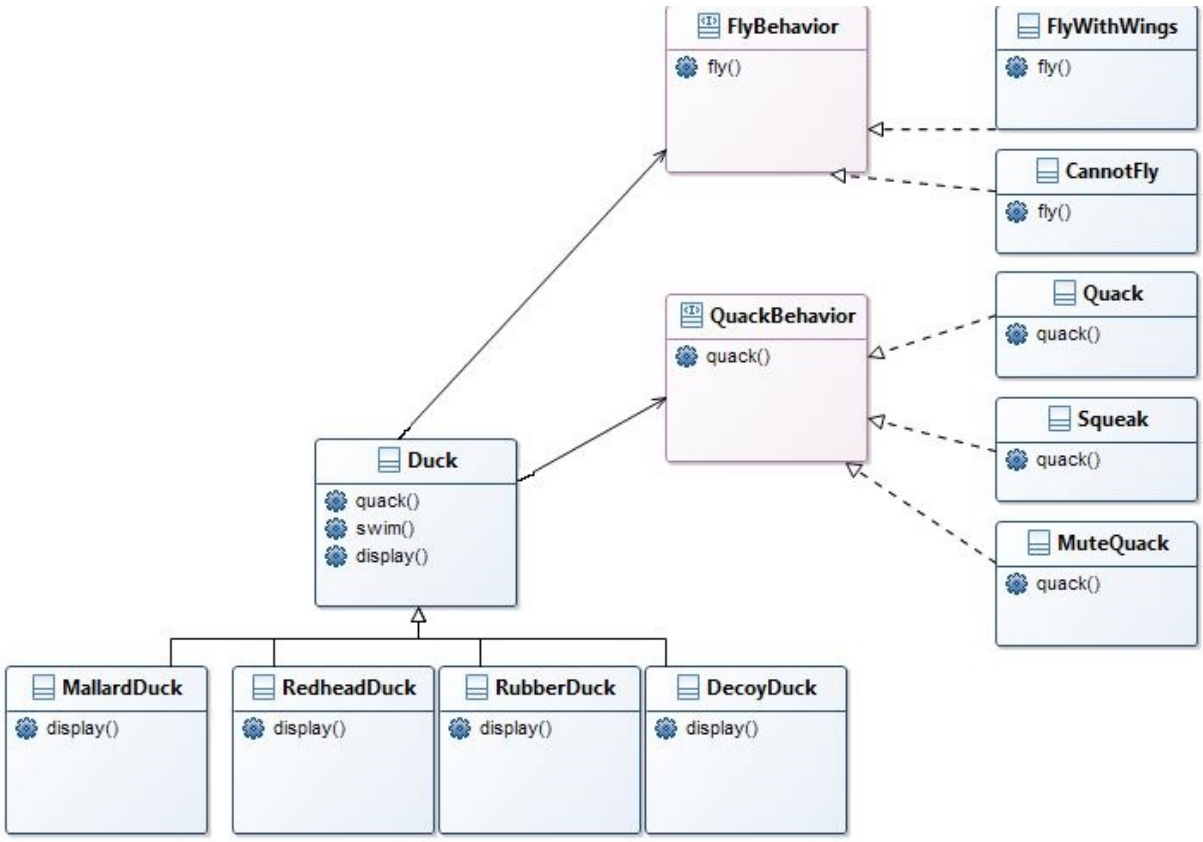
Lab 5

1. A class diagram for the DuckApp, produced below. How will the Duck class use the FlyBehavior and QuackBehavior interfaces? Implement the diagram in Java, and make sure the answer to this question is clear in your code. To implement the methods like fly() and quack(), just print a phrase to the console, like “Flying with wings” or “Quack by squeaking.”



To test your code, create a Main class like the following:

```
public class Main {
    public static void main(String[] args) {
        Duck[] ducks =
            {new MallardDuck(), new DecoyDuck(), new RedheadDuck(), new RubberDuck()};
        for(Duck d: ducks) {
            System.out.println(d.getClass().getSimpleName() + ":");
            d.display();
            d.fly();
            d.quack();
            d.swim();
        }
    }
}
```

Output should look like this:

```
MallardDuck:
  display
  fly with wings
  quacking
  swimming
DecoyDuck:
  displaying
  cannot fly
  cannot quack
  swimming
RedheadDuck:
  displaying
  fly with wings
  quacking
  swimming
RubberDuck:
  displaying
  cannot fly
  squeaking
  swimming
```

2. Create Java classes for `Triangle, Rectangle,` and `Circle` which implements Figure Interface. The figure has one method.

```
public double computeArea()
```

Make all of these classes immutable. (Follow the guidelines in the slides for creating this type of class – included with this lab.) Provide one constructor for each class; the constructor should accept the data necessary to specify the figure, and to compute its area. The values accepted by the constructor should be stored in (private) instance fields of the class. For example, `Rectangle` should have instance fields `width` and `length`, and the constructor should look like this

```
public Rectangle(double width, double length)
```

For `Triangle`, you may use arguments `base` and `height`. And for `Circle`, use `radius` as the constructor argument.

Whenever you create instance fields for one of these classes, provide public accessors for them (but do not provide mutators since the class is supposed to be immutable – for instance, the dimensions of a `Rectangle` should be read-only). For example, you will have in the `Rectangle` class:

```
private double width;
public double getWidth() {
    return width;
}
```

Create a fourth class `Main` that will, in its `main` method, create multiple instances of these figures (you may invent your own input values), store them in a single list, and then *polymorphically* compute and write to the console the sum of the areas.

Typical output:

```
Sum of Areas = 37.38
```

Draw a class diagram that includes all the types mentioned in this problem.

Here are some area formulas:

```
Area of a rectangle  =  width * height

Area of a triangle = 1/2 * base * height

Area of a PI * radius * circle = radius
```

3. Do implementation for the given UML diagram using Factory Pattern.