Max Williams

3/19/2018

Assignment 3

1. Problem 7.1

Removed unnecessary comments that describe what code is doing, rather than good comments that should describe what the code is supposed to do:

```
// Use Euclid's algorithm to calculate the GCD.
private long GCD( long a, long b )
{
    a = Math.abs( a );
    b = Math.abs( b );


    for( ; ; )
    {
        long remainder = a % b;
        If( remainder == 0 ) return b;
        a = b;
        b = remainder;
    };
}
```

2. Problem 7.2

Bad comments might show up in code because the programmers are writing their comments as they revise the program, or waiting until the program is complete and adding comments then. Instead, comments should be written preemptively, describing what the code is supposed to do, not how it does it.

3. Problem 7.4

Before calculating the GCD of long a and b, this function could check to see if the absolute value of a or b causes an overflow, and if so return an error. Additionally, this function could also throw an offensive error if either a or b is 0 to avoid issues with modulo operations.

4. Problem 7.5

Yes, error handling should be added to this code because proper error handling enables easier maintenance and debugging of the code through human readable error messages. Additionally, error handling can be written prior to the logic of a function, outlining its purpose and requirements.

5. Problem 7.7

    a. Prepare - Gather required items such as keys, driver's license, and directions.

    b. Start - Get into the car, adjust the mirrors, lock seatbelt, and start the car.

    c. Navigate - Using the directions, navigate to the supermarket using the gas pedal, brake pedal, turn signals and traffic signals.

    d. Stop - Find a parking spot, drive car into the spot, turn off the car and disembark.

    Assumptions:

    - The driver has access to a car and other required materials.
    - The driver has good vision and can physically operate a car.
    - The driver knows traffic laws and how to obey them.

6. Problem 8.1

```
def isRelativelyPrime_test(self):
      self.assertEquals(isRelativelyPrime(21, 35), False)
      self.assertEquals(isRelativelyPrime(0, -1), True)
      self.assertEquals(isRelativelyPrime(11, 2), True)
      self.assertEquals(isRelativelyPrime(11, 11), False)
      self.assertEquals(isRelativelyPrime(78, 102), False)
```

7. Problem 8.3

The testing method used in Problem 8.1 is black box testing since I didn't write the isRelativelyPrime() function and have no knowledge of how it works. However, if I did write this function or had knowledge of how it works than white box testing could be used.

8. Problem 8.5

Since AreRelativelyPrime() calls GCD() to perform its calculations its inputs have the same conditions as GCD()'s inputs. Therefore, testing with inputs that may cause overflow errors are important test cases that should be added.

```
def isRelativelyPrime_test(self):
     self.assertEquals(isRelativelyPrime(21, 35), False)
     self.assertEquals(isRelativelyPrime(0, -1), True)
     self.assertEquals(isRelativelyPrime(11, 2), True)
     self.assertEquals(isRelativelyPrime(11, 11), False)
     self.assertEquals(isRelativelyPrime(78, 102), False)
     self.assertEquals(isRelativelyPrime(14901470985302202821278,
1082739393028112), True)
     self.assertEquals(isRelativelyPrime(-282330020211773387, 0),
False)
```

9. Problem 8.9

Exhaustive testing falls into the category of black box testing. This is because exhaustive testing relies on testing every possible combination of types of inputs. This approach prefers no prior knowledge of how the code works to remain unbiased and test all of these possible inputs.

10. Problem 8.11

Alice and Bob: (5 * 4) / 2 = 10

Alice and Carmen: (5 * 5) / 2 = 12.5

Bob and Carmen: (4 * 5) / 1 = 20

Average = (10 + 12.5 + 20) / 3 = 14.166 ~ 14 bugs found

Depending on the size of the program, and what parts of the program have been tested we can then use this number to estimate the total number bugs remaining.

11. Problem 8.12

If two testers don't find any bugs in common it means that there is still an excess of bugs remaining in the software, and because of this the two testers aren't finding any overlapping bugs. When testers are finding overlapping bugs it provides a lower bound for the remaining number of bugs in the software, but without it we can't estimate the number remaining.