CS2030 Programming Methodology II

Semester 1 2024/2025

Week of 2 – 6 September 2024

Problem Set #2 Suggested Guidance

Abstraction and Encapsulation

1. Study the following classes `P` and `Q`:

```
 1:   class P {
 2:       private final int x;
 3:
 4:       P(int x) {
 5:           this.x = x;
 6:       }
 7:
 8:       P foo() {
 9:           return new P(this.x + 1);
10:       }
11:
12:       P bar(P p) {
13:           p.x = p.x + 1;
14:           return p;
15:       }
16:   }
17:
18:   class Q {
19:       P baz(P p) {
20:           return new P(p.x + 1);
21:       }
22:   }
```

   (a) Which line(s) above violate the `final` modifier of property `x` in class `P`?

   Line 13 violates the `final` modifier of `x` due to the assignment.

   (b) Which line(s) above violate the `private` modifier of property `x` in class `P`? Relate your observation to the concept of an "abstraction barrier".

   Line 20 violates the `private` accessibility modifier of `x`, while line 13 does not.

   The abstraction barrier sits between the client and the implementer. Here class `P` is the implementer, and `Q` is the client that makes use of an object of `P` (via `p`).

   The barrier is not broken when one one object of type `P` accesses the instance variables of another type `P` object, since they are objects of the same class.

2. We modify the `Point` class to represent the coordinates with a `Pair` object.

```
class Point {
    private final Pair<Double, Double> coord;

    Point(double x, double y) {
        this.coord = new Pair<Double, Double>(x, y);
    }

    private double getX() {
        return this.coord.t();
    }

    private double getY() {
        return this.coord.u();
    }

    double distanceTo(Point otherPoint) {
        double dx = this.getX() - otherPoint.getX();
        double dy = this.getY() - otherPoint.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }

    public String toString() {
        return "(" + this.getX() + ", " + this.getY() + ")";
    }
}
```

(a) Do `getX()` and `getY()` in `Point` violate *Tell-Don't-Ask*?

The principle relates to an external client asking an object for data and acting on that data, instead of the client telling the object what to do. Since both methods have `private` *access, clients are not allowed to call them, and hence the principle is not violated. The methods here actually serve as useful helper methods that abstract out the implementation details of retrieving the individual coordinate values from the underlying data representation.*

(b) Do `t()` and `u()` in the `Pair` class violate *Tell-Don't-Ask*?

For the case of the `Pair` *class, although* `t()` *and* `u()` *seems to be exposing the internal details of a pair of values, it is the responsibility of the* `Pair` *class to provide these functionalities (or services) for clients to use. "Tell" is intended to reveal functionalities of objects, while "Ask" is to reveal the implementation details of objects; and one should avoid the latter. For a simple object like* `Pair`*,* `t()` *and* `u()` *are its functionalities. In summary, "tell" the object what to do (functionalities) via methods; don't "ask" object for data (implementation details).*

3. During the lecture, we developed the following `Circle` to support both circles at a fixed location, as well as empty circles.

```
import java.util.Optional;

class Circle {
    private final Optional<Point> centre;
    private final double radius;

    Circle(Point centre, double radius) {
        this.centre = Optional.of(centre);
        this.radius = radius;
    }

    Circle(double radius) {
        this.centre = Optional.empty();
        this.radius = radius;
    }

    public String toString() {
        return "Circle " +
            this.centre.map(c -> "at " + c).orElse("") +
            " with radius " + this.radius;
    }
}
```

Include the instance method `boolean isOverlap(circle)` that returns `true` if there is an overlap, or `false` otherwise.

```
    boolean isOverlap(Circle circle) {
        return this.centre
            .flatMap(c ->
                circle.centre
                    .map(k -> c.distanceTo(k) < this.radius + circle.radius))
            .orElse(false);
    }
```

Try to see the similarities between declarative programming using stream and optional, particularly in the use of `map` *and* `flatMap`.