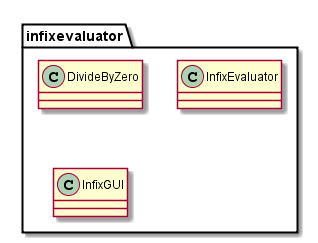
CMSC 350 – Project 1: Infix Evaluator

UML for written classes: 1

Test Plan: 2 – 4

Lessons Learned: 5

1. **UML**



1. **Test Plan**

* *Test each operator:*
  + *Testing no-spaces, order of operations, parentheses*
  + Input: (2+3\*5)-8/5\*(5-2)
  + Expected Output: 14
  + Actual Output: 14

**PASS**

* + Input: 72+9
  + Expected Output: 81
  + Actual Output: 81

**PASS**

* + *Testing with spaces*
  + Input: 8 - 8
  + Expected Output: 0
  + Actual Output: 0

**PASS**

* + *Testing no spaces and spaces*
  + Input: 8- 22
  + Expected Output: -14
  + Actual Output: -14

**PASS**

* + *Testing spaces and no spaces*
  + Input: 4 \*7
  + Expected Output: 28
  + Actual Output: 28

**PASS**

* + Input: 10 / 2
  + Expected Output: 5
  + Actual Output: 5

**PASS**

* + *Testing integer division (rounding down)*
  + Input: 20 / 7
  + Expected Output: 2
  + Actual Output: 2

**PASS**

* + *Testing integer division with result = 0*
  + Input: 4/5
  + Expected Output: 0
  + Actual Output: 0

**PASS**

* *Test precedence/order of operations*
  + *Testing higher precedence ops after lower precedence, parentheses*
  + Input: 4+3\*5-2/(1+1)
  + Expected Output: 18
  + Actual Output: 18

**PASS**

* + *Testing with lower precedence after higher precedence ops, parentheses*
  + Input: (5+5)/2\*3+1-4
  + Expected Output: 12
  + Actual Output: 12

**PASS**

* + *Test nested parentheses*
  + Input: (((5-2)+3)\*4)
  + Expected Output: 24
  + Actual Output: 24

**PASS**

* *Test division by 0*
  + Input: 5/0
  + Expected Output: JOptionPane with message “Denominator cannot be 0” (program still runs)
  + Actual Output: JOptionPane with message “Denominator cannot be 0” (program still runs)

**PASS**

* *Test unexpected/invalid input*
  + *Test no input – was not sure what type of error (if any) to expect*
  + Input:
  + Expected Output: JOptionPane with error message (program still runs)
  + Actual Output: JOptionPane for EmptyStackException (programs still runs)

**PASS**

* + *Test with invalid characters*
  + Input: 2a \* 3
  + Expected Output: JOptionPane – IllegalArgumentException (program still runs)
  + Actual Output: JOptionPane – IllegalArgumentException (program still runs)

**PASS**

* + *Test with invalid characters*
  + Input: b
  + Expected Output: JOptionPane – IllegalArgumentException (program still runs)
  + Actual Output: JOptionPane – IllegalArgumentException (program still runs)

**PASS**

* + *Test malformed equation - parentheses*
  + Input: (5 – 2)) + ((3 – 2)
  + Expected Output: JOptionPane – IllegalArgumentException (program still runs)
  + Actual Output: JOptionPane – IllegalArgumentException (program still runs)

**PASS**

* + *Test malformed equation – orphaned operator*
  + Input: 5 ++ 3
  + Expected Output: JOptionPane – IllegalArgumentException (program still runs)
  + Actual Output: JOptionPane – IllegalArgumentException (program still runs)

**PASS**

* + *Test malformed equation – orphaned operator*
  + Input: + 5 + 3
  + Expected Output: JOptionPane – IllegalArgumentException (program still runs)
  + Actual Output: JOptionPane – IllegalArgumentException (program still runs)

**PASS**

1. Lessons Learned

The biggest takeaway from this project is that I explicity leveraged the power of a stack data structure. This is something that I may have implicitly done in the past via recursion, but an integral part of the algorithm requires the use of a stack in order for this to work. Additionally, this was the first time I have been had to write the code for a GUI myself rather than having it generated by an IDE. So although I was already aware of how it worked, I had to learn or relearn the parts for the actual layout of the GUI.

Although DivideByZero was required, I wanted the program to run gracefully no matter what exception and to be able to give useful feedback to the user. This definitely helped in my understanding not just through practice, but also by forcing myself to understand where an exception was ocurring and to evaluate whether I should simply catch it or prevent the program from ever reaching that state. One example was with the EmptyStackException that would get thrown when *solveNext()* was called. This would happen because it pops 2 off of numbers and 1 off of operators. I figured out that the reason this state would ever be reached is due to a malformed equation or an invalid format. So, instead I check for valid stack sizes and if they aren’t then an IllegalArgumentException is thrown since this is the cause of even getting an empty stack in this case.