I write all my test codes in CellTest.java and MainPanelTest.java, all the tests are based on the modification I made in the Cell.java and the MainPanel.java. I found methods to be enhanced and modified in those two functions. I pushed all the stuffs to the GitHub, the link is at the top of the next page.

For this deliverable, the first thing to do is detect which method to modify in this project. I profiled this application through VisualVM, which is helpful to determine which part within this code can I improve the efficiency. I started by running the program with VisualVM running, and was able to shrink a few big-problem methods as target from the start. I ran it with a continuous run for a while, then also tried the write function. In this case, Cell.toString() and MainPanel.convertToInt() are the methods that took the most CPU time among all.

For Cell.toString(), I went through the code to determine the end result of the function. I found out some methods are unnecessary and removed those, replaced those with more performable codes by myself. Besides, I wrote several pinning tests to make sure the results were the same as the original, to keep the functionality unchanged as expected. Following the requirements and my organization, then, I removed the loop from the function and simplified how the function returned the value without doing extra work. Running the pinning test code to verify the functionality.

Similarly, following the same process for MainPanel.convertToInt(). It seems that the entire function was redundant – the value that needed to be converted to an int was already an int. I also wrote pinning tests to ensure the functionality running correctly, then changed some code in the function to simply return the value that was expected. Because the convertToInt() function was private, I wrote a function that accepted an int, called convertToInt(). So as to apply the pinning tests. Making sure the functionality was not changed and the above two methods were more performable. After that, I reran the VisualVM and noticed two other functions to be modified – Cell's constructor and MainPanel.runContinuous(). For Cell's constructor, I found that MainPanel.backup() was creating entirely new objects every time it was called, and it was called every iteration. However, the only use for the backup file was to undo the last run and the undo functionality only needed to know whether the Cell was alive or not. So I refactored at this place which was discussed in the code comment*. For the runContinuous function, I decided to make manual tests for this since there is no way to run unit test as there's no input/output values in that method (void function).* I checked functionality during continuous run and made sure there's no change after I refactored by removing the unnecessary loop.

At last, the runContinous method still shows as taking the most CPU time in VisualVM, but it seems to be as a natural phenomenon due to this method is running continuously.

Manual Tests in this project:

1.

Preconditions: There is a vertical line of three alive cells on the screen.

Execution: Press the Run button.

Postconditions: The top and the bottom cells of that vertical line should die and the cells immediately to the right and the left of the middle cell will become alive.

2.

Preconditions: There is a vertical line of three alive cells on the screen.

Execution: Press the Run Continuous button.

Postconditions: The vertical line of three alive cells should alternate between a horizontal line of alive cells, centered on the middle of the original line, and turn back to the original line of cells.

3.

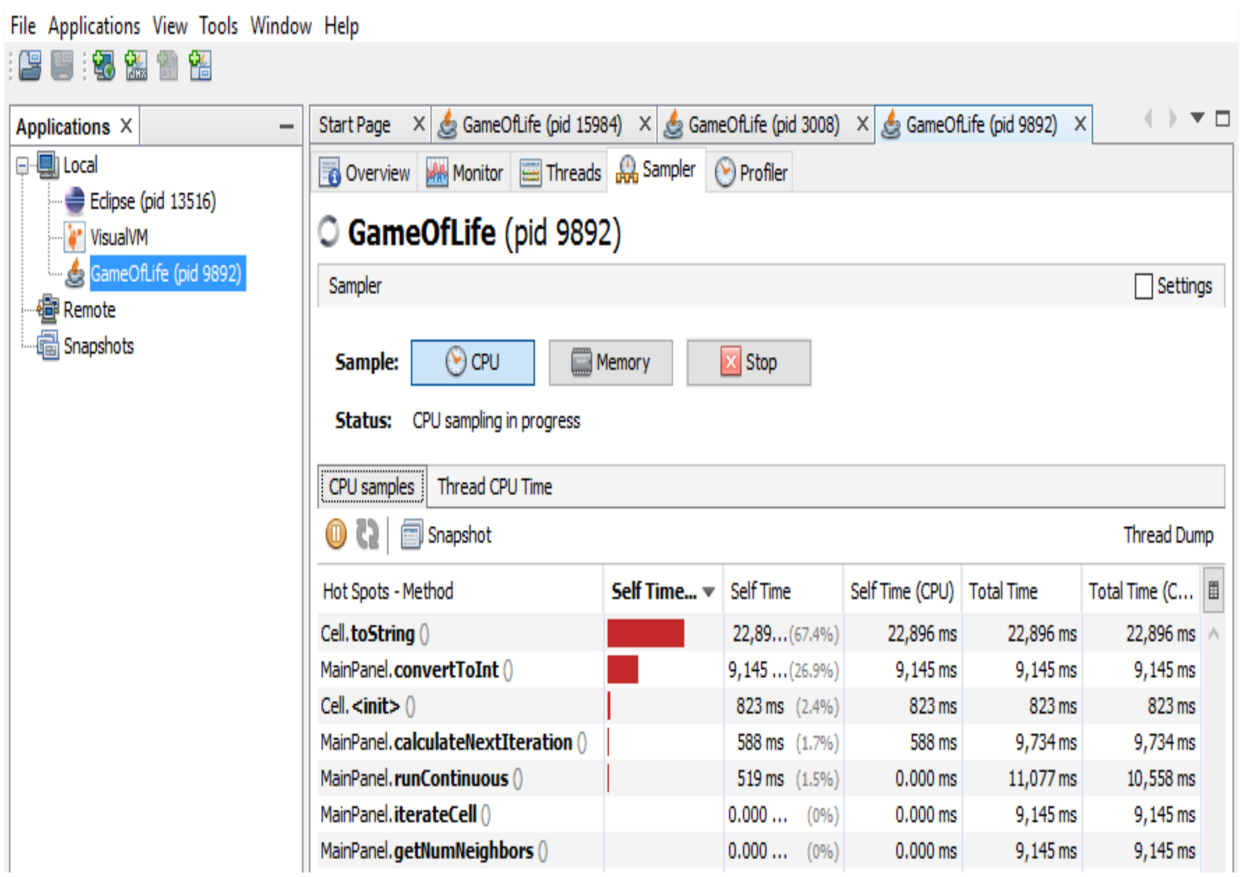
Preconditions: There are arbitrary numbers of alive cells on the screen.

Execution: Press the Run Continuous button.

Postconditions: The messages "Calculating…" and "Displaying…" are shown repeatedly in the terminal, proving that the program is running continuously as expected.

Results Screenshot:

The original program running without any refactoring (including with running continuously and then a write):



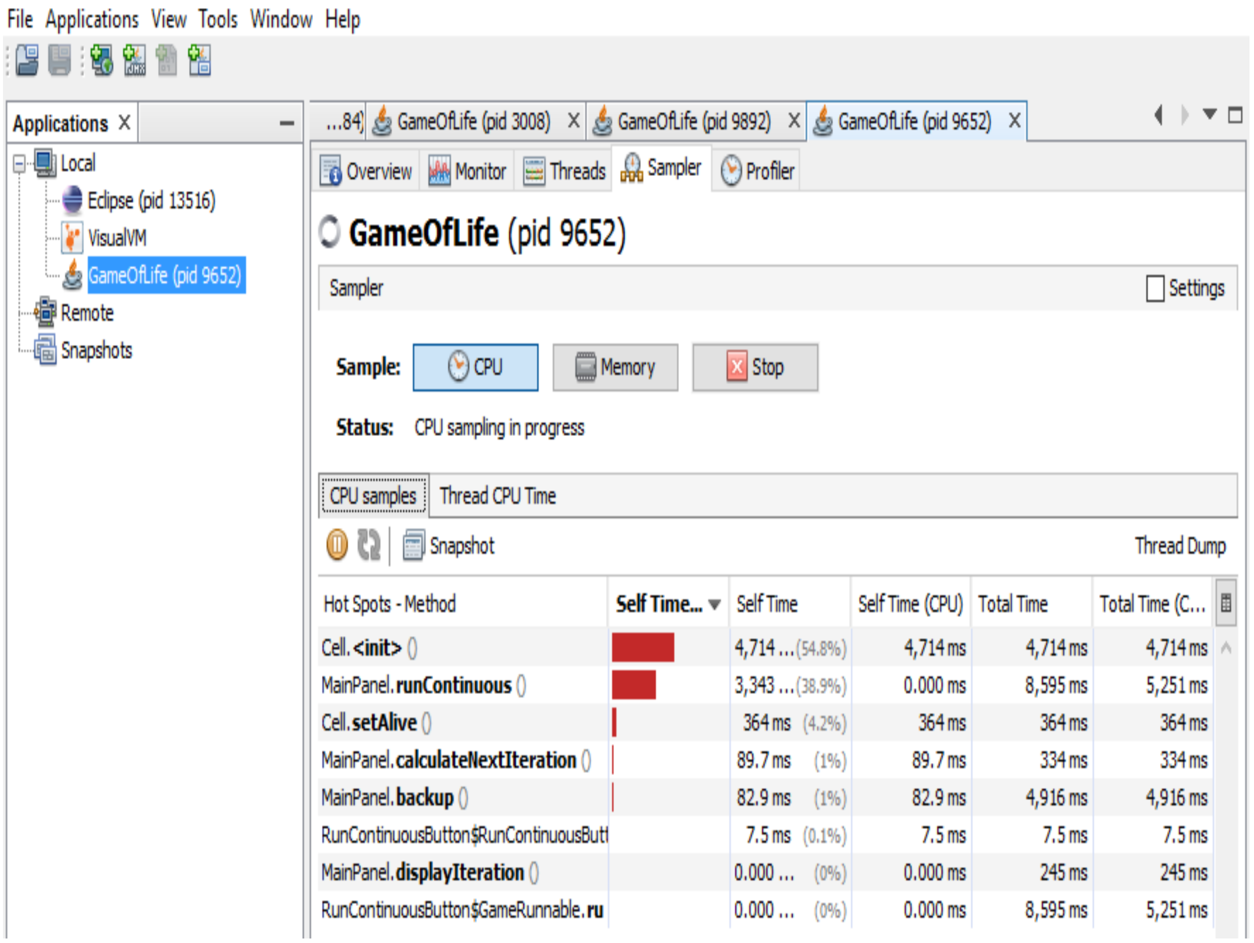
Sample: CPU

Status: CPU sampling in progress

Hot Spots - Method	Self Time...	Self Time	Self Time (CPU)	Total Time	Total Time (C...
Cell.toString ()	🟥	22,89... (67.4%)	22,896 ms	22,896 ms	22,896 ms
MainPanel.convertToInt ()	🟥	9,145 ... (26.9%)	9,145 ms	9,145 ms	9,145 ms
Cell.<init> ()	🟥	823 ms (2.4%)	823 ms	823 ms	823 ms
MainPanel.calculateNextIteration ()	🟥	588 ms (1.7%)	588 ms	9,734 ms	9,734 ms
MainPanel.runContinuous ()	🟥	519 ms (1.5%)	0.000 ms	11,077 ms	10,558 ms
MainPanel.iterateCell ()		0.000 ... (0%)	0.000 ms	9,145 ms	9,145 ms
MainPanel.getNumNeighbors ()		0.000 ... (0%)	0.000 ms	9,145 ms	9,145 ms

Figure 1

After I modified the convertToInt() method and the toString() method, running the program again(including with running continuously and then a write):


File Applications View Tools Window Help
Applications
Local
Eclipse (pid 13516)
VisualVM
GameOfLife (pid 9652)
Remote
Snapshots
...84) GameOfLife (pid 3008) GameOfLife (pid 9892) GameOfLife (pid 9652)
Overview Monitor Threads Sampler Profiler
Sampler
Settings
CPU
Memory
Stop
CPU samples
Thread CPU Time
Snapshot
Thread Dump

GameOfLife (pid 9652)

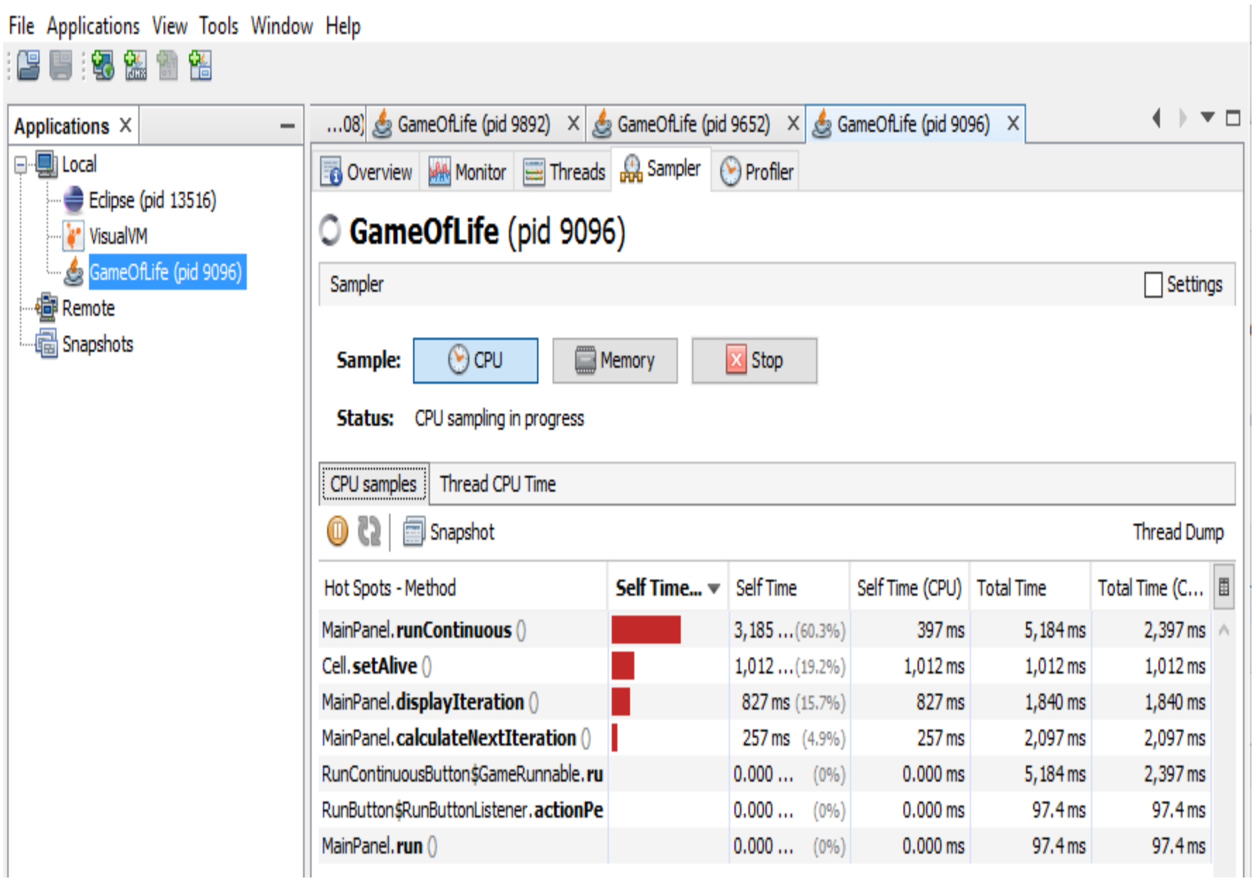
Sample: CPU

Status: CPU sampling in progress

Hot Spots - Method	Self Time...	Self Time	Self Time (CPU)	Total Time	Total Time (C...
Cell.<init> ()	🟥	4,714 ...(54.8%)	4,714 ms	4,714 ms	4,714 ms
MainPanel.runContinuous ()	🟥	3,343 ...(38.9%)	0.000 ms	8,595 ms	5,251 ms
Cell.setAlive ()	🟥	364 ms (4.2%)	364 ms	364 ms	364 ms
MainPanel.calculateNextIteration ()	🟥	89.7 ms (1%)	89.7 ms	334 ms	334 ms
MainPanel.backup ()	🟥	82.9 ms (1%)	82.9 ms	4,916 ms	4,916 ms
RunContinuousButton$RunContinuousButt		7.5 ms (0.1%)	7.5 ms	7.5 ms	7.5 ms
MainPanel.displayIteration ()		0.000 ... (0%)	0.000 ms	245 ms	245 ms
RunContinuousButton$GameRunnable.ru		0.000 ... (0%)	0.000 ms	8,595 ms	5,251 ms

Figure 2

After I refactored the method MainPanel() and the method backup():



GameOfLife (pid 9096)

Sampler

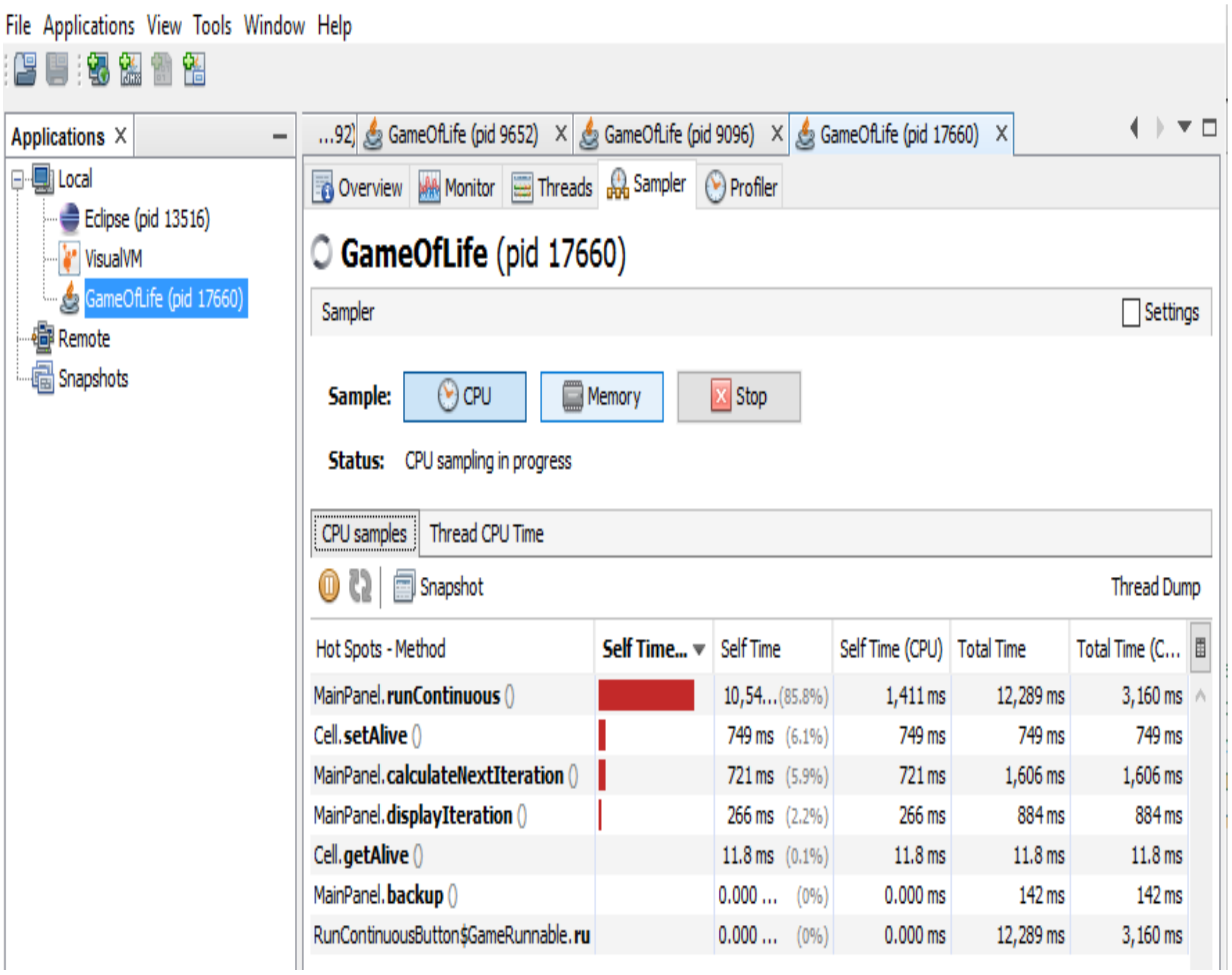
Sample: CPU Memory Stop

Status: CPU sampling in progress

Hot Spots - Method	Self Time...	Self Time	Self Time (CPU)	Total Time	Total Time (C...
MainPanel.**runContinuous** ()	🟥	3,185 ... (60.3%)	397 ms	5,184 ms	2,397 ms
Cell.**setAlive** ()	🟥	1,012 ... (19.2%)	1,012 ms	1,012 ms	1,012 ms
MainPanel.**displayIteration** ()	🟥	827 ms (15.7%)	827 ms	1,840 ms	1,840 ms
MainPanel.**calculateNextIteration** ()	🟥	257 ms (4.9%)	257 ms	2,097 ms	2,097 ms
RunContinuousButton$GameRunnable.**ru**		0.000 ... (0%)	0.000 ms	5,184 ms	2,397 ms
RunButton$RunButtonListener.**actionPe**		0.000 ... (0%)	0.000 ms	97.4 ms	97.4 ms
MainPanel.**run** ()		0.000 ... (0%)	0.000 ms	97.4 ms	97.4 ms

Figure 3

After modifying the method runContinuous()(The final change by me):



Status: CPU sampling in progress

Hot Spots - Method	Self Time...	Self Time	Self Time (CPU)	Total Time	Total Time (C...
MainPanel.**runContinuous** ()	🟥	10,54... (85.8%)	1,411 ms	12,289 ms	3,160 ms
Cell.**setAlive** ()	🟥	749 ms (6.1%)	749 ms	749 ms	749 ms
MainPanel.**calculateNextIteration** ()	🟥	721 ms (5.9%)	721 ms	1,606 ms	1,606 ms
MainPanel.**displayIteration** ()	🟥	266 ms (2.2%)	266 ms	884 ms	884 ms
Cell.**getAlive** ()		11.8 ms (0.1%)	11.8 ms	11.8 ms	11.8 ms
MainPanel.**backup** ()		0.000 ... (0%)	0.000 ms	142 ms	142 ms
RunContinuousButton$GameRunnable.**ru**		0.000 ... (0%)	0.000 ms	12,289 ms	3,160 ms

Figure 4