Alex Verdin

CMSI 402

Prof. Johnson

March 11, 2018

Homework 3

7.1) The comments are explaining in English what the code does but does not properly explain what the code does or why it is needed. Fixes to the comments can be seen as follows:

// Use Euclid's algorithm to calculate the GCD. See:

// https://en.wikipedia.org/wiki/Euclidean\_algorithm

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;

};

}

* 1. The two conditions that may generate bad comments are writing comments as you code or writing comments after code is already written.
  2. You can apply offensive programming by ensuring that the inputted values of a and b are valid inputs that can reasonably be calculated. Fixed code will look like:

private long GCD(long a, long b)

{

Debug.Assert (a >= 0 && b >= 0, “not a valid input”);

checked

{

a = Math.Abs(a);

b = Math.Abs(b);

for (; ; )

{

long remainder = a % b;

if (remainder == 0) return b;

a = b;

b = remainder;

};

}

}

* 1. Yes, error handling should be added to this code that way the code will not immediately crash when an exception is thrown.
  2. Some assumptions to make before organizing the highest level would be that the person first understands how to properly drive a car (how to turn the car on, backout of their driveway, follow rules of the road, park car, follow directions from a GPS or map, etc.) and that they have knowledge of how to use access a GPS (whether it is from their phone, from an integrated car GPS, Map Quest, etc.). The first instruction would be to locate the closest supermarket using a GPS; let’s call this class locateClosestMarket(). The next step would be to follow the instructions given from the GPS by driving in your car; let’s call this class followGPSInstructions(). Note: Although functions like startCar() or backOutOfDriveway() may still be relatively high level, I am not considering them the highest possible level and am assuming these as part of the followGPSInstrictions() class. The last step would be to finally reach the destination; let’s call this reachedDestination(). Note: reachedDestination() most likely returns a boolean value, though other lower level instructions like parkCar() may be part of this step or follow this step but I am assuming are not as high level as determining that the destination has been reached.
  3. IsRelativelyPrimeTest()

// assume a and b as two variables of IsRealativelyPrime()

{

// Test if both values are the same and not 1

if(IsRelativelyPrime(a,a) (where a is not 1) == false)

{

// Test passed

}

// Test where a and/or b are above or below the max and min values

IsRelativelyPrime(a,b);

if (a and/or b > 1 million || a and/or b < -1 million)

{

// If some exception thrown then test passed

}

//Test where no b is inputted

IsRelativleyPrime(a);

// should return some exception that a second number is not entered

// Test max and min values for a and b

IsRelativelyPrime(a = 1 million, b = -1 million);

// Test passed if no exception is thrown

// Test a and b not numbers

IsRelativelyPrime(a,b)

// test passed if exception thrown

// Test 1000 random a and b values between 1mil and -1mil

for (int i = 0; i < 1000; i++)

{

IsRelativelyPrime(a = random#, b = random#);

// Test passed if no exceptions are thrown or crashes

}

}

* 1. The technique used for exercise 1 was a black-box technique. The goal was to test the robustness of the code, not necessarily weather it actually calculated the proper values. Black-box was good for this situation since the actual implementation of the function was not given or known. You could use white-box testing under the condition that you fully understand how the function was executing. This technique could be more focused on picking specific examples that could potentially break the code. Gray-box testing could also be used if you had some prior understanding to what method you would use for the function. Gray-box could cover some specific examples while also testing overall robustness of the function. An exhaustive technique is most likely not applicable because there are too many combinations of numbers to justify testing every single option.
  2. There are not necessarily bugs in my initial testing code (8.1) however, the implementation here does not really account for the same kind of robustness that I was testing for initially. For example, I tested more for if exceptions were thrown if wrong inputs were inputted rather than specific examples of the functionality.
  3. Exhaustive testing falls into the black-box category because it is a type of testing that does not necessarily need to know any information about how the function is being implemented. Exhaustive testing is only worried about seeing if every possible combination will pass through your function.
  4. You can calculate the estimation of the number of bugs by finding the Lincoln estimate for every combination of two testers (Alice and Bob, Alice and Carmen, and Bob and Carmen). Then take the average of the three Lincoln estimates to estimate the number of bugs. Using this method, the total number of bugs is about 14 bugs so there are still 4 bugs unaccounted for.
  5. If no bugs in common are found, the Lincoln estimate would be divided by zero meaning that the Lincoln estimate cannot be accurately calculated with the data (or even calculated at all as you cannot divide by zero). However, you can obtain a lower bound estimate of the number of bugs since if you know that both testers have no bugs in common, there are at least the sum of both testers bugs that can be accounted for.