JavaLife

**Jordan McAleer**

CS 1699 – DELIVERABLE 4: Performance Testing Conway’s Game of Life

VisualVM was the suggested profiler for the project, therefore there was a high probability that the majority of the class would use it and my classmates could be a good resource if I did not understand using the application. The profiler’s CPU analyze option allows for a time measurement per method, which was the proposed measurement, but what was more important was the percentage of CPU time used. The percentage gives a proportional view of time consumed per method compared to the total time used.

After running different world sizes and iterations, the *World.toString* method was taking over 75% of the CPU’s time. The *toString* method is simply a String creator and not even displaying the String, therefore it should not be taking a large portion of CPU time and was likely to be an inefficient implementation of the method or at least calling an inefficient method. At first glance of the method, I thought it was unnecessary to run two for loops with the modulus operation. After examining the output of the program, I realized it was used to find print the position of the Cell, a display preference. My next instinct was the *Cell.getStateRep* method was inefficient. However, review of the CPU utilization proved that the *getStateRep* method was called almost twice per iteration and used less than 0.5% of the CPU utilization. So, it must be the for loops or the String concatenation. Evaluating the for loops, any modification would change the entire String. I then remembered that Strings are immutable and that a += operation would create three different String objects, the first String, the operand String, and the concatenation result.

Because String objects are immutable, the Java compiler creates a new instance of a *StringBuilder*, appends the operand, and returns the String conversion. This means that every += operation creates a new object and converts that object to a String. By creating a *StringBuilder* at the beginning of the method, appending any additional Strings, and converting the *StringBuilder* to a String at the end of the method after all concatenation; it avoids N object instances and N String conversions per method call, where N is the number of += operations. There were five concatenation operations, thus the method time was reduced to a 10th of its original time: the CPU time percent is less than 5%.

**JUnit Tests:**

The unit tests are grouped into five types because the tests in a group could probably be compressed into a single JUnit Test, but the tests fail for entirely different reasons which I thought could be helpful in fixing solving any errors. All the Worlds are of size 20 or less, so that the Worlds are not of abundance size.

**Absolute Minimum Size tests**: the absolute minimum size without causing exceptions is zero; this is not a very useful table size as there is no table, but the *toString* method of World still returns something when the World size is zero making it an edge case.

* *sizeZeroIsNotNull*: before the output of an empty World can be tested, it must be tested that it is not null, otherwise a *NullPointerException* will occur.
* *sizeZeroIsNotEmptyString*: prior to realizing the *toString* method return could be predicted, I tested that the String was simply not empty; this test is therefore no longer necessary with the *sizeZeroIsExpectedString* test, but maintained in case an error occurred with that test.
* *sizeZeroIsNoCallToStateRep*: the *toString* method should not be asking for the character to describe any Cell’s state representation because the tables empty and therefore no Cell’s exist
* *sizeZeroIsExpectedString*: when the table is empty, the *toString* method should return “[space][space]\n” because of the programmer’s design choice, but this needed to be maintained.
* *Other tests considered*: table sizes outside the norm (INF or negative values) seemed to be implied and unnecessary

**Actual Minimum Size tests**: the actual minimum size of a table while still expecting Cell changes is one; this is thus another edge case and the first instance where *Cell.getStateRep* should be called and tested. To keep the test from being dependent upon the Cell class working, I mocked the Cell class in each of these tests, created a two-dimensional array assigned as this Cell, and created an instance of the World class with this Cell array. Thus, I can stub the *getStateRep* method to always return X and I can predict the output.

* *sizeOneIsNotNull*: before the output of an empty World can be tested, it must be tested that it is not null, otherwise a *NullPointerException* will occur.
* *sizeOneIsNotEmptyString*: prior to realizing the *toString* method return could be predicted, I tested that the String was simply not empty; this test is therefore no longer necessary with the *sizeOneIsExpectedString* test, but maintained in case an error occurred with that test.
* *sizeOneIsExpectedString*: when the table is empty, the *toString* method should return “[space][space]0\n0[space]X\n” because of the programmer’s design choice and the *Cell.getStateRep* stubbed method.
* *Other tests considered*: getStateRep called 1 time, maximum table size (hard to find, possibly int or line size)

**Consistency tests**: the *toString* method should always return the same String when created by the same World or same arguments. I could possibly implement the actual arguments by using a spy class, but it was just easier to use the World constructor with a two dimensional Cell array filled with mock Cell classes (to avoid class dependencies), since I knew that would work. The tests use Worlds of size [1,10] which is decided by the Java Random object.

* *sameWorldHasTheSameString*: the same world should produce the same String.
* *twoWorldsWithSameArgumentsResultsInSameString*: two Worlds with the same exact Cell table (or a mock in this case) should produce the same String.

**Correct Character tests**: the *toString*method should contain spaces, newlines, ordered numbers 0-9 repeatedly and the results of the *Cell.getStateRep* method which will be tested in the **Correct Number of Characters tests**. The following tests have an instruction that replaces any characters but the one’s specified with “B” and then checks that there are no “B”s in the String.

* *hasSpacesAndNewLineChars*: the *toString* method always returns spaces and newline characters; this check that those characters are within the returned string. Random size [1,10].
* *hasNumbersInOrder*: the *toString* method always has a substring of ordered numbers from 0-9 depending on the size of the Cell table; the table is set to size 10 to be sure of the exact substring.
* *hasDoubleNumbersInOrder*: the *toString* method always has a substring of ordered numbers from 0-9 depending on the size of the Cell table; this checks that when the size is greater than 10, the numbers are repeated correctly. The table is set to size 20 to be sure of the exact substring.
* *Other tests considered:* random sizes having the exact substring (would have to do the String creating basically), vertical numbers being correct

**Correct Number of Characters tests**: the *toString* method should have a specific number of characters of the Cell state representation and a specific length. The Cell class is mocked and a World instance is created using a 10x10 Cell table of this mocked class to stub the *getStateRep* method to return ‘X’.

* *hasCorrectNumberOfCells*: the *toString* method should have 100 Xs in a 10x10 World with the stubbed method; it calculates the number of Xs by replacing everything else with the empty string and checking the length of the resulting String against size x size.
* *correctLength*: due to the [space][space] at the beginning of the first line, the proceeding numbers, and the newline character at the end of each line, each row should have size + 3 characters in it and there should be size + 1 characters per column to compensate for the first row being number indexes; thus, the total length of the String (no matter the table size) should be (size + 3) x (size + 1)
* *Other tests considered*: number of times *getStateRep* was called (*hasCorrectNumberOfCells* basically accomplishes the same idea)

**Challenges:**

The first issue was running/compiling the JavaLife program. It appears that Eclipse may have been used when creating the program and that means certain code had to be commented out to run it in the command prompt (the first package line of every class).

However, the primary challenge was just using the VisualVM profiler. The JavaSDK VisualVM showed a profiling option, but did not allow it to be clicked on and a profiling tab never appeared. I have pictures to prove the craziness of that situation. Finally, I decided to try Eclipse with the application, but Eclipse did not recognize the project. (Note: I do not use Eclipse, so that could have been a user error.) Finally, I was able to get the profiler working using NetBeans.

Finding the inefficiently written method was a lot easier than expected. It would have been even easier had I remembered the implications of the += operation. Upon recalling the disadvantages, it was easy to search for a more efficient String concatenation.

The next difficulty arose in deciding what to tests for the unit tests. How could you possibly test that the String of the reformed method equals the String of the original method without the method? That is when I asked a friend for a nudge in the right direction and he reminded me to look at edge cases for the program: in this case, strange table sizes, negative arguments. The only one that seemed to be important to the *toString* method was table size being zero. Later, I realized that the properties of the String also needed to be tested: the same if the Worlds were the same, the correct characters, the correct ordering of indexes, the correct number of representations for Cells, and the String length.

Overall, it was a very easy deliverable, and probably the most difficult part was finding a working application and writing the paper:

Code: Output:

I.hasSkills(x);

if(!I.Skills.contains(paper\_writing))

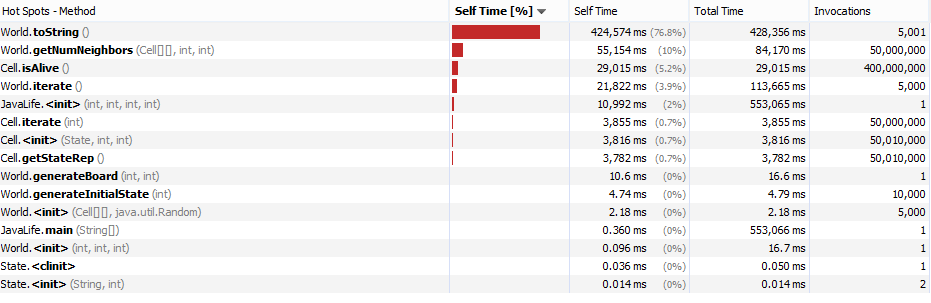
System.out.println(“I suck at writing papers”);

else

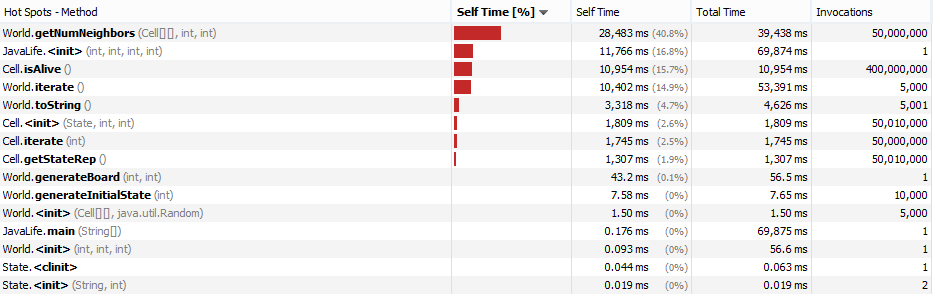
System.out.println(“I’m awesome at writing papers”);

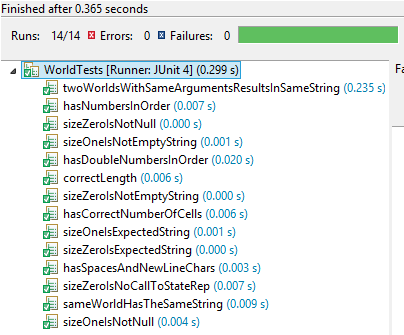
**NOTE**: I am not sure this will work if run with Eclipse or anything, but the specific way that I ran it: included the World.java, Cell.java, and State.java in the same directory as the WorldTests.java. I added import lines 7-9, and I think that should make it work in Eclipse.

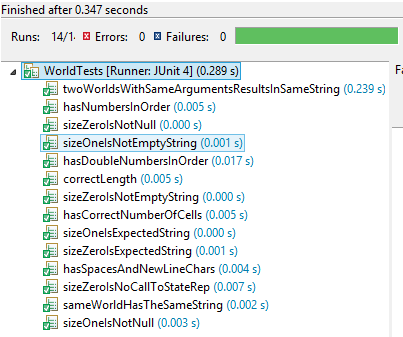
**CPU Analyzer Profiler Before Refactor:** arguments 100-12345-50-5000

****

**CPU Analyzer Profiler After Refactor:** arguments 100-12345-50-5000

****

**JUnit tests Before Refactor: JUnit tests After Refactor:**

****