JavaLife

Therese Dachille

CS 1632 – DELIVERBLE 5: Performance Testing

Conway’s Game of Life

https://github.com/tcd12/SlowLifeGUI

In working on this deliverable, I utilized a combination of exploratory testing and profiling through VisualVM to determine which methods were taking the longest to run in this program and/or were written the most inefficiently. The first method I came across was the convertToInt() method in the MainPanel class. I found this through the VisualVM profiler, as it was the method using the most time to execute. Upon further inspection, I realized that this method was incredibly inefficient, as it was taking an integer and then returning it after “converting” it to an integer. The method had an unnecessary loop in it, similar to the one I next discovered in the runContinuous() method, that did nothing but add to a variable 1000 times, despite the updated variable never being used. Lastly, I found that initializing new Cell objects was taking a lot of time, so I investigated the Cell class, and found that its toString() method also included unnecessary code (yet again, another unwarranted loop). After fixing these errors, the speed (performance) of the program definitively improved.

I created three to four unit tests for each refactored method, with only the test for negatives in convertToInt() including asserting that my refactored code *not* equal the original. This is because the original version of the method does not handle negative numbers, and could lead to an error because of it. In contrast, my refactored method still returns a negative number, but also includes a printed error to warn that a negative number has been passed, which the original code fails to do. I chose to test negative numbers, MAX\_VALUE, zero, and base case numbers between 1 and 100 for this method, in order to be as thorough with base cases and with edge cases as possible.

Testing the runContinuous() methods was definitely the hardest part of the deliverable, as the method itself is void; this makes it difficult to test output against other output, which is why I opted to create manual tests for this method, as listed below.

IDENTIFIER: runContinuousTest

TEST CASE: PulsarPatternTest

PRECONDITIONS: See Figure 1

INPUT VALUES: 17 (board size)

EXECUTION STEPS: Click <Run Continuous>, then <Stop> to see correct pattern, matching that specified by Pulsar pattern description

OUTPUT VALUES:n/a

POSTCONDITIONS: See Figures 2, 3

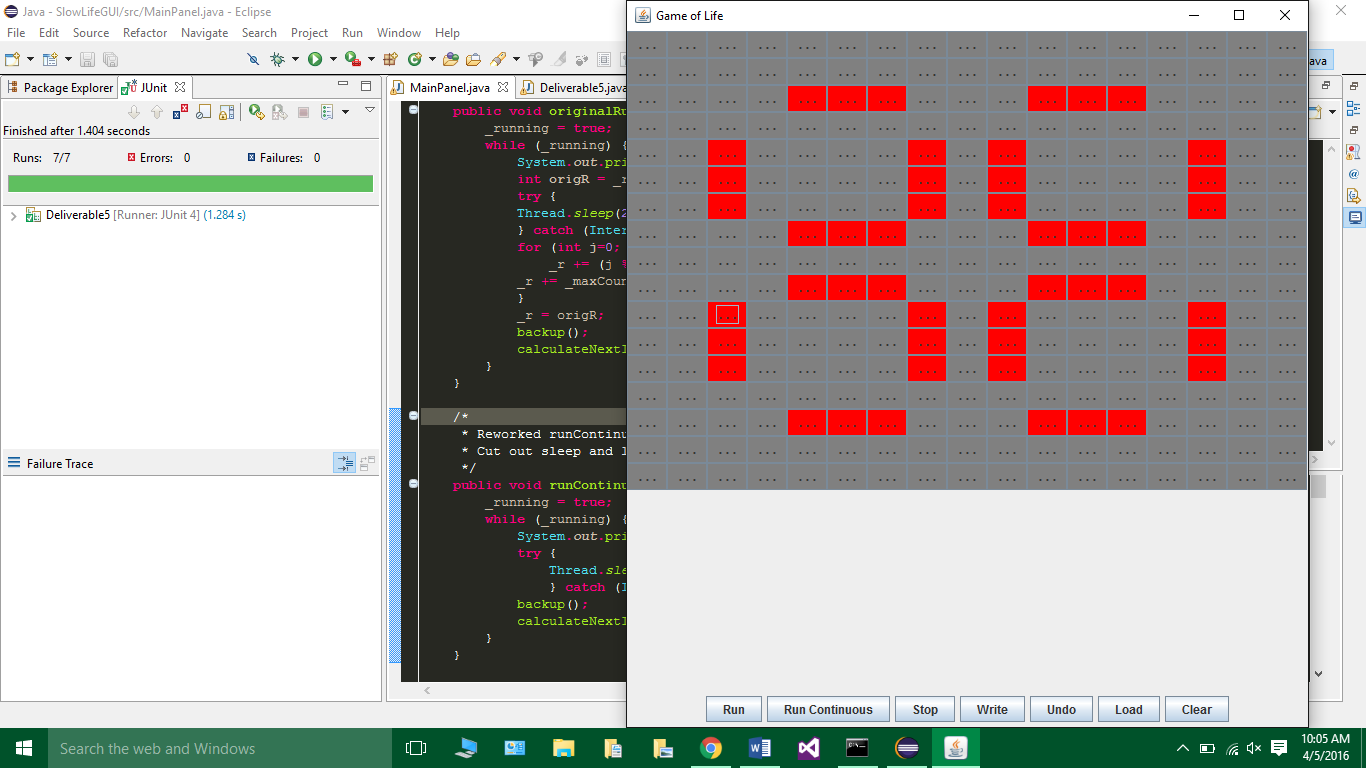


Figure 1

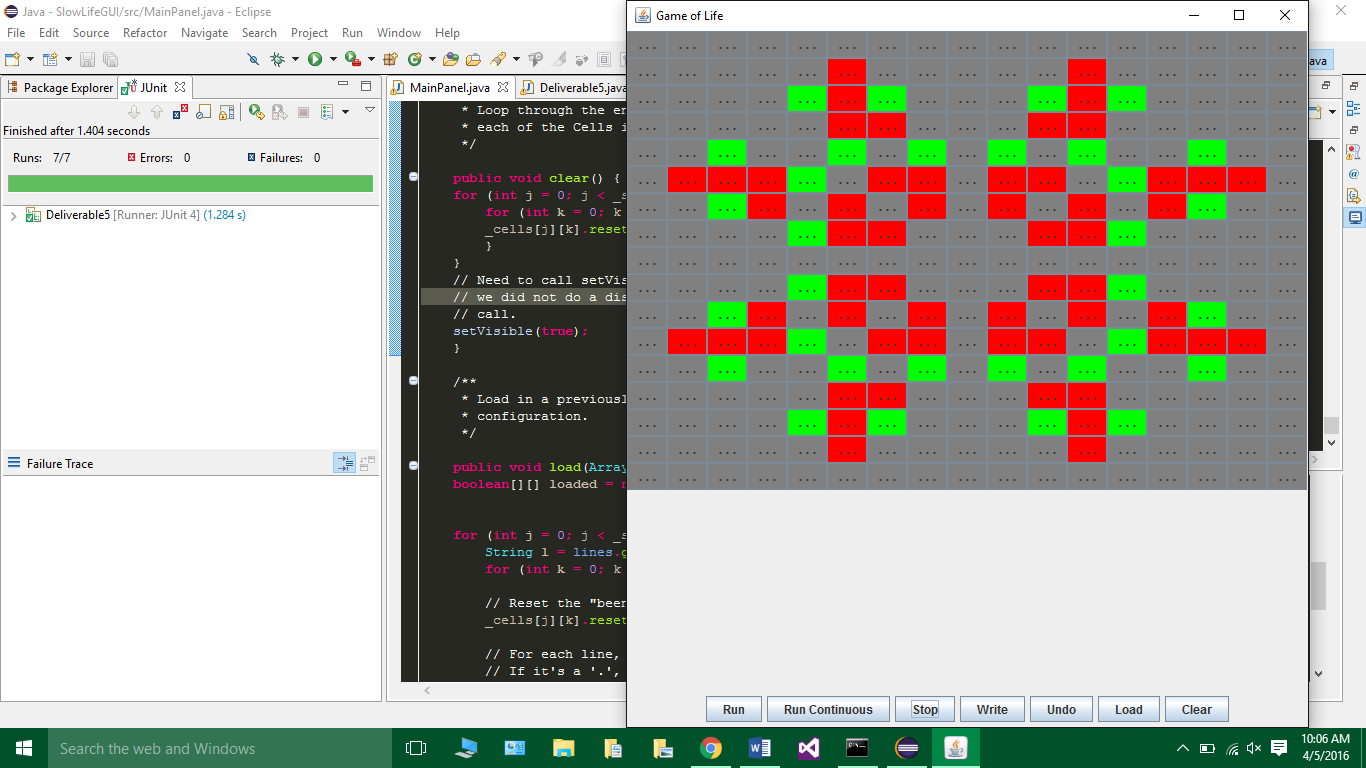


Figure 2

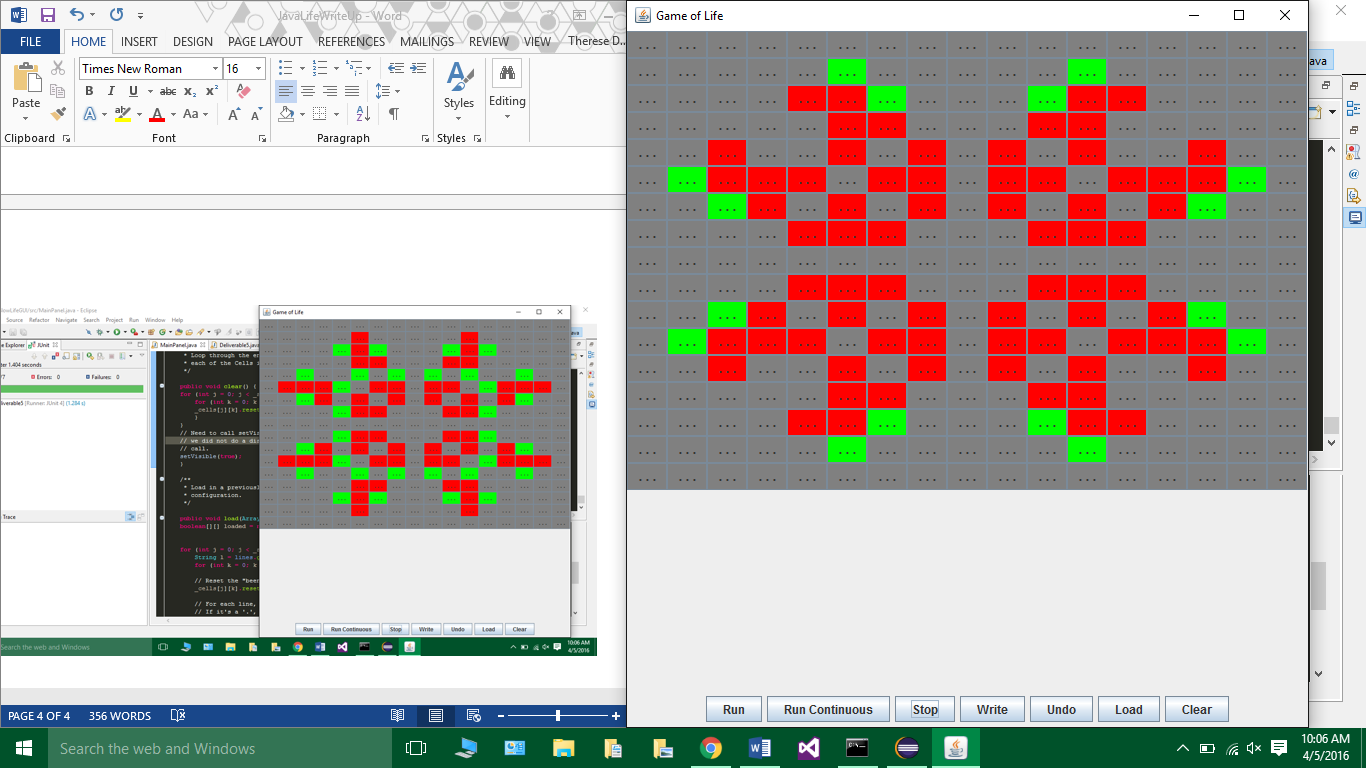


Figure 3

IDENTIFIER: runContinuousTest

TEST CASE: BlinkerPatternTest

PRECONDITIONS: See Figure 4

INPUT VALUES: 5 (board size)

EXECUTION STEPS: Click <Run Continuous>, then <Stop> to see correct pattern, matching that specified by Blinker pattern description

OUTPUT VALUES:n/a

POSTCONDITIONS: See Figure 5 IDENTIFIER: runContinuousTest

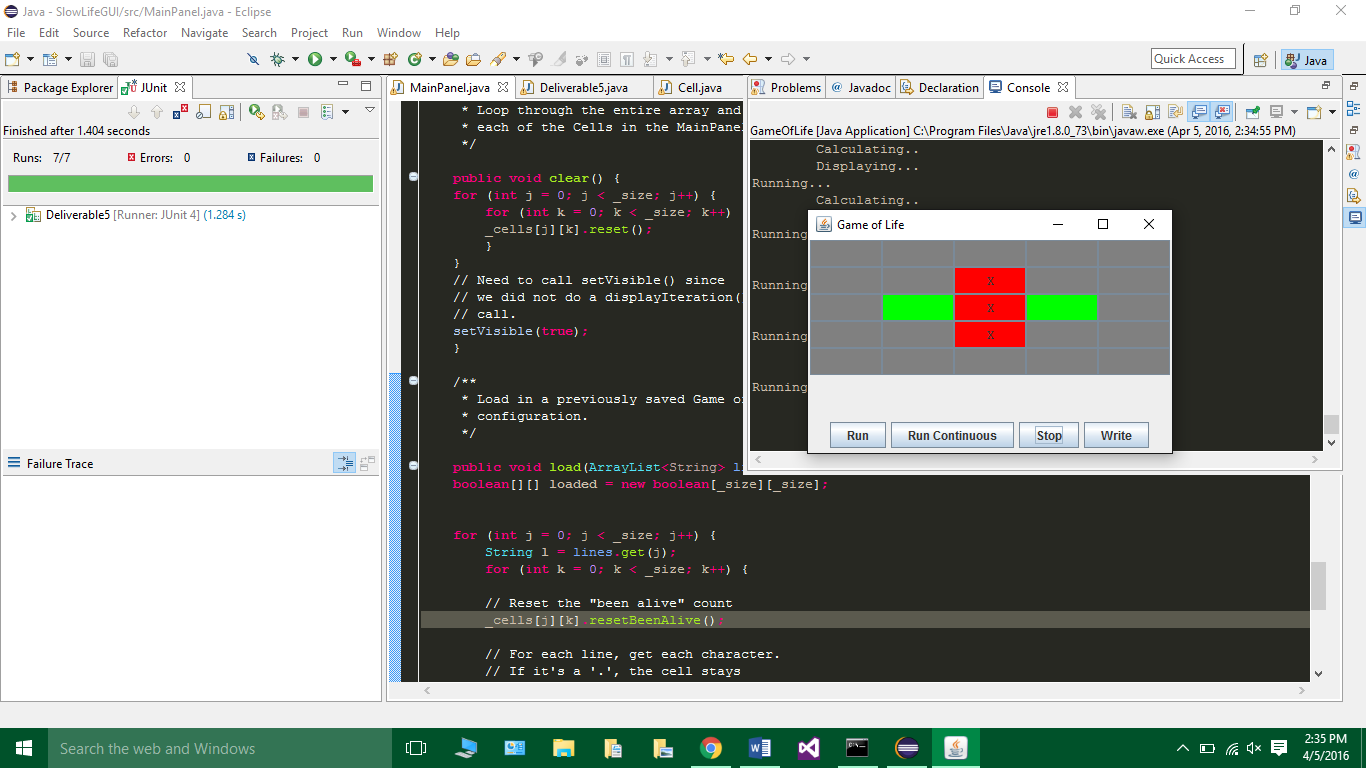


Figure 4

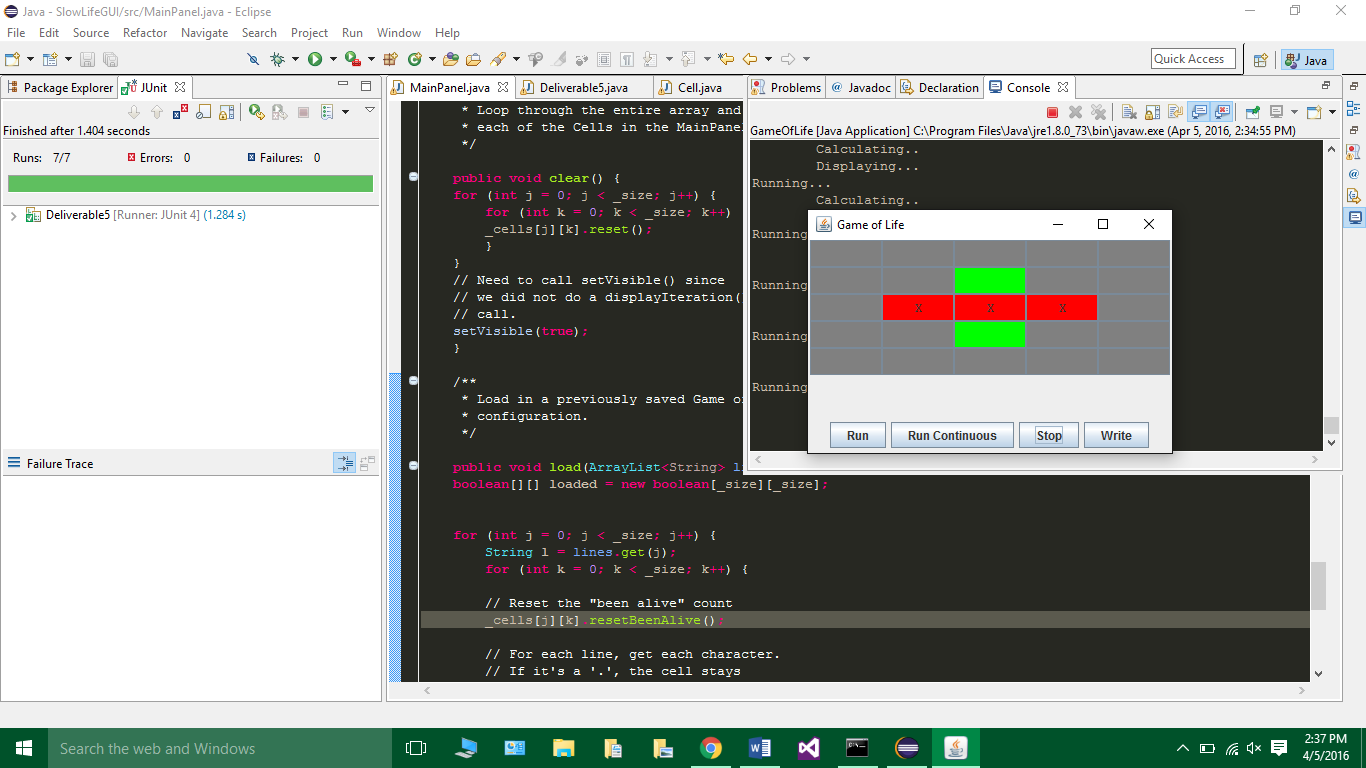


Figure 5

IDENTIFIER: runContinuousTest

TEST CASE: ToadPatternTest

PRECONDITIONS: See Figure 6

INPUT VALUES: 6 (board size)

EXECUTION STEPS: Click <Run Continuous>, then <Stop> to see correct pattern, matching that specified by Toad pattern description

OUTPUT VALUES:n/a

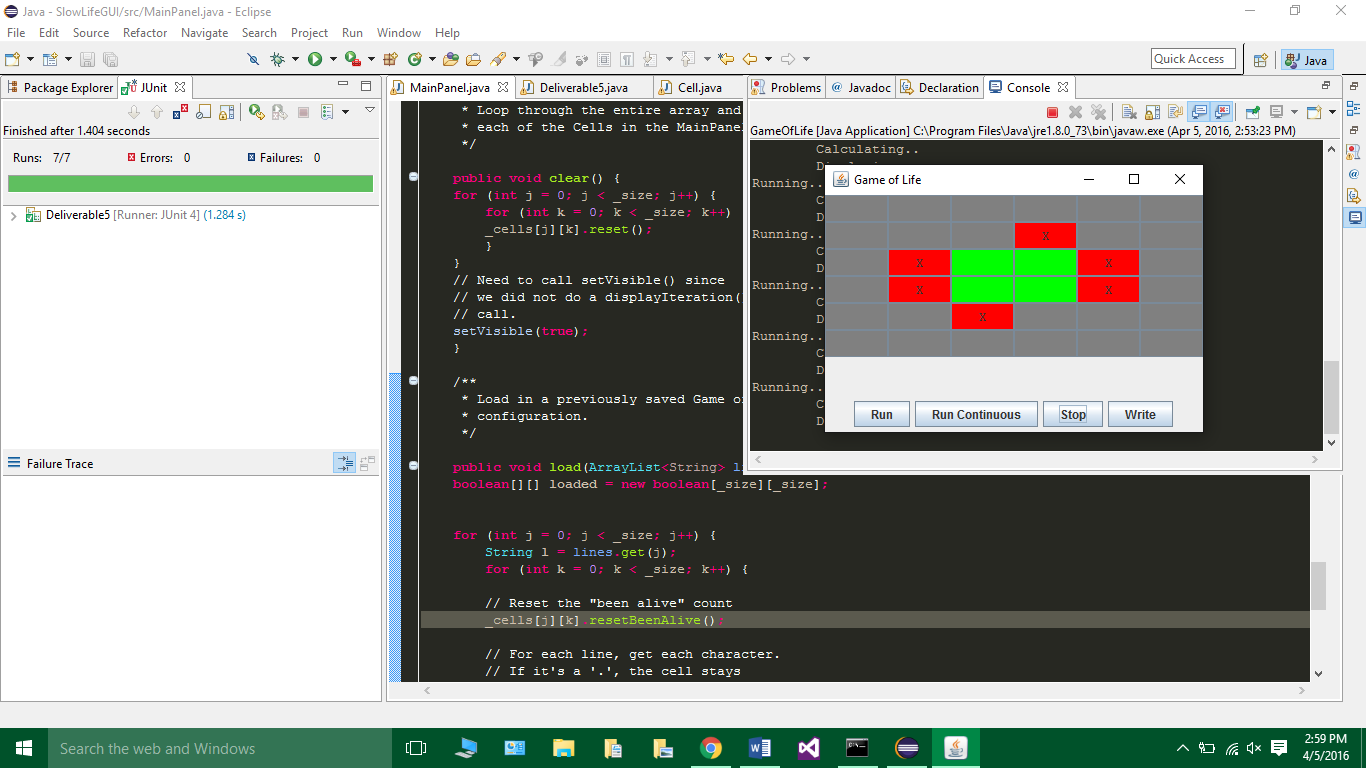
POSTCONDITIONS: See Figure 7

Figure 7

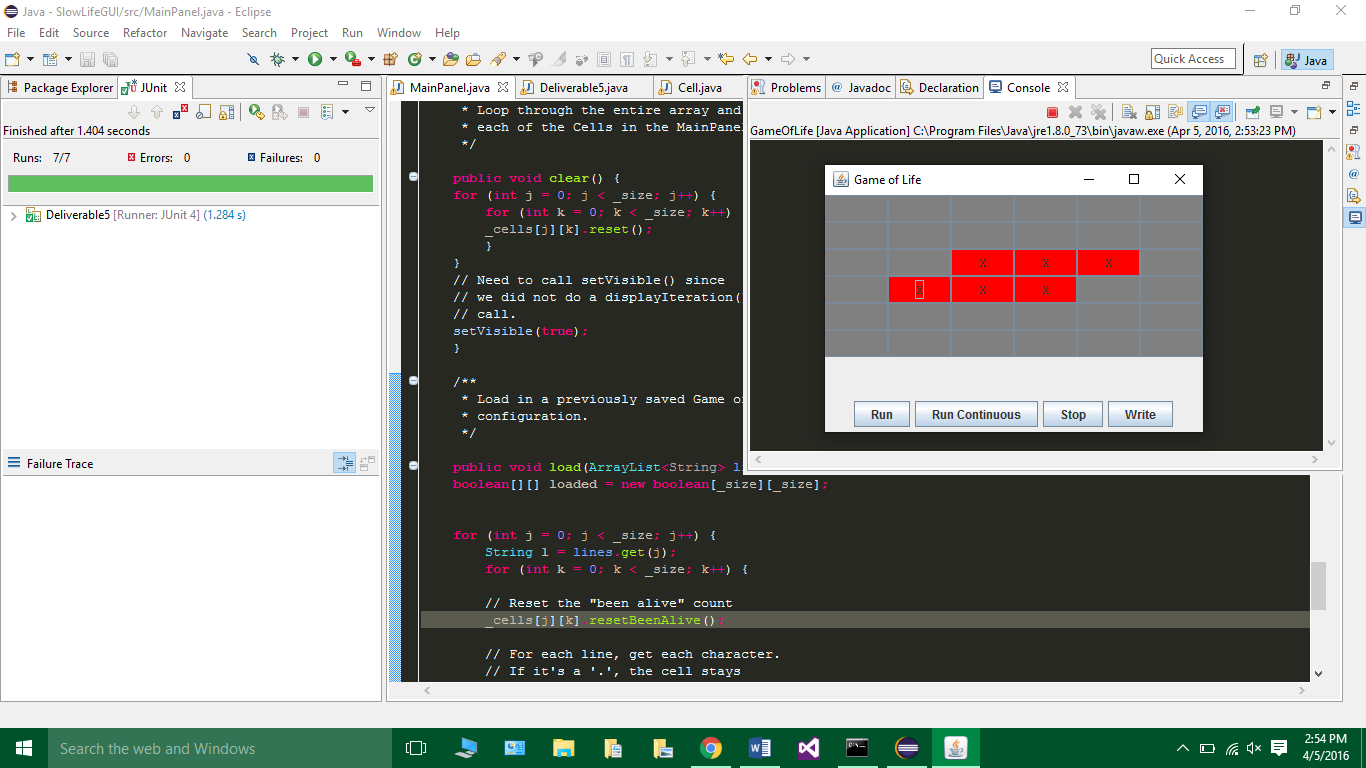
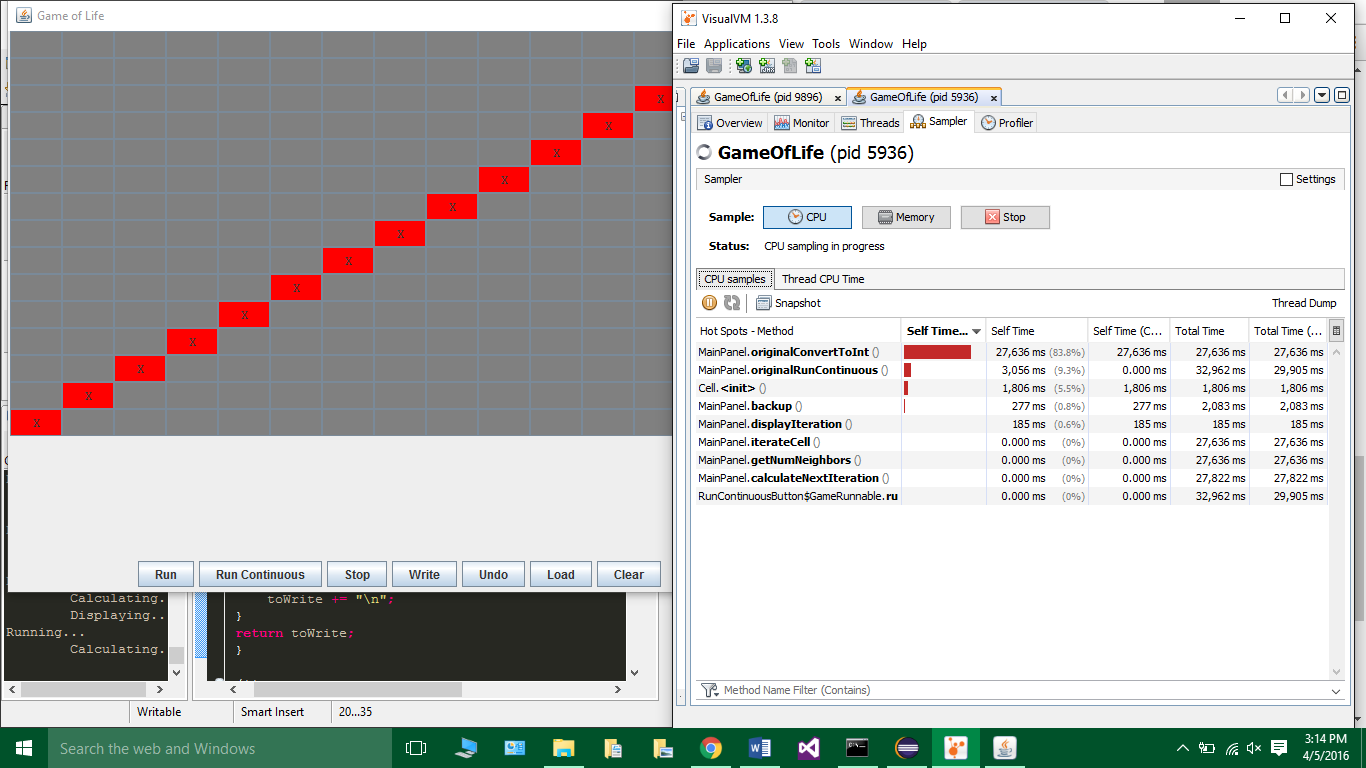
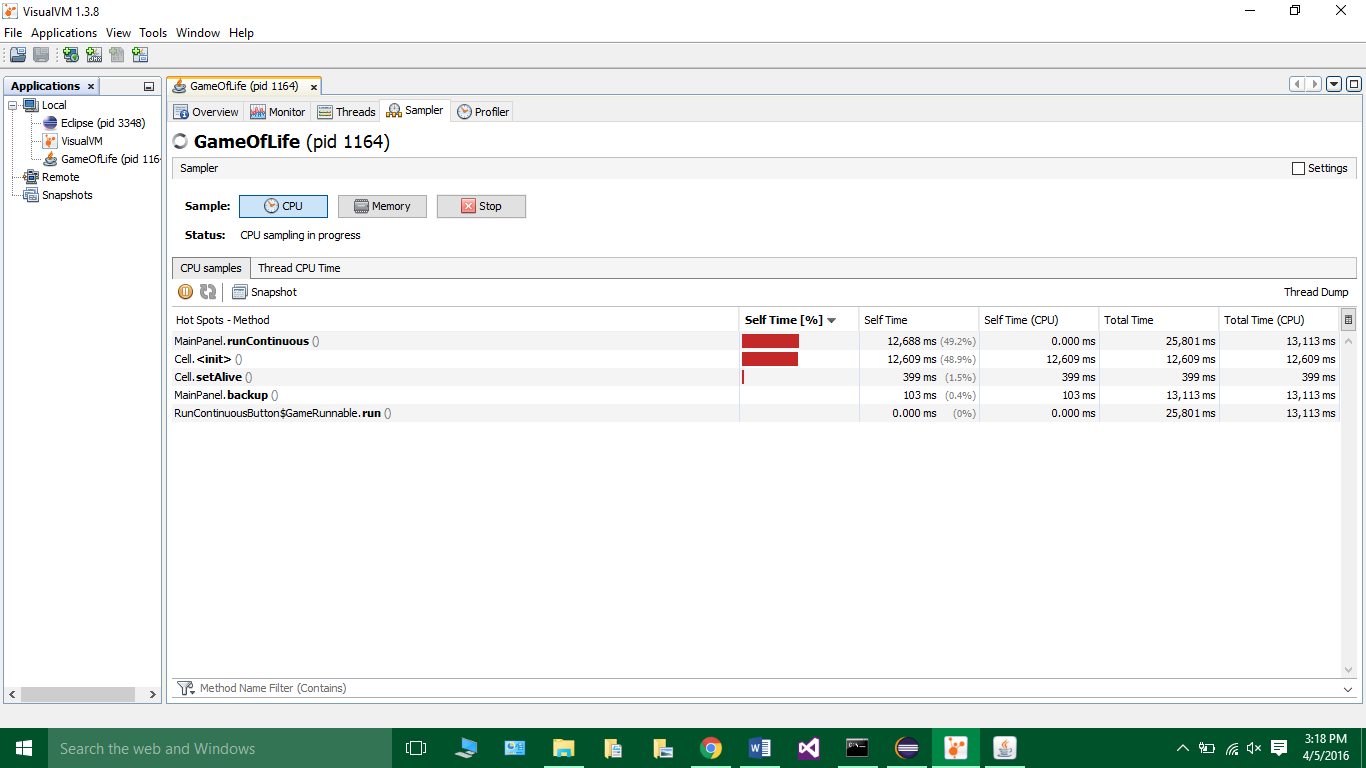


Figure 6

Figure 6

Slow Methods: [originalRunContinuous(), originalToString(), originalConvertToInt()]

Refactored Methods: [convertToInt(), runContinuous(), Cell.toString()]