**Repository Link:**

https://github.com/jose98orellana/CPSC501A1

**Report:**

1. **Rename Method**

public void func1(String value) {

firstName = value;

}

From A1.java file

The name of the method didn’t reveal its purpose, so I used the Rename Method, with which I renamed the method giving it a more detailed overview of what it does.

public void setFirstName(String value) {

firstName = value;

}

From refactoredA1.java file

It was tested by setting the first name of a Doctor and then printing it out with the get method. It is better structured since it doesn’t need comments to make its purpose clearer, its function is straightforward and doesn’t need explanation.

1. **Pull Up Method**

class Specialist extends Doctor {

public String getLastName() {

return lastName;

}

class Resident extends Doctor {

public String getLastName() {

return lastName;

}

From A1.java file

I had methods with identical results on subclasses, so I used the Pull Up Method, which removed the identical methods in the subclasses and just added it to the super class in this case Doctor.

class Doctor {

public String getLastName() {

return lastName;

}

From refactoredA1.java file

If it worked in the subclasses it works in the super class, tested with a print statement. It reduces lines of code which enhances efficiency and doesn’t bring up any other complications.

1. **Template Method**

class Specialist extends Doctor {

public double calculateWage(int daysWorked) {

double subWage = daysWorked \* ((erHours \* 40) + (surgeryHours \* 50));

double taxCalc = subWage \* 0.06;

return subWage - taxCalc;

}

class Resident extends Doctor {

public double calculateWage(int daysWorked) {

double subWage = daysWorked \* ((erHours \* 25) + (surgeryHours \* 30));

double taxCalc = subWage \* 0.06;

return subWage - taxCalc;

}

From A1.java file

Two subclasses sharing the same super class had similar methods but not identical, I used the Template Method, which is allocating the similar code into a common super class and the differing code into sub classes.

class Doctor {

public double calculateWage(int daysWorked) {

double subWage = calcSubWage(daysWorked);

return subWage - calcTaxCalc(subWage);

}

public abstract double calcSubWage(int daysWorked);

public abstract double calcTaxCalc(double subWage);

class Specialist extends Doctor {

public double calcSubWage(int daysWorked) {

return daysWorked \* ((erHours \* 40) + (surgeryHours \* 50));

}

public double calcTaxCalc(double subWage) {

return subWage \* 0.06;

}

class Resident extends Doctor {

public double calcSubWage(int daysWorked) {

return daysWorked \* ((erHours \* 25) + (surgeryHours \* 30));

}

public double calcTaxCalc(double subWage) {

return subWage \* 0.06;

}

From refactoredA1.java file

This can be tested by printing out the “calculated” wages of each class and see if it is the right output. It is better structured since you’re moving common stuff that both sub classes use to the super class, so that it is not repeated.

1. **Primitive Obsession**

class Surgery {

public int roomNumber;

From A1.java file

The primitive data type used doesn’t allow for all the attributes needed from a room, so I did “Replace data value with object through the primitive obsession refactoring, to allow for thing like range check or formatting, in my case to keep records of the room the patient is in.

class Room {

private int type;

private int roomNumber;

private boolean status;

public Room(int newRoomNumber) {

if (roomNumber < 301) {

type = 1;

} else if (roomNumber > 300 && roomNumber < 601) {

type = 2;

} else {

type = 3;

}

this.roomNumber = newRoomNumber;

this.status = TRUE;

}

public double getRoomCost() {

if (type == 1) {

return 500;

} else if (type == 2) {

return 750;

} else {

return 1000;

}

}

}

From refactoredA1.java file

Tested by looking for the same output as the non-refactored program. Although it gives the program more flexibility and simpler ways of managing the room object, this method requires us to write more lines of code, so it is possible that after the refactoring it enabled further refactoring’s.

1. **Lazy Class**

class Bill {

public static double calculateBill(Patient patient) {

double surgeryFee = patient.surgery.getSurgeryCost();

double surgeonFee = patient.surgery.s.getSurgeonCost();

double roomFee = 500;

double hospitalFee = 300;

return surgeryFee + surgeonFee + roomFee + hospitalFee;

}

From A1.java file

This class just has one method, which is not enough to justify its existence, so we in the Lazy Class problem we use the move method and move the calculateBill method that the patient uses to the patient class.

class Patient {

public double calculateBill() {

double surgeryFee = surgery.getSurgeryCost();

double surgeonFee = surgery.surgeon.getSurgeonCost();

double roomFee = surgery.room.getRoomCost();

double hospitalFee = 300;

return surgeryFee + surgeonFee + roomFee + hospitalFee;

}

From refactoredA1.java file

This can be tested by expecting the same output as in the non refactored program when calculating the patients bill. This improves the structure by moving the method to a more context related class, reduces lines of code and it removes the need and wasted resources of compiling one more file.