Lab 5

1. *Rules Framework.* Use the rules framework introduced in the lecture (see lesson5.lecture.factorymethods2) along with the sample UI code provided in lesson5.labs.prob1.samplecode to extend functionality of the UI to incorporate rule validations. Implement the following rules:

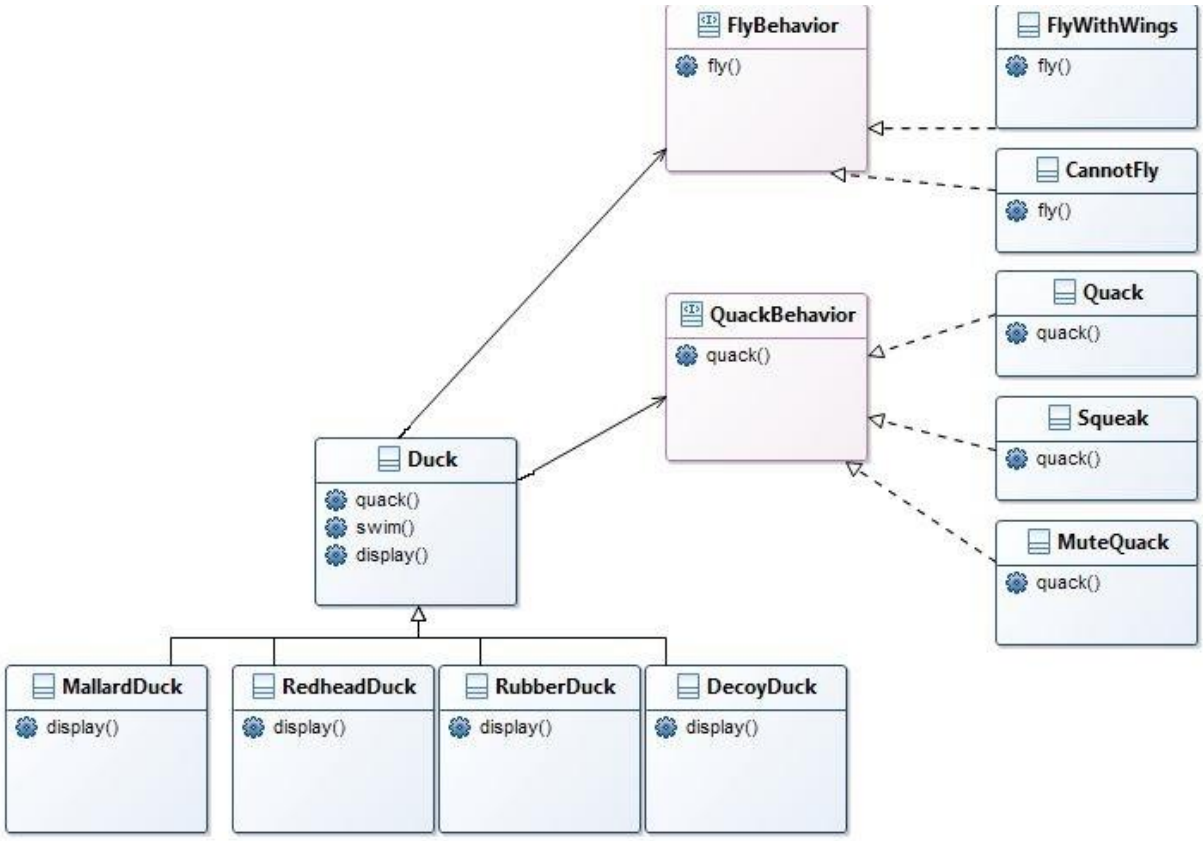
   Address Rules.

   A. All fields must be nonempty
   B. ID field must be numeric
   C. Zip must be numeric with exactly 5 digits
   D. State must have exactly two characters in the range A-Z
   E. ID field may not equal zip field.

   Profile Rules.

   A. Favorite restaurant cannot equal favorite movie
   B. All fields non empty
   C. ID must be numeric
   D. firstname and lastname fields may not contain spaces or characters other than a-z, A-Z.

2. In class, we made progress toward a class diagram for the DuckApp, reproduced 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."



3. Create Java classes for `Triangle`, `Rectangle`, and `Circle`. Provide each class with a method

```
public double computeArea()
```

Make all of these classes <u>immutable.</u> (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. In order to do this, you will need to create an appropriate interface.

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
```