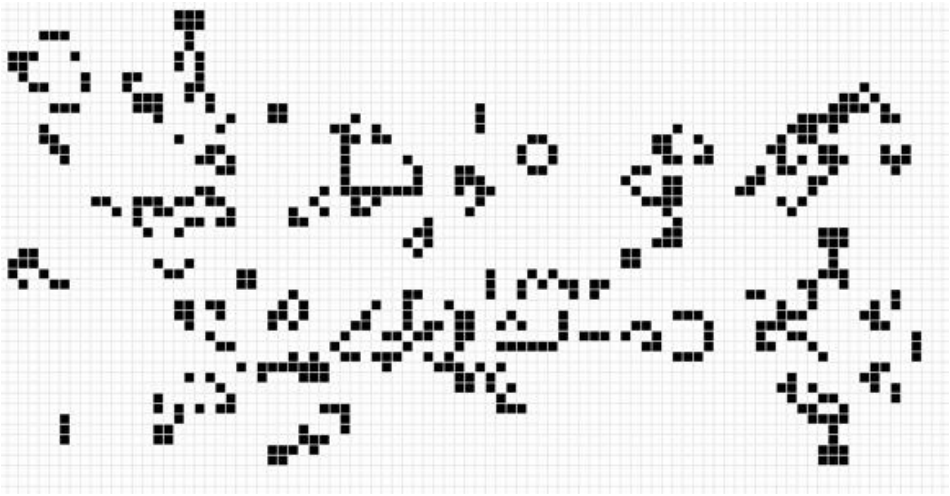
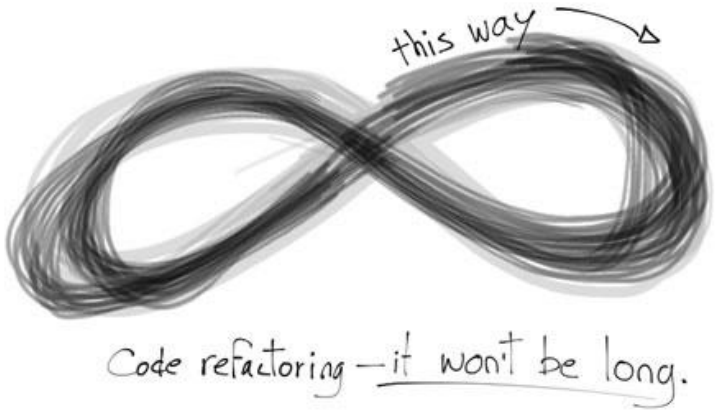


INFSCI 2545 Software Quality Assurance
Deliverable #5
Performance Testing Conway's Game of Life

Joseph Bender

jdb117@pitt.edu



How application was profiled:

I used the application VisualVM as a profiler to analyze the performance of the SlowLife application. When the program is run in eclipse, I can then sample the load put on the CPU by the Java application running. It will provide me with how much time it took each method to run, and what percentage of the total time it took. This is critical information in comparing which methods were the most taxing to the processing power. I started refactoring with the slowest methods, and quickly found errors that were increasing CPU consumption.

Method selected for refactoring:

1. The method that was taking up the most processing time and power in the initial program was MainPanel.convertToInt(). After some experimental testing of the method, I was puzzled by its actual functionality. Basically what it did is create a string of 1000 zeros, and then append the passed in argument to it. Therefore, the returned integer was identical to the argument. It iterated through a for loop 1000 times for absolutely nothing! This was obviously bogging down the application. I modified the method to simply return the variable that was already an integer.
2. Another method taking up a lot of processing power was Cell.toString(). This method is accessed through MainPanel.toString(). The Cell object has a variable called _maxSize that is assigned the value of 10000. This value is then used to control a for loop inside of Cell.toString(). It uselessly duplicated the current cell state thousands of times. However, the only thing returned was the first substring of this new long string. I removed this for loop. Therefore, if the cell was alive it returns the contained text “X”. If it was dead, it returns “.”. These characters are then inserted into a larger string in MainPanel.toString(), and saved for future use.
3. The MainPanel.runContinuous() method was also a heavy load on the CPU. Obviously, it is a looping method that performs a primary action in the program, so it is going to take up processing power regardless. However, with closer examination it had a lot of unneccesary code! It had a Thread.sleep(20) call that executed every time the method was performed. In the long run this could really slow down the functionality of the game. It also had an arbitrary for loop that iterated 10000 times. Both of these contributed to MainPanel.runContinuous() needing major refactoring.

Why I changed what I did:

1. I had to change the MainPanel.convertToInt() method to a public scope in order to unit test it. I also had to change the _r variable to static in order to test it using junit. This variable held the integer value of 1000, and controlled the for loop in the MainPanel.convertToInt() method. It was making the for loop iterate 1000 times for no reason. I then removed all the code in MainPanel.converToInt(), and simply had it return the argument that was passed in with zero functionality. The program still functions as it did originally, just a lot faster.
2. In Cell.toString, there was a pointless for loop. The method is supposed to return “X” if the cell is alive, and “.” if the cell is dead. The loop would get the text within a cell, then needlessly iterate 10000 times appending the cell text onto a string each iteration. However, when it was time to return a string, it would use substring to only return the first character of this new super long string. This made the loop perform an operation and take up CPU power for no reason! I did

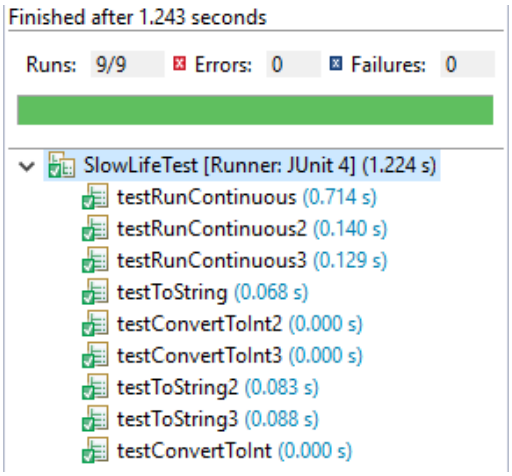
away with the for loop, simply returning "." if the cell was dead, and returning the text contained within a cell if it was alive (which is "X").

3. In MainPanel.runContinuous(), I eliminated the Thread.sleep() call in order to stop the delay it contributed. I also deleted the for loop that iterated 10,000 times. All that was really needed in this method is the calculateNextIteration() to be looped. I made this call the primary functionality of the method and removed the rest.

Pinning Test Implementation:

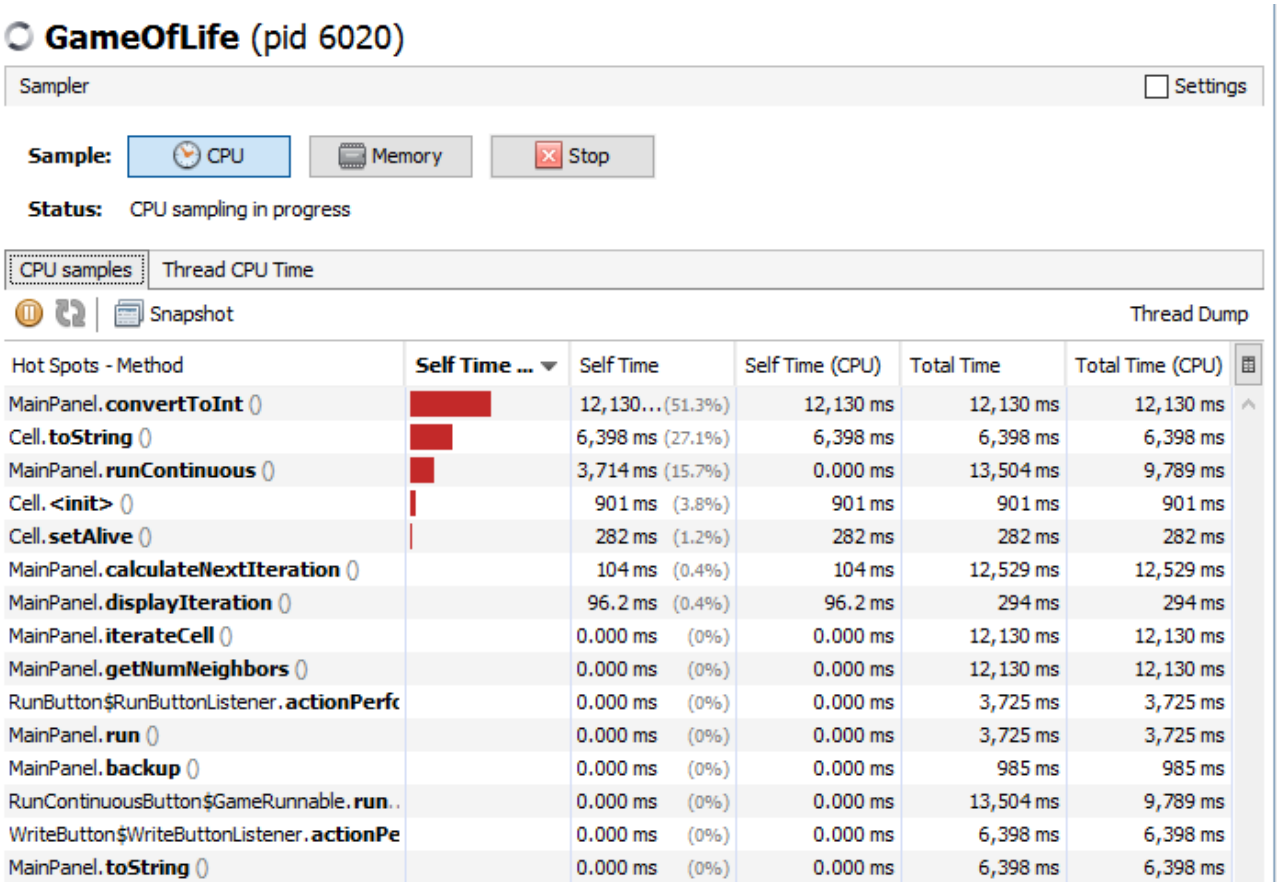
1. My first set of three pinning tests (testConvertToInt(), testConvertToInt2(), and testConvertToInt3()) ensure that the value of the argument passed into convertToInt() is the same value of the variable returned to the call. The method was arbitrary, and its functionality was pointless. This test ensures that after my modification of the method, the functionality was the same.
2. My second set of three pinning tests (testToString(), testToString2(), and testToString3()) ensure that when the current state of a game is saved and loaded, it presents the same grid to the user. I removed a for loop from the Cell.toString() method, and simply returned a single character instead of a substring of a massive string. These tests populate the grid with fake data, write to the grid, and then reload it. If the reloaded data is equal to the original fake data, then the Cell.toString() method's functionality is unaltered by my refactor!
3. The third set of three tests (testRunContinuous(), testRunContinuous2(), and testRunContinuous3()) ensure that the functionality of the "Run Continuous" button remained the same after refactoring. To test this, I simulate a predictable continuous run of the game that ends in a dead-end. I first load predictable fake data into the grid. Then, I simulate the runContinuous() method by calling calculateNextIteration() method 100 times in a loop. Now, after that many iterations it should always arrive at the predictable dead end configuration. I have the end configuration already saved in three text files, so I load those and compare them to the end result in the game. If they are exactly the same, the tests pass and prove that the runContinuous() method is still fully functional.

Screenshot of Junit Tests:



Screenshots of VisualVM (Before and After Refactoring):

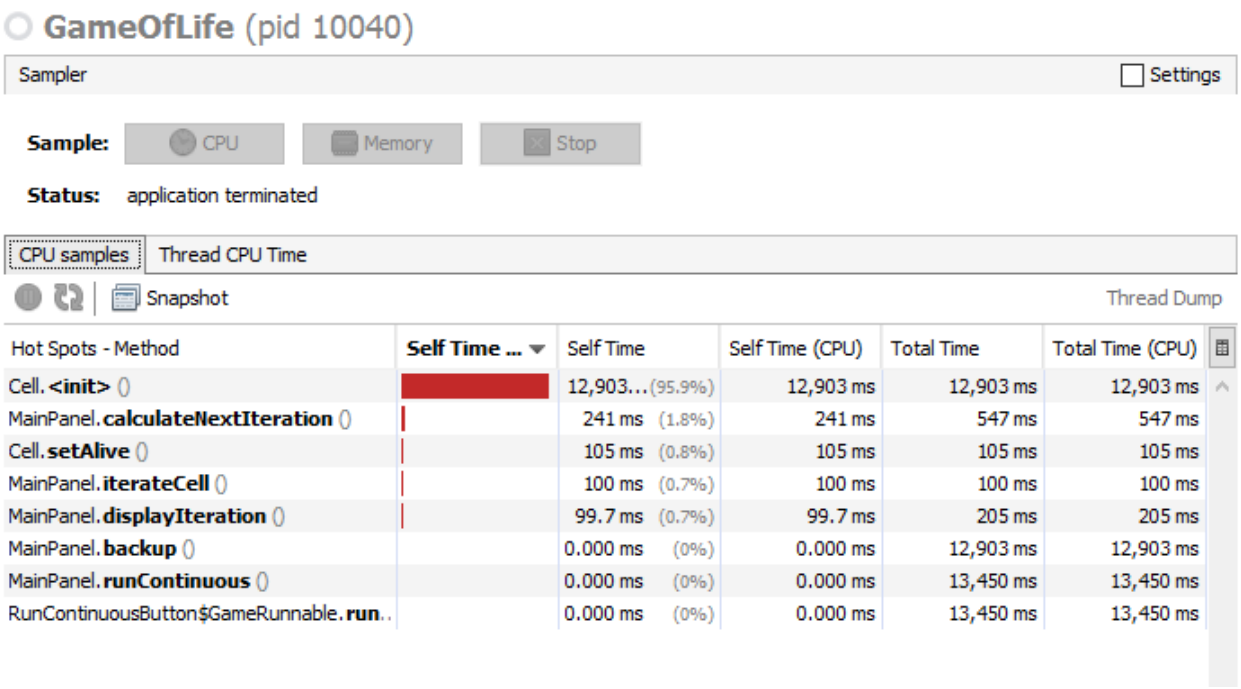
Before Refactoring:



Hot Spots - Method	Self Time ...	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
MainPanel.**convertToInt** ()		12,130... (51.3%)	12,130 ms	12,130 ms	12,130 ms
Cell.**toString** ()		6,398 ms (27.1%)	6,398 ms	6,398 ms	6,398 ms
MainPanel.**runContinuous** ()		3,714 ms (15.7%)	0.000 ms	13,504 ms	9,789 ms
Cell.**<init>** ()		901 ms (3.8%)	901 ms	901 ms	901 ms
Cell.**setAlive** ()		282 ms (1.2%)	282 ms	282 ms	282 ms
MainPanel.**calculateNextIteration** ()		104 ms (0.4%)	104 ms	12,529 ms	12,529 ms
MainPanel.**displayIteration** ()		96.2 ms (0.4%)	96.2 ms	294 ms	294 ms
MainPanel.**iterateCell** ()		0.000 ms (0%)	0.000 ms	12,130 ms	12,130 ms
MainPanel.**getNumNeighbors** ()		0.000 ms (0%)	0.000 ms	12,130 ms	12,130 ms
RunButton$RunButtonListener.**actionPerfc**		0.000 ms (0%)	0.000 ms	3,725 ms	3,725 ms
MainPanel.**run** ()		0.000 ms (0%)	0.000 ms	3,725 ms	3,725 ms
MainPanel.**backup** ()		0.000 ms (0%)	0.000 ms	985 ms	985 ms
RunContinuousButton$GameRunnable.**run**..		0.000 ms (0%)	0.000 ms	13,504 ms	9,789 ms
WriteButton$WriteButtonListener.**actionPe**		0.000 ms (0%)	0.000 ms	6,398 ms	6,398 ms
MainPanel.**toString** ()		0.000 ms (0%)	0.000 ms	6,398 ms	6,398 ms

MainPanel.convertToInt(), Cell.toString(), and MainPanel.runContinuous() are the three most taxing methods. That is why I chose to refactor those three methods.

After:



Hot Spots - Method	Self Time ...	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
Cell.**<init>** ()		12,903... (95.9%)	12,903 ms	12,903 ms	12,903 ms
MainPanel.**calculateNextIteration** ()		241 ms (1.8%)	241 ms	547 ms	547 ms
Cell.**setAlive** ()		105 ms (0.8%)	105 ms	105 ms	105 ms
MainPanel.**iterateCell** ()		100 ms (0.7%)	100 ms	100 ms	100 ms
MainPanel.**displayIteration** ()		99.7 ms (0.7%)	99.7 ms	205 ms	205 ms
MainPanel.**backup** ()		0.000 ms (0%)	0.000 ms	12,903 ms	12,903 ms
MainPanel.**runContinuous** ()		0.000 ms (0%)	0.000 ms	13,450 ms	13,450 ms
RunContinuousButton$GameRunnable.**run**...		0.000 ms (0%)	0.000 ms	13,450 ms	13,450 ms

After my refactoring, the three most CPU intensive methods are now barely using the processing power. Most of the time spent processing is on initializing the cells in the game, which is how it should be considering that is the primary function of the program.

Repository Link:

https://github.com/jdbender66/SlowLifeGUI