Behavioral Subtyping and Static Analysis

This recitation has two parts. In the first part your goal is to better understand behavioral subtyping, which provides a way to characterize good inheritance hierarchies. In the second part you will use a tool called FindBugs to perform static analysis on some provided code.

Behavioral Subtyping

The following must hold for a class `B` to be a behavioral subtype of class `A`:

1. `B` must have the same or stronger invariants than `A`.
2. Overridden methods in `B` must have weaker or the same preconditions than in `A`.
3. Overridden methods in `B` must have the same or stronger postconditions than in `A`.

For each of the supertype/subtype pairs below, is the subtype a behavioral subtype?

1. Consider a `Beverage` which can contain additional ingredients:

```
public class Beverage {
    //@invariant ingredients != null;
    protected List<Ingredient> ingredients;

    //@ensures ingredients != null;
    public Beverage() {
        ingredients = new ArrayList<>();
    }

    //@requires a != null;
    //@ensures ingredients.contains(a);
    public void addIngredient(Ingredient a) {
        ingredients.add(a);
    }
}

public interface Ingredient {
    String getDescription();
}


public class Lemon implements Ingredient {
    public String getDescription() {
        return "lemon";
    }
}
```

```
public class Milk extends Beverage {
    //Milk will curdle with lemons!
    //@requires a != null;
    //@ensures ingredients.contains(a) || (a instanceof Lemon);
    public void addIngredient(Ingredient a) {
        if(!(a instanceof Lemon)) {
            super.addIngredient(a);
        }
    }
}
```

Is `Milk` a behavioral subtype of `Beverage`? Why or why not?

2. Consider a `Point` and `ColorPoint` class which can draw themselves to a `Canvas` below:

```
public class Point {
    //@invariant px > 0 && py > 0;
    private int px, py;

    //@requires x > 0 && y > 0;
    //@ensures px == x && py == y;
    public Point(int x, int y) {
        px = x;
        py = y;
    }

    //@requires canvas != null;
    public void draw(Canvas canvas) { /* ... */ }
}

public class ColorPoint extends Point {
    //@invariant px > 0 && py > 0 && pc != null;
    private Color pc;

    //@requires x > 0 && y > 0 && c != null;
    //@ensures px == x && py == y && pc == c;
    public ColorPoint(int x, int y, Color c) {
        super(x, y);
        pc = c;
    }

    //@requires canvas != null && pc != Color.WHITE;
    public void draw(Canvas canvas) { /* ... */ }
}
```

Is `ColorPoint` a behavioral subtype of `Point`? Why or why not?

3. Consider the following *immutable* implementations of a `Rectangle` and `Square` class, noting that these immutable implementations are different from the `Rectangle` and `Square` we saw in lecture:

```
public class Rectangle {
    //@ invariant w > 0 && h > 0;
    protected final int w, h;

    //@ requires aw > 0 && ah > 0;
    //@ ensures w == aw && h == ah;
    public Rectangle(int aw, int ah) {
        w = aw;
        h = ah;
    }

    //@ requires factor >= 2;
    //@ ensures \result.w == w+factor && \result.h == h+factor;
    public Rectangle scale(int factor) {
        return new Rectangle(w+factor, h+factor);
    }

    //@ requires a >= 0;
    //@ ensures \result.w == w+a && \result.h == h;
    public Rectangle wider(int a) {
        return new Rectangle(w+a,h);
    }
}

public class Square extends Rectangle {
    //@ invariant w > 0;
    //@ invariant w == h;

    //@ requires aw > 0;
    //@ ensures h == w && w == aw;
    public Square (int aw) {
        super(aw, aw);
    }

    //@ requires factor >= 0;
    //@ ensures \result.w == w+factor && \result.h == h+factor;
    public Square scale(int factor) {
        return new Square(w+factor);
    }
}
```

Is `Square` a behavioral subtype of `Rectangle`? Why or why not?

FindBugs

FindBugs is an open source program that statically analyzes Java bytecode for bugs. It has been developed by William Pugh at the University of Maryland since 2003 and can detect more than 400 common programming mistakes in the following categories:

- Bad practice
- Correctness
- Internationalization
- Malicious code vulnerability
- Multithread correctness
- Performance
- Security
- Dodgy code

How to use FindBugs

- Once installed, FindBugs can be directly run on your project or class(es) with right-click → Find Bugs → Find Bugs. When a bug is found, it can be viewed either by selecting the bug icon on the left or by opening the Bug Explorer and Bug Tracker views. The Bug Explorer view present all the bugs found for the given project(s); the Bug Tracker view provides details for each bug.
- Configure FindBugs as follows: Set 'Minimum rank to report' to 20 and check 'Security' and 'Experimental' in the 'Reported bug categories' panel.

  FindBugs can be configured per project (Properties → FindBugs) or per workspace (Preferences → Java → FindBugs). To work with project properties, you need to enable project specific settings.

  In the *reporter* configuration tab, you can select which types of bugs are reported. You can furthermore select the detectors you want to run in the *detector* configuration tab. Other menus enable to create filters and provide many additional settings.
- Import the `rec05` project into your workspace. Run FindBugs on this project. How many bugs does FindBugs report? Are these actual bugs? Are they worth fixing?

Your FindBugs task

Scan your solution of Homework 1 with FindBugs with the same settings described above. Describe three findings from different bug categories. For each finding, explain whether the finding is a real bug or a false positive. If your solution doesn't have enough findings, select a partner and use his or her findings. Are the findings issues that you would fix?