**School of Computer Science**

**COMP-3220 Object-Oriented Software Analysis and Design**

**Assignment 4**

**Due: November 30, 2021 – 11:59 pm**

**100 marks – 15% of grade.**

**Objective:** Help students understand techniques relating to refactoring, GUI programming, graphics, MVC pattern, and composite pattern.

**Tasks:**

What’s that funky smell? It’s poorly-written code. The most recent slides describe “code smells”, and provide some solutions for making code less smelly.

I am providing you with a mostly-functional program that allows the user to draw different shapes (rectangles, triangles, and ovals) on a black canvas. The user clicks a button for their desired shape, and then clicks and drags in a diagonal-down-to-the-right motion on the drawing canvas. Upon release the mouse drag, the program will draw the desired shape within a bounding box described by the endpoints of the line drawn by the user. Once the user has at least one shape drawn, they can click the “move” button to enter “move mode”. In “move mode”, they can put their cursor on any number of shapes (shapes can be on top of each other), and then click and drag to move the selected shapes (all shapes under the cursor during the initial click) to a new location upon release of the drag. Finally, there are two more buttons: merge and unmerge. When merge is clicked, all shapes currently on the canvas are merged into a composite shape. A composite shape may be composed of any composite or atomic shapes (e.g. the user may make some shapes, merge, then make some more shapes, then merge again, and so on). The unmerge button removes any composition and makes all shapes independent again. When shapes are merged into a composition, they will be moved by **any** mouse drag in the canvas region.

1. **(Zero marks – just for fun)** Run the program as it stands! Make some rectangles, move your mouse around, and drags those rectangles around, because that’s about all you can do right now.
2. **(25 marks)** Refactor the class Rectangle. Rectangle is great on its own, but we want to implement Oval and Triangles as well. Thus, we will need an abstract class called Shape which encapsulates all the general properties and behaviors of shapes. That is, all Shapes (for our purposes) may have x, y, width, and height. All Shapes will need a move method, and a changeCoords method. All Shapes will need to define their own paint and toString methods (because how we paint, and the String we return from toString, will depend on the specific shape). Thus use the “Extract Class” refactoring technique to divide the Rectangle class up as necessary.
3. **(10 marks - 5 for each class)** Implement a class called Triangle, and another called Oval. Triangle is a green triangle that exists tightly within the provided bounding box, and Oval is a red oval that exists tightly within the provided bounding box.   
     
   See <https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html>, as well as the provided Rectangle class, for inspiration on how to draw a triangle and oval.
4. **(20 marks - 5 for Atomic Shape and 15 for CompositeShape)** Implement Composite Pattern. I have provided you with an abstract Shape class. That’s great, shapes should be abstract because you need to know the specific shape in order to draw one. However, Shape on its own does not allow us to leverage composite pattern. Thus, you need to implement a class called AtomicShape, which is also abstract, which extends Shape only by providing a color attribute. Rectangle, as well as all other concrete shape classes (Oval, Triangle) should also subsequently extend AtomicShape rather than Shape. More importantly, you need to implement a class called CompositeShape. You can zero out its coordinates, width, and height. CompositeShape is the “Composite” class of the composite design pattern. Thus, it needs to maintain a list (e.g. ArrayList) of child shapes (“shapes”) that make this composition. Importantly, the child shapes can be either atomic or composite themselves (as in, the composition should be defined recursively – e.g. like a file structure). CompositeShape must override the following methods from Shape: paint, move, changeCoords, toString(). It has to override them because now each of those methods must be called recursively on all the shapes (atomic or composite) making the composition. toString() should return “Composite: {“ + {toString of each child on a new line} + “}”. In addition, CompositeShape must have a method called “getAllChildren” which returns all of its atomic children, recursively, in a new ArrayList. Lastly, it must have a simple “addShape” method which takes a shape instance and adds it to its ArrayList of Shapes, “shapes”.
5. **(20 marks)** I have provided you with a class called DrawingArea. DrawingArea is missing some things. First, it currently does not have a way to draw Oval or Triangle. Thus, in paint(), you need to extend its functionality to draw ovals and triangles (see how Rectangles are used, and extend from there). You will also have to use the states in the “states” enum to implement these new shapes. DrawingArea also needs a definition for mergeAll(), which needs to create a CompositeShape object and add all the currently-existing shapes to its list of shapes, clear the current list of shapes, and add to it only the new CompositeShape. The class also needs an unmergeAll() method, which creates a new ArrayList of Shapes and adds to its ALL atomic shapes from the current hierarchy of Composite and Atomic shapes (it then replaces its “shapes” instance with the new one, consisting only of atomic shapes). Once this class is ready, uncomment the “mergeAll” and “unmergeAll” calls in the SmellyCode class.
6. **(5 marks)** SmellyCode is a long class with one very long method (code smell: “Long Method”). Refactor SmellyCode by using the “Extract Method” technique to form 5 new and concise methods. One should do all the things currently done in main(). One should construct an individual button, add its ActionListener, and add it to the button panel (e.g. makeAndAddButton). One should perhaps be called “createAllButtons” which creates each of six buttons and adds them to the panel, by repeatedly calling makeAndAddButton. One should create the drawing area, one should create the text area.
7. **(10 marks)** Implement a basic Model-View-Controller (MVC) pattern. The class DrawingArea has some view, control, and model code within it. As discussed during lectures, mixing some view and controller code is reasonable for relatively small classes (like this). However, mixing the model code in with controller and view code is smelly. So, extract a new class, e.g. ShapeManager and migrate some of the functionality from DrawingArea into this new ShapeManager class as necessary. A good place to start would be to move shapesList into this new class, and then migrate any associated functionality into this class. This involves forwarding the method calls to mergeAll and unmergeAll to the new class as well.
8. **(10 marks)** Modify the code in program such that when the user drags on the drawing area, the shapes are moved in real-time. That is, instead of only showing the shapes’ movement upon button release, any selected shapes should move the entire time while being dragged. Ensure that this works on the composite shapes as well.