**CS 1699 - DELIVERABLE 4: Performance Testing Conway's Game of Life**

**for JavaLife**

Mike Byrne

mjb187@pitt.edu

https://github.com/mjb187/JavaLife/tree/mjb187\_deliverable4

CS 1699: Software Testing

Bill Laboon

March 31st, 2015

**SUMMARY**

**Profiling Process**

For profiling JavaLife, I used the VisulaVM application that’s included in the Java SDK. I opened the program in Eclipse, set the arguments to 60,000 iterations, hit run, began sampling and profiling as soon as VisualVM recognized the running JavaLife program, and then allowed the program to run to completion. To ensure accurate results, the modified JavaLife and original JavaLifeOrig programs were run sequentially under the exact same conditions.

To decide what to refactor, I looked at the CPU time by method, the VisualVm hotspots via class, and the heap data for the program. I recognized from the output that the World.toString() method was taking almost as much time as the JavaLife main method and significantly more time than the World.iterate() method. The profiler revealed that there were a significant amount of Hashtable and ArrayList operations, and the used heap space was growing and shrinking fairly often. After glancing at the code, this information led me to believe that the World.toString() method was creating the string representation of the World naively and could thus be improved.

**Refactoring**

Looking at the code, I noticed that both World.generateBoard() and World.iterate() go through and touch each Cell object in the World. World.toString() also iterated through each Cell object, and the World object itself is not updated on iterations since it returns a new World object each time. I realized that the iterating through the Cell objects in the World.toString() method was superfluous, as the work done in this function could be merged into the functions that create the array of Cell objects. Thus, I decided to have the String representation of the World be a class-level variable that is generated in the loops within World.generateBoard() and World.iterate() in order to reduce touching each Cell from twice to just once. This involved modifying one of the constructors as well in order to take the String representation of the World as a parameter.

When this alone did not give significant performance improvement, I investigated further and realized that the problem most likely involved the multitude of String concatenations. For a 25x25 grid, there were 625 concatenations for the Cell object representations alone, plus the axis labels and other miscellaneous operations. Considering that there were 60,000 sets of at least 625 String manipulations, I did some research into handling large Strings. It was through this research that I found the StringBuilder class, which is supposedly significantly more efficient at handling many small String manipulations on large String objects. Implementing the use of this class, combined with reducing the amount of loops through the array of Cell objects, significantly reduced the runtime of the program in these functions. Heap usage was significantly reduced as well.

The downside to my refactoring is that some of the time that was spent in the World.toString() function has been dispersed into the JavaLife main method and the World.iterate() function. I find this acceptable, because the total amount of time added to these two segments is still less than the amount of time originally spent in the World.toString() method before refactoring. Consult the “Results” section for a more detailed breakdown of this.

**Unit Testing**

For the unit tests, I assumed that all other functions were working correctly and that World.toString() was the only method that needed testing. Thus, I only tested the String representations of the World object instead of something like whether or not the simulation algorithm was correct or whether or not the program handles invalid size or percentage arguments. My unit tests are then designed around making sure the String representations meet certain conditions.

The unit tests included in this project test the following conditions:

* A World of size N should have N\*N ‘X’ or ‘.’ characters representing N\*N Cell objects.
* A World of size N and percentage value 0 should have N\*N ‘.’ characters representing N\*N “dead” Cell objects.
* A World of size N and percentage value 100 should have N\*N ‘X’ characters representing N\*N “alive” Cell objects.
* A World’s String representation after one iteration should differ from the initial String representation under “normal” conditions.
* A World of size 5, seed 1699, and percentage 30 should have " 01234\n0 X.X..\n1 .X...\n2 X....\n3 X.X..\n4 ..XX.\n" as its String representation initially.
* A World of size 5, seed 1699, and percentage 30 should have " 01234\n0 .....\n1 .....\n2 .....\n3 .....\n4 .....\n" as its String representation after 42 iterations.
* Two Worlds with identical size, seed, and percentage values should have the same String representations initially.
* Two Worlds with identical size, seed, and percentage values should have the same String representations after 42 iterations.
* Two Worlds with identical size and percentage values but differing seed values should have different String representations initially.
* Two Worlds with identical size and percentage values but differing seed values should have different String representations after 42 iterations.

**Results**

Arguments:

JavaLife 25 1699 35 60000

Before:

|  |  |
| --- | --- |
| **Method Name** | **Sampled Execution Time** |
| JavaLife.<init>() | 10,335 ms |
| World.toString() | 8,590 ms |
| World.iterate() | 399 ms |
| **Total** | 19,324 ms |

After:

|  |  |
| --- | --- |
| **Method Name** | **Sampled Execution Time** |
| JavaLife.<init>() | 11,268 ms |
| World.toString() | ~0 ms |
| World.iterate() | 1,496 ms |
| **Total** | ~12,764 ms |

**Challenges**

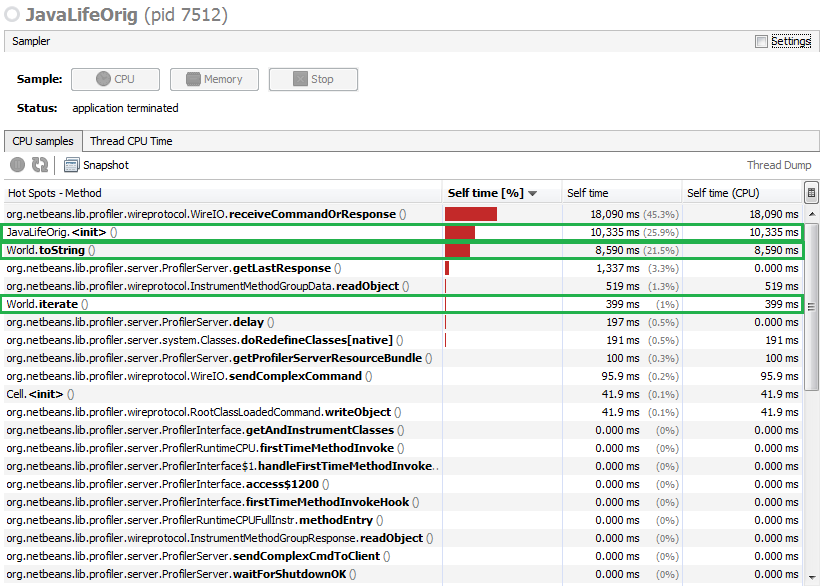
The most difficult part of the whole process for me was interpreting the profiler output and identifying a problem area. I don’t know what the expected conventions for a toString() method are, so my choice to refactor this method in particular came from the fact that it was the method that took the longest to execute. I figured a simple program like JavaLife should not take too terribly long to burn through 60,000 iterations. On top of this, the profile outputs a lot of information, of which I was not sure how to identify the useful bits among all the unnecessary data. For example, now that the refactoring is complete, I realize that the screenshot of the Hashtable and ArrayList durations are probably not necessary for understanding the impacts of my code revisions.

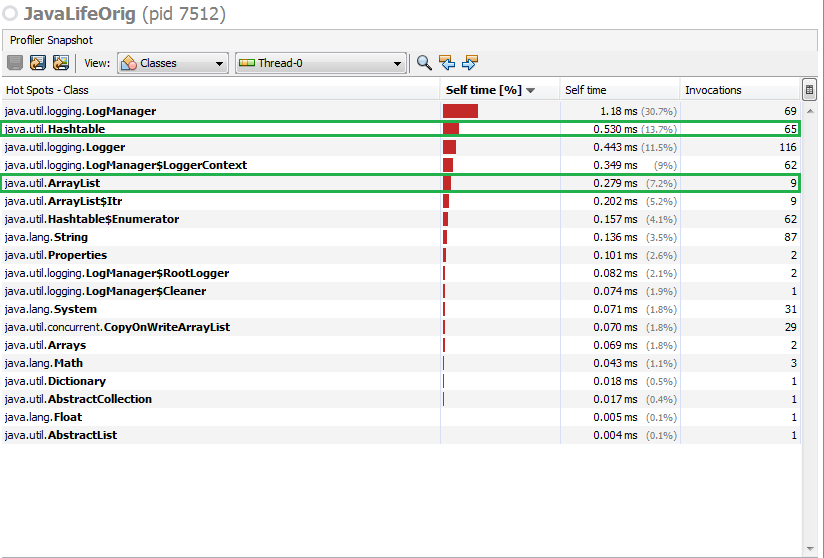
The other challenge faced in this project was deciding how to refactor the code. Since I was targeting the World.toString() method as the problematic method, I was not sure if that meant I should localize changes to that particular method or if I was allowed to modify other sections of the program. Ultimately, I decided on allowing edits to other methods, but I am unsure if this is the expected course of action. It seems almost as if I “refactored” multiple methods instead of the World.toString() method exclusively.

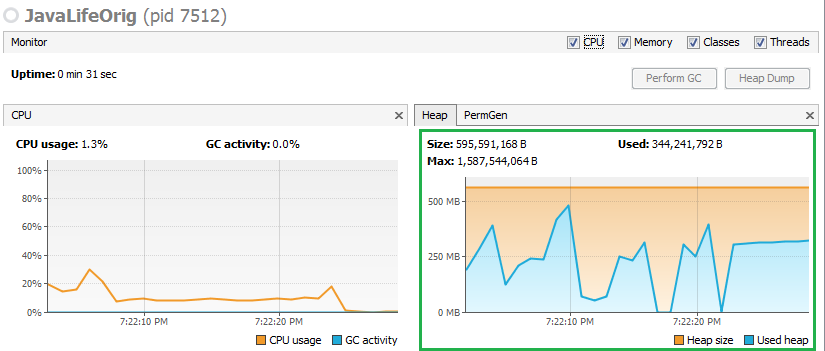
**PROFILER OUTPUT**

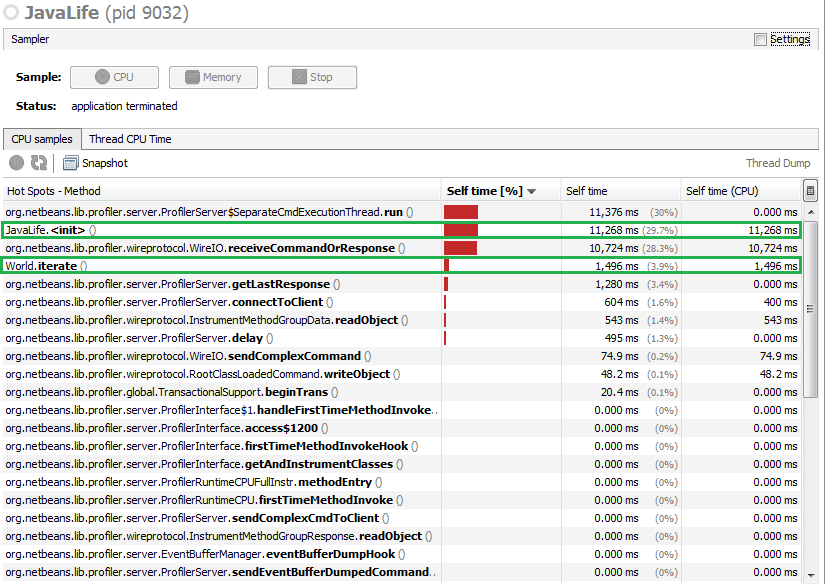
**Arguments**

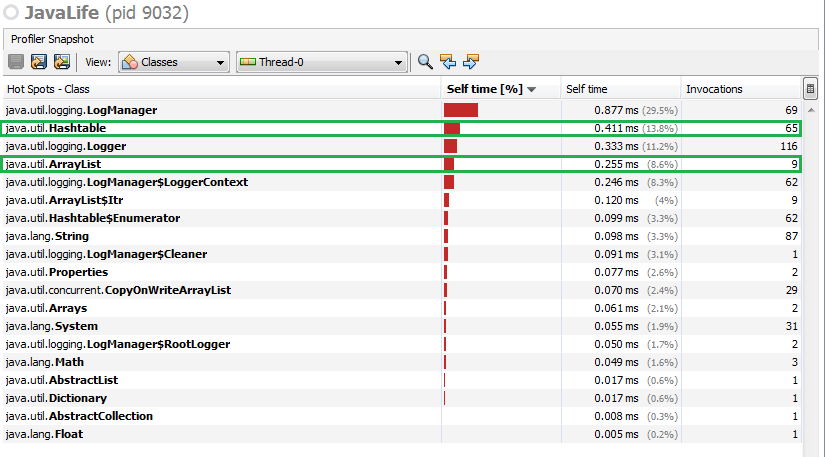
JavaLife 25 1699 35 60000

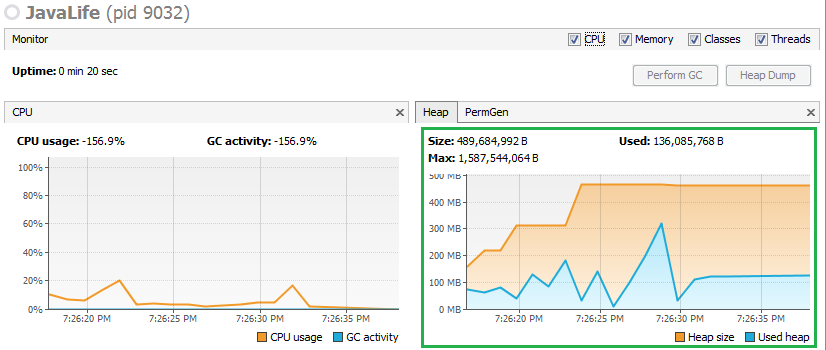
**Before**





**After**

****

****