The University of the West Indies, St. Augustine
COMP 3607 Object Oriented Programming II
2020/2021 Semester 1
Lab Tutorial #4

This tutorial focuses on the SOLID design principles.

Learning Objectives:

- Reinforce your knowledge on each of the SOLID design principles
- Discuss examples of SOLID design principles in code snippets
- Refactor existing code to conform to one or more SOLID design principles

Section A: Interface Segregation Principle

The following files are used in this section. Download the java source and examine the code.

```
.
├── Door.java
├── SensingDoor.java
├── Sensor.java
├── TimedDoor.java
└── Timer.java
```

1. What is the purpose of `timeOutCallback()` method of the `Door` interface?

   To lock timed doors after some specified period of time

2. What is the purpose of `proximityCallback()` method of the `Door` interface?

   This is a method that will trigger if a person is near and will assign a boolean value to a locked variable

3. What method(s) of the `Door` interface is `TimedDoor` class forced to implement, even though it doesn't need it?

   proximityCallback() is not needed

4. What method(s) of the `Door` interface is `SensingDoor` class forced to implement, even though it doesn't need it?

   timeOutCallback() is not needed

5. How does this design violate Interface Segregation Property?

This is because the Door interface has become bloated and classes that implement it are forced to use methods that will not be used.

6. How would you change the design so that ISP is not violated?

Create separate interfaces so that we have two types of door interfaces

7. Draw a class diagram to capture these design changes.

See the attached .mdj file

8. Refactor the code to reflect your design changes in an individual project repository.

See the attached java files

Section B: Dependency Inversion Property

The following files are used in this section. Download the java source and examine the code.

```
.
├── Button.java
└── Lamp.java
```

1. There are two classes defined here: Button and Lamp. Are there any dependencies between them? Which class depends on the other?

Yes. Button - a high level module - contains a lamp - a low level module. So, the button depends on the functionality in the lamp class.

2. Suppose we wanted to add a `Fan` class that can be turned on and off to this design and a `Fan` also needed a button.

(a) Describe the problems that may occur.

The Fan will have to turn on and off. There is no guarantee that the on and off functions will be named turnOn() amd turnOff() in the Fan class. So we have a case where the button - a high level module - will be affected if we want to use a Fan instead of a Lamp.

(b) Suppose we handle this by adding a `FanButton` class that allows the `Fan` to be turned on and off. What would the class diagram look like in this case (including `Lamp`, `Button`, `Fan`, and `FanButton`)? Draw the diagram.

See the attached .mdj file

3. What is the common behaviour between `Lamp` and `Fan`?

They both turn on and off

4. Suppose we define an `Equipment` interface for the common behaviour of `Lamp` and `Fan` classes and have `Lamp` and `Fan` classes implement that interface. What would the class diagram look like in this case?

See the attached .mdj file

5. We can now modify the `Button` class to have a reference to an instance of `Equipment` object. Draw the class diagram after this modification:

See the attached .mdj file

6. Refactor the code to reflect your final design in an individual project repository.

See the attached java files

Section C: Multiple Principles

1. Explain how the Open/Closed and the Liskov Substitution principles are automatically satisfied when the Dependency Inversion principle is adhered to.

   This is because you introduce interfaces and if you need to modify anything you can provide a new interface that extends the prior interface, this helps achieve the open/closed principle. While for the Liskov Substitution Principle interfaces you implement can be replaced with interfaces without breaking your application.

2. Consider the code snippet below in relation to the Single Responsibility (SR) Principle

```
public class Customer{
   //state variables and constructors
   public void storeOrder(Order o){ ... }
   public Order findOrder(int orderID){ ... }
   public boolean cancelOrder(Order o){ ... }
   public String getCustomerName(){ ... }
   public String getCustomerAddress(){ ... }
   public String getCustomerEmail(){ ... }
}
```

Figure 1

Assuming that there are no errors in the code in Figure 1:

(a) Explain how the SR principle is being violated. **SRP was violated because the class has multiple responsibilities. It has unnecessary coupling between customer and order details**

(b) Discuss how you would refactor the code so that the SR principle is followed. **Move the order related methods into a new class**

(c) Identify ONE substantial benefit of following the SR principle. **Makes code easier to understand; reduces bugs; reduced side effects of future changes**

(d) Suggest a simple technique that a programmer can use to avoid violating the SR principle when adding a new feature to an existing class.

If when you describe the functionality of the class, you use the word "and", there's a good chance that the class has multiple responsibilities. Move the other responsibilities into a new class