**David Anderson, Daniel Justice**

**CS 1632 –   
Deliverable 4: Performance Testing Using VisualVM**

<https://github.com/danieljustice/Quality-Assurance-D4>

**Introduction**

To start off the program, we used Visual VM 1.3.9 to sample the cpu usage of the JBefunge program when taking in the input specified in the deliverable requirements. Upon looking at the cpu usage, we saw three methods that took up a large amount of cpu time in comparison to every other method. They were ProgramArea.getOpCode , ProgramStack.toString and ProgramExecutor.modulo.

The first method that we decided to perform pinning tests on and refactor was ProgramStack.toString. In this method, there is a string called throwaway that has a new string created and then that new string is added to itself a thousand times. This is a very expensive set of actions to be making, and it also it has no relevance in turning another data type into a string. For refactoring, we took out this functionality and thus, the speed of toString increased. For testing, we checked that before and after refactoring the toString method properly handled taking in different sized integers, could handle negatives and would successfully return Max Int and Min Int as strings.

The second method we investigated was ProgramArea.getOpCode. For this method, there is a double for loop that iterates over all the x y coordinates and returns a bunch of values starting from -80000 leading up to the passed in x and y values. This is very expensive because there is no need to go through these strings just to return the x and y of the area. In refactoring, we took out the unneeded double for loop and then returned the x and y input that is passed into the function. For testing, we checked that it would successfully return null when Max Int or negatives were passed in as parameters. We also checked that if there was a ‘0’ to be read in it will return the ASCII value for ‘0’ and not the ASCII value for ‘nul’.

The last method we investigated was ProgramExecutor.modulo. In this method, there is a for loop that keeps track of a location and performs some Math functions on this location. The for loop performs these 120,000 times. This is a very large amount of times just for computing a modulus. In refactoring, we took out this for loop and deleted the location variable. There is no need for this because modulus has nothing to do with a location. It should just use the stack to get its A and B data sets. For testing, we checked four specific instances of modulus. We tested negative values for both sides of the modulus function, zero for the left side of the modulus function, max integer modulus, and testing that it successfully throws an ArithmeticException when having zero on the right of the function.

**Average Times**

**Before Changes**

Run 1: 20,062ms

Run 2: 20,242ms

Run 3: 20,268ms

**Arithmetic Mean: 20190.667ms**

**After Changes**

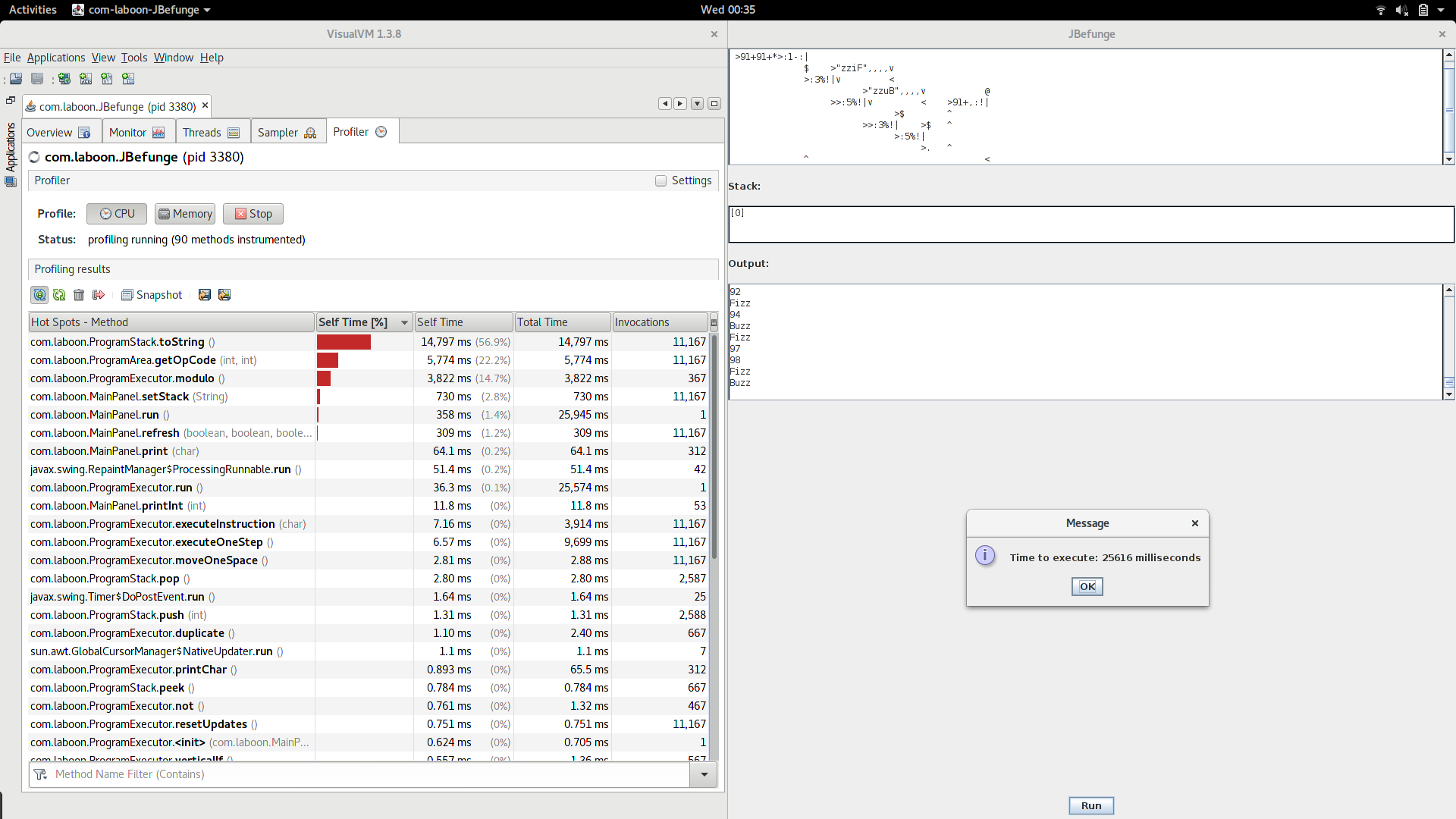
Run 1: 1413ms

Run 2: 1363ms

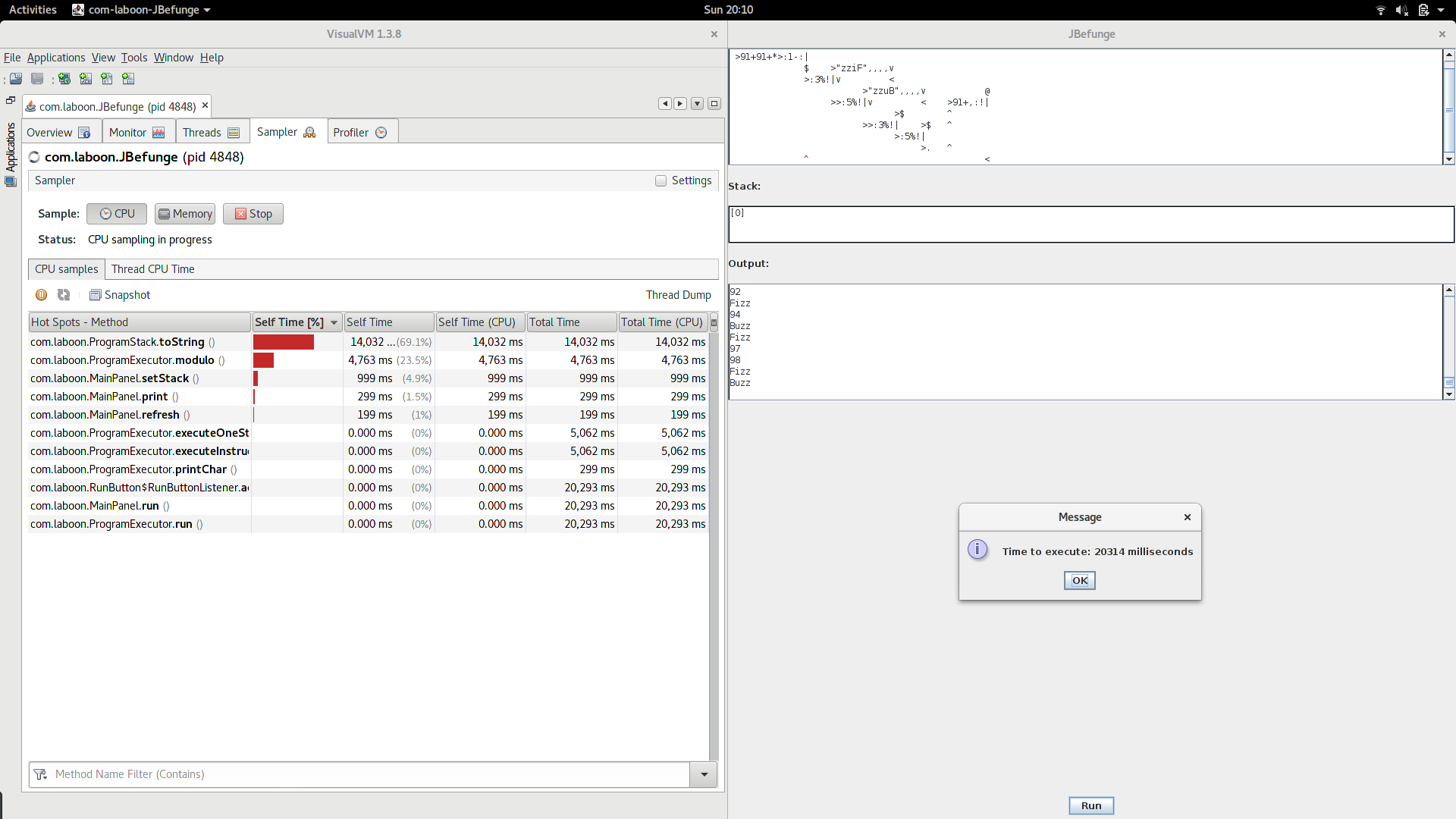
Run 3: 1370ms

**Arithmetic Mean:** 1382ms

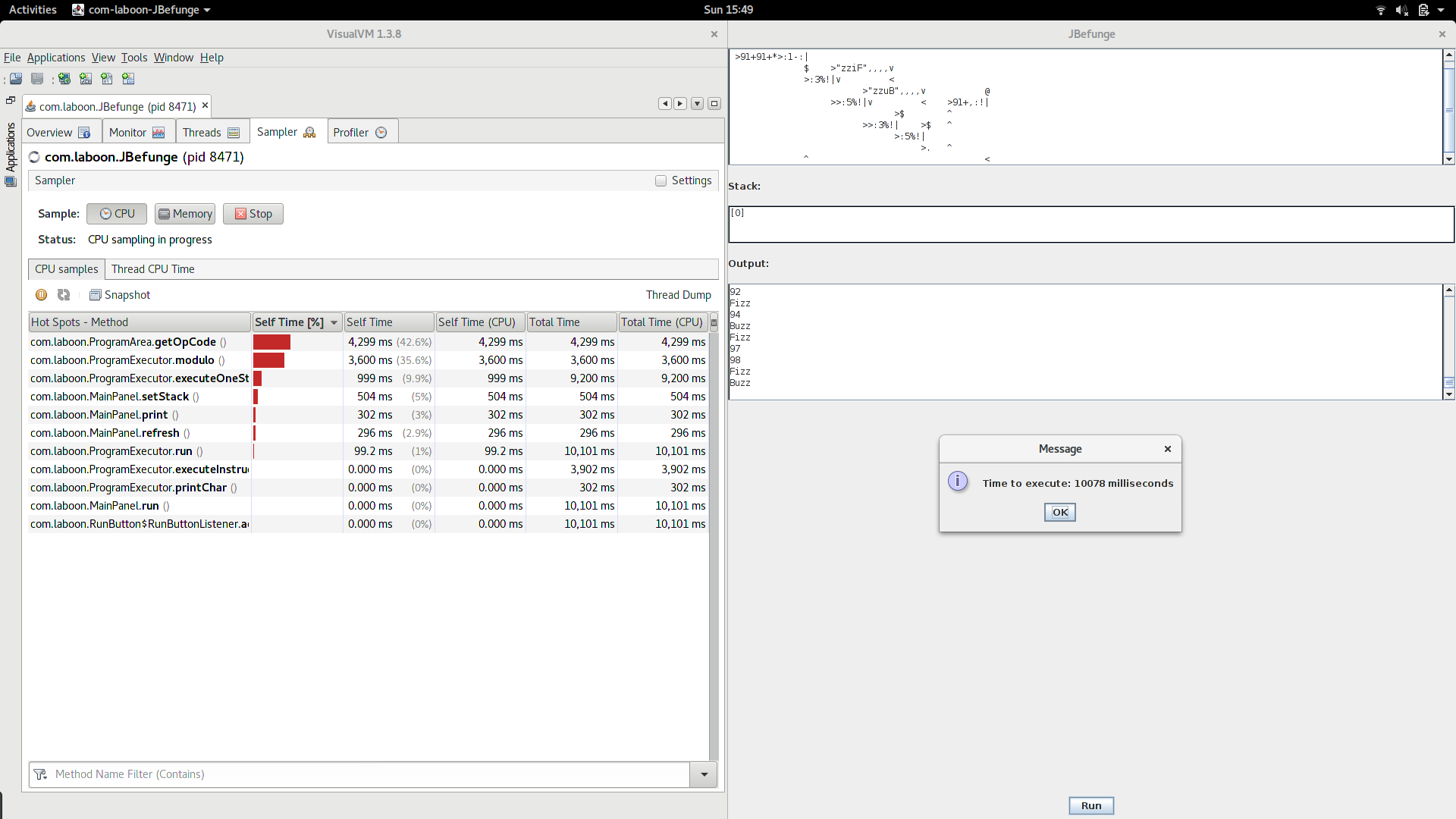
**Before Refactoring.**



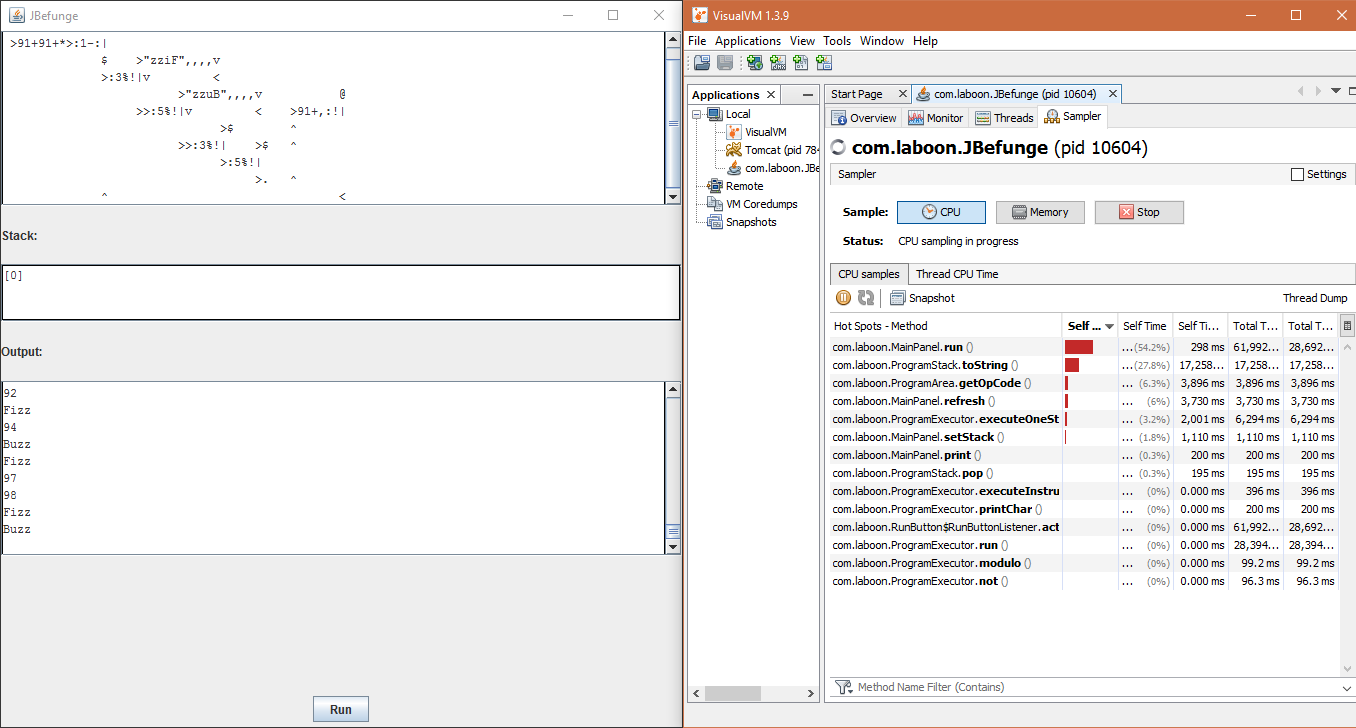
**After Get Op Refactor**



**After ToString Refactor**



**After Modulus Refactor**



**After All Refactors**

