Justin Lee

CMSI 402

Assignment #3

03/12/18

1. You could replace the comment “Get absolute values of a and b” with “Make a and b positive”. You could also get rid of the comment “Repeat until we’re done” as it’s relatively intuitive that the for loop is going to repeat a process and find a solution. You can replace “Set remainder to the remainder of a / b” with “Update the remainder as we’re dividing a and b”. Also, replace “If remainder is 0, we’re done. Return b” with “If remainder is 0, we’re done. Grab the previous remainder”. With these comments, it describes more about what the code is supposed to do in regards to following Euclid’s algorithm rather than describing what the code is doing.
2. One condition is that the programmer is commenting as he or she is coding. They write comments in regards to what the code is doing. However, that means that each time you rewrite your code, you have to rewrite your comments. The second condition is writing all of your code first, and then writing your comments. This leads to comments that are just barely good enough, which actually end up being comments that may not be very intuitive. Both of these conditions leads to writing comments that describes what the code is doing. But, describing what the code is doing is not all that intuitive and results in comments that are not very helpful. Both of these scenarios must focus on what the code is supposed to do. Comments regarding what the code is supposed to do is much more helpful for the person reading the code.
3. When trying to validate your calculations, you are essentially trying to catch errors in your logic. With offensive programming, the whole idea is to have the program indicate those errors loud and clear. You would be applying offensive programming in the sense that if there is something wrong with the calculation or logic, you want the program to scream that there are errors in the code. This can be achieved through numerous ways (e.g. try catch, assertions, throwing exceptions, checking inputs, etc.) It is the opposite of defensive programming where you want to generate an output regardless of the logical flaws in the code.
4. Yes, the whole idea of offensive programming is to catch those errors, scream those errors, and not let the program proceed until those errors are handled. To prevent the program from proceeding and generating an output that has logical flaws, you would want to implement error handling (e.g. If the user enters an invalid input, you don’t want the program to proceed and generate an output from the invalid input. You need to let the user know that the input is incorrect and that a different input is needed).
5. How to drive my car to the nearest supermarket

Assumptions: Person has the basics of driving such as turning the steering wheel, how to park, shifting lanes, knowing what actions can lead to accidents, knowing basic driving laws, etc. (I’m making these assumptions as the questions asks for a **very** high level of describing the steps for the task)

1. Google the nearest supermarket, then get the address of the supermarket
2. Input address of supermarket to a GPS
3. Get in the car

- Open the car door on the driver’s side, then sit on the seat on the driver’s side

1. Start the car/Put the car in drive

- Put the key in the ignition and turn the key until you hear a roar

- Take the gear stick and move the gear stick from park to drive, while holding foot on the brake pedal

1. Follow the instructions of the GPS to know which direction to drive in to get to destination

* Press gas pedal to accelerate/ Press brake pedal to brake
  + When the GPS tells you to turn left, turn left
  + When the GPS tells you to turn right, turn right
  + When the GPS tells you to go straight, go straight
  + When the GPS tells you that you have arrived at your destination, you park in a valid parking spot in that general area.
    - If you’re in a parking lot, park in between two lines
    - If parallel parking, make sure two tires facing curb are less than twelve inches from the curb

1 – 5 are the top level of steps, and each step has sub steps that break down the top-level step to achieve the overall task.

6.

private static void displaySuccessIfTrue(boolean value) {

System.out.println(value ? "success" : "failure");

}

private static void test\_isRelativelyPrime() {

// we’ll call the class for the method the Prime class

Prime p = new Prime();  
 try {  
 displaySuccessIfTrue(p.isRelativelyPrime(19, 5) == true);

} catch(exception e) {

displaySuccessIfTrue(false);

}

try {  
 displaySuccessIfTrue(p.isRelativelyPrime(8, 9) == true);

} catch(exception e) {

displaySuccessIfTrue(false);

}

try {  
 displaySuccessIfTrue(p.isRelativelyPrime(12, 15) == false);

} catch(exception e) {

displaySuccessIfTrue(false);

}

try {  
 displaySuccessIfTrue(p.isRelativelyPrime(12, 13) == true);

} catch(exception e) {

displaySuccessIfTrue(false);

}

try {  
 displaySuccessIfTrue(p.isRelativelyPrime(12, 14) == true);

} catch(exception e) {

displaySuccessIfTrue(false);

}

// you could write many more test cases

}

1. I used black box testing because I have no idea how the method actually determines if two numbers are relatively prime. All I know is what the method is supposed to do, which is to return a Boolean whether two numbers are relatively prime. In my method, I threw some random inputs for the method to see if the method would return the right answer. I could use white box testing if I knew specifics of the code. If I knew only what the code is supposed to do, then I could do black box testing. If I could perform white box testing and black box testing, that gives me some flexibility to perform gray box testing as well which is a combination of both black box and white box testing. You could technically use exhaustive testing if the amount of inputs for the method was finite, but assuming that there are an infinite amount of prime numbers, that wouldn’t be possible in this case.
2. In my initial version of the method, I ran into some bugs, but that was because I implemented the Euclid’s GCD in a slightly different way. Instead of using a for loop, I used a while loop. My initial condition for the while loop resulted in a GCD that was always 0. Therefore, my isRelativelyPrime method would always return false. Therefore, the bugs were due to my logic for the condition of the while loop being flawed, not the logic of the code written in the problem. I fixed the condition that I wrote for the while loop and I ran several test cases for the method, in which they all passed. Therefore, the testing code provided a great benefit, which was determining if my code actually ran correctly. If I didn’t test my code, I wouldn’t have known that my isRelativelyPrime method would always return false regardless of the input.
3. It would probably fall under gray box testing because in black box testing, you’re testing random inputs without knowing where potential weaknesses are. Therefore, there’s a likelihood of missing some test cases. In white box testing, you know where to target the weaknesses, so you can cover those test cases. However, you may overlook test cases that you assume work. Therefore, with gray box testing, you incorporate both types of testing. You would be testing cases that attack the weaknesses of the code, as well as testing cases that you assume work. Therefore, you would be covering all possible test cases.
4. You can use the Lincoln index by multiplying the number of bugs each person has found, then dividing by the number of bugs that they all have in common. Therefore, since Alice found 5 bugs, Bob found 4 bugs, and Carmen found 5 bugs, multiplying those together would give 100 bugs. They all found 1 bug in common, so you divide 100/1, which gives you about 100 bugs that are in the code.
5. The Lincoln estimate would produce a value that’s undefined since you would be dividing a number by 0 (the number of bugs found in common). This means that you can’t assume that the odds of finding a certain bug are the same for everybody. One explanation for this is that people may test code in different ways, which produces different chances for people to find a certain bug. To produce the “lower bound” estimate, you could add up all the bugs that each programmer found.