**CS1699: Software Testing**

**Ryan McDonald**

**Deliverable 4:**

**Performance Testing Conway’s Game of Life**

While I had painfully vivid nightmares prior to looking deeper at this project about what eclipse plugins I may have to install and set up in order to get started, after a few minutes of googling, I realized I already had the VisualVM profiler installed on my system and all I needed to do was run the executable. Before tinkering around I tried browsing around in the code to see if I could notice anything particularly inefficient. Unfortunately, my novice eyes were no match for the inefficient code lurking about. Thus, I had to put on my VisualVM goggles and dive into the profiling.

Taking a bit of time to get comfortable with the mechanics of how VisualVM worked, I soon discovered where to look to see what java methods were hogging up all of my processing power and thereby causing the aforementioned inefficiency within the Java code. I noticed that the World Class’s toString() method was our culprit. After running a really obscene amount of iterations of the Game Of Life, the toString() method accounted for close to 100% of the Self Time while all of the other hot spot methods accounted for nearly nothing.

Once I had found the treasure of inefficient code, it was time to embark on a quest in order to determine just how severe this issue was. While looking at the code, I’ll admit, our culprit hid well. In Java I to will be lazy and concatenate Strings at the tip of a hat but then it hit me… That’s so inefficient! The abstracted tentacles of hell that piece together different Strings during the many concatenations throughout the method spawn a little Satan in the program. When Java concatenates Strings with the “+=” operator, it has to build a whole new String in order to make this possible due to Java Strings being immutable which adds quite a bit of overhead to the already lengthy operation. Therefore, I decided to refactor this method so that less String objects would be instantiated.

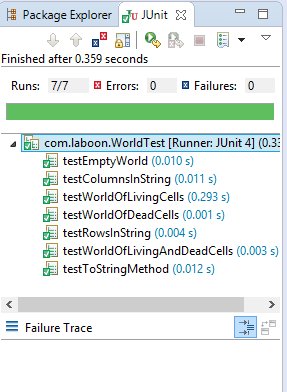
In order to refactor the method, I decided to use the StringBuilder class. The StringBuilder class has the capabilities to store a mutable sequence of characters. So rather creating multiple String Objects, using the StringBuilder class lets us append onto the current character sequence which it is holding. We can then build our String using one Object and then, when we are done, we can turn it into a String object and return it to the caller!

Upon realizing how I can refactor the method, I then went about writing test cases to verify tidbits of the toString() method as so I could verify after the modifications that my changes did not break anything. While figuring out edge cases took a bit of thought I easily realized that I could test three different types of worlds: all living, all dead and a mixture of the two. Once those were out of the way, I thought a little harder about edge cases and noticed that, when the game board is larger than 10, the column and row indicators restart to 0 and increase once more. Thus, I decided to test that with the added bonus of testing that the rows and columns are indexed in general. I also figured I could verify the String of an empty game board to make sure all instances of creating a game board would work appropriately.

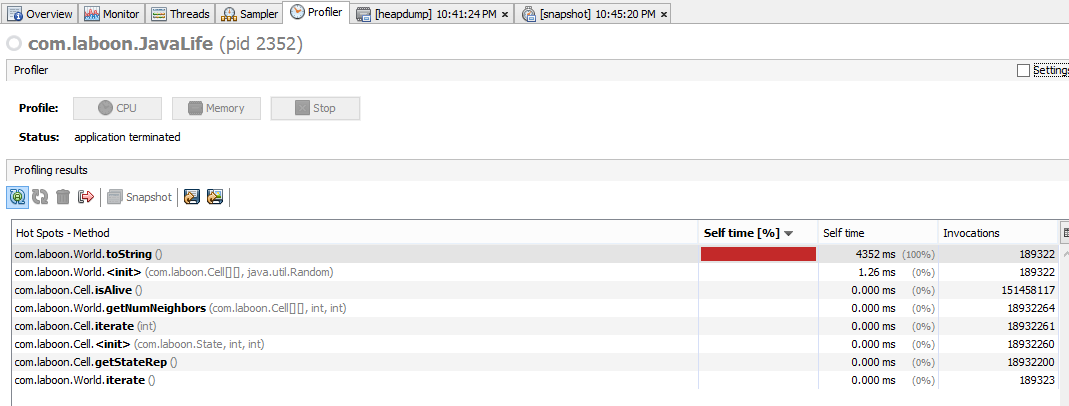
The real difficulty in all of this came from using the profiler. While I managed to refactor the method in such a way that even Stackoverflow agrees with, I feel as though I could not get the profiler to show that adequately enough. I decided to just take a before and after screenshot of the time it took to run with 200,000 invocations of the game. We can see that when I started profiling them both, I captured more invocations of the refactored toString() method and yet the time is still much less than the time of the non-refactored toString() method.

To conclude my journey full of refactoring obstacles and creeping hard on my CPU performance via Profiling, I am now much more aware of the lurking dangers of non-optimal code. Thus, I pledge to do my best to overcome laziness and write code so optimal that it could withstand even the inevitable heat death of the universe.

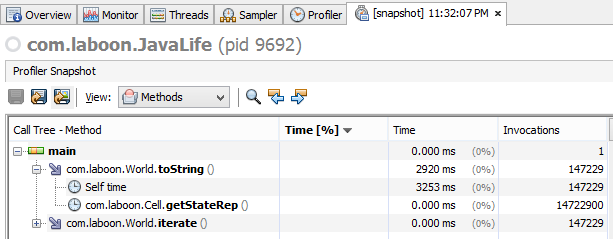
**Junit Tests: Passed Before and After**



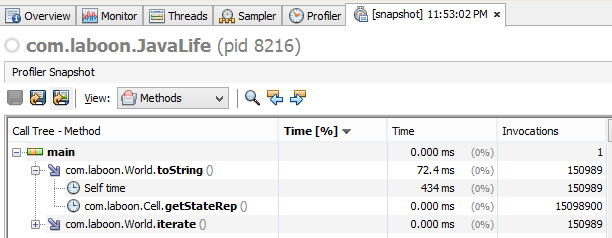
**VisualVM: Snapshot of large % of toString() Self time provoking refactoring**

****

**VisualVM: Snapshot Before Refactoring**



**VisualVM: Snapshot After Refactoring**

****