Software Testing - Deliverable 4

Performance Testing - Conway's Game of Life and Java VisualVM

Jeff Warner - JavaLife

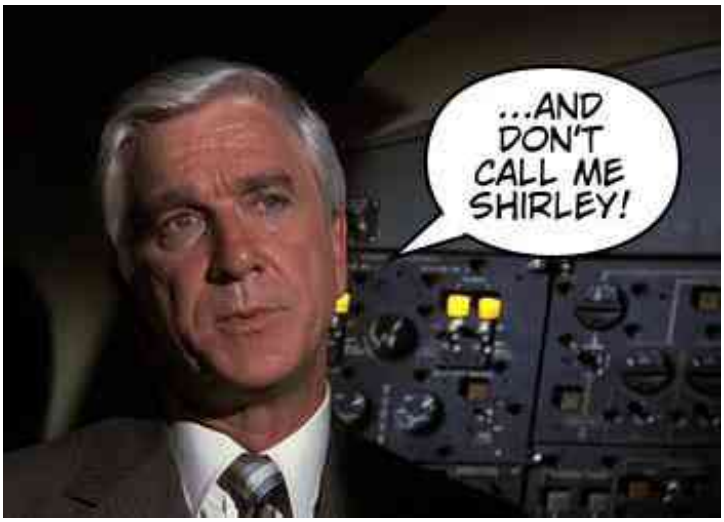
For Deliverable 4, our goal was to do a performance test on `JavaLife.java`, an intentionally inefficient implementation of Conway's Game of Life. Using Java's `visualvm`, we were to profile the application as it ran such that we could view a break-down of time spent in each method of the program. This way, we could effectively find a bottleneck in the program and see what was causing all of the trouble.

I began by compiling the program and running it with the parameters:
- `n = 100`
- `seed = 15`
- `% = 10`
- `iterations = 30000`

As the program ran, I started the `visualvm` tool and, opening the process, ran a CPU profile on it. I instantly noticed the method that was causing an obscene majority of the runtime– an obvious bottleneck.

Call Tree - Method	Total Time [%]	Total Time	Total Time (CPU)
main		273,019 ... (100%)	273,019 ms
JavaLife.main ()		273,019 ... (100%)	273,019 ms
JavaLife.<init> ()		273,019 ... (100%)	273,019 ms
World.toString ()		263,687 ...(96.6%)	263,687 ms
java.lang.String.valueOf ()		203,919 ...(74.7%)	203,919 ms
java.lang.StringBuilder.append ()		43,864 ms (16.1%)	43,864 ms
java.lang.StringBuilder.toString ()		15,498 ms (5.7%)	15,498 ms
Self time		304 ms (0.1%)	304 ms
java.lang.StringBuilder.<init> ()		101 ms (0%)	101 ms
java.io.PrintStream.println ()		4,011 ms (1.5%)	4,011 ms
Self time		3,127 ms (1.1%)	3,127 ms
World.iterate ()		2,193 ms (0.8%)	2,193 ms
Self time		0.000 ms (0%)	0.000 ms

Now I could see instantly that the problem was with the `World` class. Not just that, but that it was in its `toString()` method. It took an entire *264ms!* Even further, in Java's implementation of `valueOf()`. Surely, I could drastically reduce runtime by using a different method.



When inspecting the code, I noticed that `World.toString()` was taking the majority of the CPU time when the application was running. After exit, I got to see a Call Tree that showed me that it was in fact the `String.valueOf()` method being called within this other method that was taking by far the majority of the time. Moreover, this is taking the value of a primitive `int`, which is already printable to String in Java. I simply removed the call and replaced it with its raw equivalent. Ultimately, I was pleased to see that I was right, and that removing that method improved performance by reducing runtime by almost *2600%*.

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		11,991 ms (100%)	1
World.**toString** ()		10,430 ms (87%)	122
Self time		10,311 ms (86%)	122
Cell.**getStateRep** ()		118 ms (1%)	1220000
World.**iterate** ()		1,547 ms (12.9%)	123
Cell.**getStateRep** ()		13.8 ms (0.1%)	3326
Thread-0		3.70 ms (100%)	1
java.util.logging.LogManager$Cleaner.**run** ()		3.70 ms (100%)	1
SIGINT handler		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.**run** ()		0.000 ms (0%)	1

When testing, I wanted to be sure that the program itself was stable, and that I didn't break anything. To test that it was stable, I chose to test to see that when you enter the program with a series of seemingly valid inputs, that it would react appropriately. Moreover, I compare these results to those of the original code such that I can be sure my changes had no effect. Finally, because I'm affecting a `toString()` method, I make sure to compare an instance of the output under the same conditions before and after the refactoring.

I faced significant issues while working on this project– most of them technical, or in setting up the project. As provided, I found that the files I was given could not compile– I remedied this by removing `package` info because every file existed in the same folder anyway. This lead to a problem with unit tests, however, which I'll elaborate on later. The `visualvm` also misbehaved frequently, refusing to provide me with CPU profile information. I eventually got it to work, but I still don't understand why it didn't work in the first place. My last and biggest issue was trying to resolve how exactly to test the output of

the system without actually running the instances of Java.

Because I couldn't run my files in a package, I was completely unable to get unit tests to make use of the application or its classes. This is a complication to do with the build/test suite Gradle. However, I did manually set the output for pre- and post- re-factor code such that it can be tested directly. This is evident in each of my tests.

Viewing Test Results and other files

Because of the way that Gradle works, test results are visible as an HMTL index file located in the project. The path for this, and for other files including the outputs tested in jUnit, are available at `README.md` at the root of the project.