7.1-
The comment "`// Use Euclid's algorithm to calculate the GCD.`" can be replaced with:
//Using Euclid's algorithm to calculate GCD which you can find at en.wikipedia.org/wiki/Euclidean_algorithm
This allows for the other comments to be removed from the code as an explanation for the algorithm can be found at the link.

7.2-
Two conditions where you might end up with the bad comments shown in the example are when using top-down design or if comments were jammed into the code after the code was written in an effort to explain each line.

7.4-
You could add offensive programming to the modified code by making sure that the variables provided fit into the requirements. This can be done by using a try/catch to make sure that a and b are greater than 0 and throw an error if they are not or change them to their positive values so that it is runnable.

7.5-
Yes, you could add error handling code to the modified code.

7.7-
Instructions:
Start your car
Exit the parking space
Drive up to the gate and stop
Wait for gate to open
Drive through the gate opening
Look to make sure there are no cars coming from left or right
Make a left
Stop at the stop sign
Look to make sure there are no cars coming from left or right
Make another left
Make a right at the stop sign
Go straight at the light
Turn left into the parking lot
Find a parking space
Park
Turn off the car
Exit the car

Walk into the store

Assumptions:
Driver is already in their car
Their car is located at my parking space
Driver will continue to drive straight until another stop sign or light
Driver will check to make sure it is clear to turn
Driver will stop at the light or will drive through if green
Driver will yield to oncoming traffic at turns

8.1-
validPrime() tests that the two integers are relatively prime

testPrime():
For 500 simulations get random a and b then
Assert IsRelativelyPrime(a, b) = validPrime(a, b)

Assert IsRelativelyPrime(-1,000,000, -1,000,000) = validPrime(-1,000,000, -1,000,000)
Assert IsRelativelyPrime(1,000,000, 1,000,000) = validPrime(1,000,000, 1,000,000)
Assert IsRelativelyPrime(-1,000,000, 1,000,000) = validPrime(-1,000,000, 1,000,000)
Assert IsRelativelyPrime(1,000,000, -1,000,000) = validPrime(1,000,000, -1,000,000)

For 500 simulations get random a then
Assert IsRelativelyPrime(a, a) = validPrime(a, a)
Assert IsRelativelyPrime(a, 1) is relatively prime
Assert IsRelativelyPrime(1, a) is relatively prime
Assert IsRelativelyPrime(a, -1) is relatively prime
Assert IsRelativelyPrime(-1, a) is relatively prime
Assert IsRelativelyPrime(a, 1,000,000) = validPrime(a, 1,000,000)
Assert IsRelativelyPrime(a, -1,000,000) = validPrime(a, -1,000,000)
Assert IsRelativelyPrime(1,000,000, a) = validPrime(1,000,000, a)
Assert IsRelativelyPrime(-1,000,000, a) = validPrime(-1,000,000, a)
If a is not +/- 1 then
Assert IsRelativelyPrime(a, 0) relatively prime
Assert IsRelativelyPrime(0, a) relatively prime

8.3-
This is a black-box test as it is primarily focused on the functionality of the program. There is the ability to use an exhaustive test, but it would be too complicated because of the allowed range of numbers. If the range were smaller, an exhaustive test would be more reasonable to use.

8.5-
Outside of implementation from my pseudocode, the largeness of the number range was another hurdle. The allowed range for the method was the biggest hardship to overcome.

8.9-
Exhhuastive tests fall into black-box testing as they are concerned with the functionality rather than the underlying components of a program.

8.11-
(5 + 4 + 5) / 5 = 14/5 = 2.8
2.8 x 5 = 14
Total Bugs = 14

14 - (5 + 4 + 5) = 0
Bugs still at Large = 0

8.12-
You cannot use the Lincoln estimate if there are no common bugs found. This is because 0 would be in the denominator and that would not work. This would mean that there would be an infinite amount of bugs estimated. To get around this, you can use the total number of bugs or 1 as the denominator to still get an estimate.