Ballpark Bookie

402

Homework 3

7.1

// 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;

};

}

7.2 The two conditions that may result with poor comments is that the programmer is trying to thoroughly explain the code, which can lead to redundant comments and also the comment might not necessarily explain the purpose of the line of code, but rather what it does, preventing onlookers from getting a deeper insight of the code.

7.4

The code written in problem 3 is already pretty offensive. It validates the inputs, results, and the Debug. The assert method throws an exception if there is a problem.

7.5

The way the code works right now, if there are any exceptions, they get passed up to the calling code so they can be handled there. This means we don’t need to add error handling code here. We could add it to the GCD method, but because we want the the errors to be handled by the calling code, there is no point.

7.7

a. Find my car.

b. Open the door to the car.

c. Start the engine

d. Back out of the parking space.

e. Make a U-turn in the cul de sac and make a left at the next intersection.

f. Turn right. Drive until the street ends.

g. Turn right. Drive through the first and second stop sign. Drive to the third stop sign.

h. Turn right. Drive through four stoplights and be prepared to make a left.

i. Turn left. Make the first right into the shopping center parking lot.

j. Find an empty parking space and park in it.

l. Stop the car and get out.

m. Go buy strawberries.

This description makes the assumptions:

a. The car is parked in the driveway.

b. You properly adjust the seat and mirrors when you enter the car.

c. The car has gas.

d. There’s nothing behind the car when you pull out.

e. The streets are clear and you don’t get in any form of accident

f. There are empty parking spaces in the supermarket parking lot.

g. The supermarket is open.

H. The store has strawberries in stock and they are in the produce section.

8.1

|  |
| --- |
| private bool Test\_AreRelativelyPrime(int val\_1, int val\_2) {  //Convert to positives  val\_1 = Math.abs(val\_1);  val\_2 = Math.abs(val\_2);   //Prime if either value is 1  if( (val\_1 == 1) || (val\_2 == 1) ) {  return true;  }   //Not prime if either value is 0  if( (val\_1 == 0) || (val\_2 == 0) ) {  return false;  }   //Checking for factors of val\_1 & val\_2  int val\_low = Math.Min(val\_1, val\_2);  for (int factor = 2; factor <= val\_low; factor++) {  if ( (val\_1 % factor == 0) && (val\_2 % factor == 0) ) {  return false;  }  }  return true; } |

8.3

This is a black-box test due to the fact that we only know what the “AreRelativelyPrime” method is supposed to do, but not how the method works.

If we had been shown or told how the “AreRelativelyPrime” method works, we would then be able to use either white-box or gray-box testing. If we knew how the method worked and chose white-box testing, we would then proceed to try to create tests that may make the method fail. Gray-box is a mix of the previous two options, since we know a part of how the method works but not all so specific tests are designed with that knowledge in mind.

8.5

When testing, it was found that the given version of AreRelativelyPrime had issues handling very large and very small values, so it was necessary to implement some restrictions on the permitted values of a and b. Also, in the case of Test\_AreRelativelyPrime, the values -1, 0, and 1 gave the new helper method some trouble but that ended up showing that testing is very useful in trying to think about edge cases and how to deal with them.

8.9

Exhaustive tests are actually black-box tests since there is no underlying knowledge of how the method being tested works.

8.11

Use 3 different pair of testers:

* Alice & Bob → 5 \* 4 / 2 = 10
* Bob & Carmen → 4 \* 5 /1 = 12.5
* Carmen & Alice → 4 \* 5 /1 = 20

8.12

If the two testers didn’t find any bugs in common, the Lincoln estimate is divided by 0, resulting in an infinite answer. This means that the users have no idea how many bugs there are. It is possible to get a lower bound by pretending they found a bug in common to use the Lincoln estimate.