Joseph Maxwell

Marcus Benesch

SOLID Principles in Practice

1. Single-responsibility principle

The original code has all classes and modules within a single class, employee. Within this class, there is a method called toHTML() that returns a string that presents the employee information. This method could be reused by multiple calls for different employees. For the refactored code, this method is separated into a separate class called EmployeeFormatter. Instead of using the variables within the same class, the formatter it calls variables from a specific employee instance. This reduces redundancy within the code and separates and encapsulates one function.

1. Open/closed principle

Open/closed principle is demonstrated from removing the calculations from the area calculator and puts it inside the relevant classes. This ensures that all changes made to or read from are contained locally. The original had to make multiple calls from the shape to calculate the area, but the refactored only needed to make one call and the calculations are contained within the square class.

1. Liskov Substitution Principle

The original code had three classes. The RectangleTest class contained test for both if the object is a square or rectangle. The refactor separated this and made the code in rectangle.java and square.java more consistent with each other. This made the code more simple and consistent.

1. Interface Segregation Principle

Unlike the original, the refactored code contained two interfaces for working and eating. Instead of implementing only one interface of worker for both human and robot, there are now multiple interfaces that cover many different kinds of workers and robot.java no longer requires a special exception from eating, because robots cannot eat.

1. Dependency Inversion Principle

The example code originally had only two classes, “Button.java” and “Lamp.java”, but the refactored code has four classes “Equipment.java”, “Fan.java”, “Button.java”, and “Lamp.java”. The original had a specific link to the lamp to turn on and off the lamp. Within the refactored version, this same link instead goes to a general “equipment” class that is then used to turn on/off specific equipment. The problem this soles is expandability to the code, the original only worked for the specific equipment it was linked to. The refactored code used abstractions in “Equipment.java” to allow for more equipment to be used or expanded upon.

Reflection:

The relevant program in my mind is the java program that I built for the project. It violates all five principles and there are several simple solutions that I could change to improve the program. Currently the “main.java” file does everything. There is separation within the class into different methods, but some of the methods are completely standalone and do not need to be within main.java. the downloader is almost completely standalone and could have been a completely separate class. If separated, this would make readability and discoverability. All that the downloader requires is the date to be downloaded from the state legislature.

Being a single class, I also commit the programming sin of using global variables, I use a few global variables for the SQL connection to the database. I could instead pass the variables to the methods that need it and prevent access to the variables to other classes.