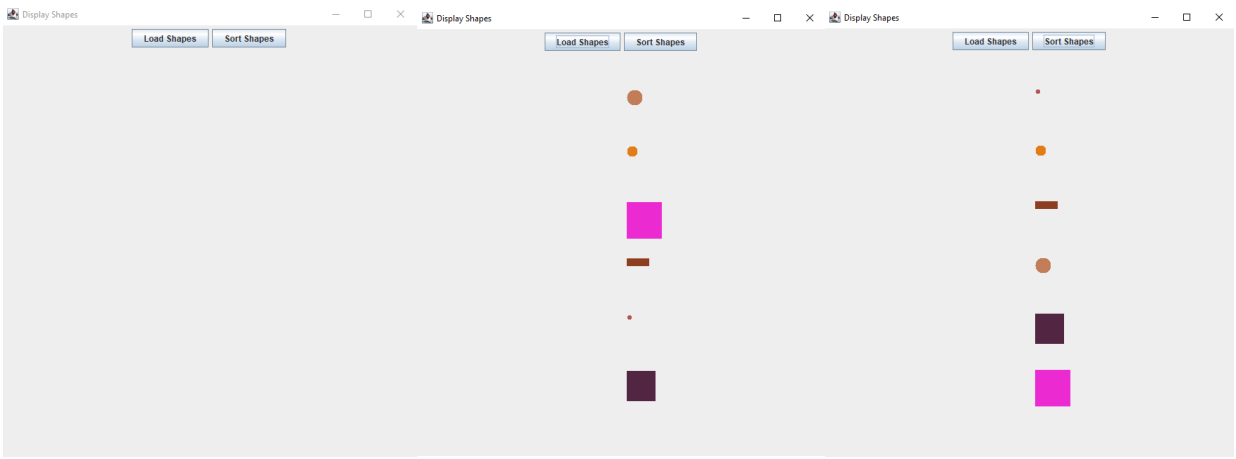
In simplest terms, it can be thought of in the mind as the process of taking each element and placing it where it ought to go on the left side of it. Once all elements have undergone the process the process, it is sorted.

Insertion sort has poor time complexity, however it runs well on small data sets, and so I felt it was appropriate given six elements is rather small.

I implemented my solution following the first UML diagram, as for several reasons already mentioned, it is the far better choice. It roughly can be summarized as an abstract Shape class existing at the top level, with two children, Circle and Rectangle. Rectangle in turn has another child, Square. The Swing library, and all its necessary objects are mostly handled by Runner class, and constructors and sorting are handled by utility classes which offer these services as static methods.

The software was implemented in the JetBrains IntelliJ Idea IDE, building in JDK 16.0.1, using the Swing library to render graphics.

The code when executed looks as follows. (In order of appearance) The initial interface upon starting the program, an example of a selection of shapes generated from pressing “Load Shapes”, and the shapes from the second image having been sorted in ascending order according to size.



Part IV: Conclusion

I feel this software project went very well. I had surprisingly little difficulty ironing out the bugs that are inevitable in any software project, and I feel I had little difficulty matching my code to the specification. What went wrong however, was just as I feared, unfamiliarity with Java's Swing library lead to a large fraction of the development time spent familiarizing myself with the library and how it works, which is quite unconventional and does not follow standard control flow most programmers come to expect. Once the Swing particularities were sorted out, things became much easier.

I learned quite a lot about Java Swing, and would be much more likely to use it again in the future for a personal project. I got experience implementing sorting algorithms in a more realistic environment, compared to previous experiences where the minimum project consisting of a few integer test cases must be sorted. This allowed me to gain a better grasp on the big picture relationships between things in the Comparable interface, the use of the compareTo() method, and passing in multiple object types by taking advantage of the OO paradigm's inheritance relationships.

My top three reccomendations to make this software project easier to complete are to Firstly, begin with getting a familiarity with Swing, as beginning the project in a way which is not Swing-friendly will be a waste of development time. Secondly, to implement the object relationships before attempting any graphical elements. The graphical elements are very implementation-specific, while the relationships between Shapes, Squares, Circles, and Squares are simple and do not depend much on what was done elsewhere in the code, this way you can write the graphical code with better knowledge. My Third reccomendation is to study the UML closely before beginning . By having a big picture understanding of the task, it is easier to remain focused on smaller details which may go unnoticed if you are preoccupied figuring out the design as you go.