CS2030 Programming Methodology II
Semester 1 2024/2025

Week of 9 September – 13 September 2024
Problem Set #3 Suggested Guidance
Interface and Polymorphism

1. Given the following interfaces.

```
interface Shape {
    public double getArea();
}

interface Scalable {
    public Scalable scale(double factor);
}
```

   (a) Suppose class `Circle` implements both interfaces above. Given the following program fragment,

```
Circle c = new Circle(10);
Shape s = c;
Scalable k = c;
```

   Are the following statements allowed? Why do you think Java does not allow some of the following statements?

   i. `s.scale(0.5);`
   ii. `k.scale(0.5);`
   iii. `s.getArea();`
   iv. `k.getArea();`

   Only `s.getArea()` *and* `k.scale()` *are permissible. Suppose* `Shape s` *references an list of objects that implements the* `Shape` *interface, so each object is guaranteed to implement the* `getArea()` *method.*

   Other than that, each object may or may not implement other interfaces (such as `Scalable`*), so* `s.scale()` *may or may not be applicable.*

   (b) Do the following statements compile?

   i. `c.scale(0.5).getArea()`
   ii. `k.scale(0.5).getArea()`

   The `scale` *method of the* `Circle` *class is defined as*

```
public Circle scale(double factor) {
    return new Circle(this.radius * factor);
}
```

as such `c.scale(0.5)` *returns a* `Circle` *type, and hence* `getArea()` *can be called after it. In the latter case,* `k.scale(0.5)` *returns a* `Scalable` *which has no* `gerArea` *method specified.*

(c) How about defining another combined interface `ScalableShape` as

```
interface ScalableShape {
    public Scalable scale(double factor);
    public double getArea();
}
```

and let class `Circle` implement `ScalableShape` instead?

If class `A` *is only required to define* `getArea()`*, while class* `B` *is only required to define* `scale()`*, then having both classes implement* `ScalableShape` *would force each class to implement methods that it does not need.*

On the other hand, clients should not be exposed to methods it doesn't need. For example, `findVolume` *as a client method that takes in* `ScalableShape` *should not be exposed to the* `scale` *method.*

This is the Interface Segregation Principle which makes up the SOLID principles for Object-Oriented design.

2. During the lecture, we have seen how we can create `Circle` and `Rectangle` as concrete implementations of the `Shape` interface, and pass it to the `findVolume` method:

```
double findVolume(Shape shape, double height) {
    return shape.getArea() * height;
}
```

Now your friend decided to create `Shape` as a class to represent both a circle and rectangle:

```
class Shape {
    private final String type;
    private final double a;
    private final double b;

    Shape(double radius) {
        this.type = "Circle";
        this.a = radius;
        this.b = 0;
    }

    Shape(double length, double width) {
        this.type = "Rectangle";
        this.a = length;
        this.b = width;
    }
```

```
    double getArea() {
        if (this.type.equals("Circle")) {
            return Math.PI * this.a * this.a;
        } else {
            return this.a * this.b;
        }
    }

    public String toString() {
        if (this.type.equals("Circle")) {
            return "Circle with radius " + this.a;
        } else {
            return "Rectangle " + this.a + " x " + this.b;
        }
    }
}
```

which when passed to `findVolume` would still return the same outcome. Justify why this is considered bad program design? *Hint*: what if we need to include a `Square` into our implementation?

By making `Shape` *a class and subsuming the responsibilities of* `Circle` *and* `Rectangle` *into it, we have inevitably transformed* `Shape` *into a "God Class" which results in a number of design issues:*

- *As it now oversees different shapes, a "type" has to be defined to denote the exact shape during object creation leading to a misrepresentation in terms of properties of the class (e.g.* `Circle` *and* `Square` *only needs one property, but* `Rectangle` *requires two);*
- *Method calls (e.g.* `getArea` *and* `toString`*) requires that the "type" to be determined before deciding on the appropriate implementation that is invoked;*

This "god class" violates the Single Responsibility Principle.

Moreover, adding a square will require the `Shape` *class to be modified. We desire a design solution where extensions to the existing code can be "plugged" into the code base with no modifications. By defining* `Shape` *as an interface, with its implementation classes* `Circle` *and* `Rectangle`*, extending the solution with a square simply requires another* `Square` *implementation to be defined. This is the essence of the Open-Closed Principle — software entities should be open for extension but closed for modification.*

3. Complete the method `and` that takes in two `IntPredicate p1` and `p2` and returns a `IntPredicate` that evaluates to `true` if and only if both `p1` and `p2` evaluate to `true`.

```
IntPredicate and(IntPredicate p1, IntPredicate p2) { ... }
```

Express your solution in three different ways:

(a) as a lambda expression;

(b) as an implementation of an anonymous inner class;

(c) as an implementation of a concrete class.

- *Using lambda:*

```
IntPredicate and(IntPredicate p1, IntPredicate p2) {
    return x -> p1.test(x) && p2.test(x);
}
```

- *Using anonymous class:*

```
IntPredicate and(IntPredicate p1, IntPredicate p2) {
    return new IntPredicate() {
        public boolean test(int x) {
            return p1.test(x) && p2.test(x);
        }
    };
}
```

- *Using an implementation of a concrete class*

```
class AndPredicate {
    private final IntPredicate p1;
    private final IntPredicate p2;

    AndPredicate(IntPredicate p1, IntPredicate p2) {
        this.p1 = p1;
        this.p2 = p2;
    }

    public boolean test(int x) {
        return p1.test(x) && p2.test(x);
    }
}

IntPredicate and(IntPredicate p1, IntPredicate p2) {
    return new AndPredicate(p1, p2);
}
```