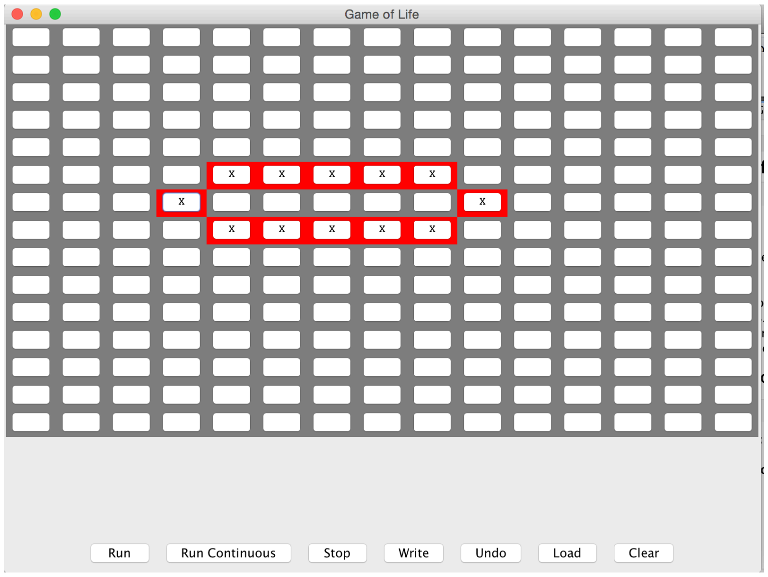
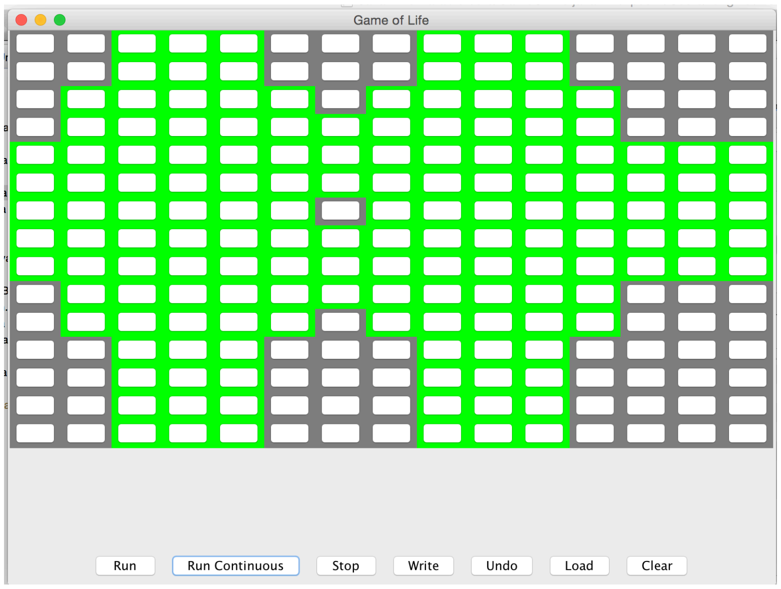
Summary

The pattern I used is shown as follows:

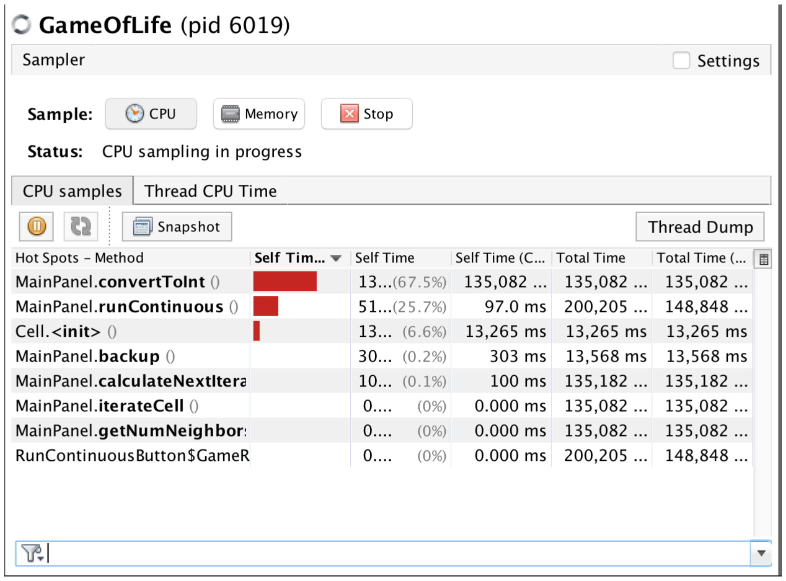


The final diagram is shown as follows when I pressed run continuous:



VisualVM is a kind of visual machine which can get the running time of each method of the game and therefore to know which method has the worst performance.

Well, if we press the running continuous button, then the result of time consuming for each method is shown as follows:


GameOfLife (pid 6019)

Sampler

Sample: CPU Memory Stop

Status: CPU sampling in progress

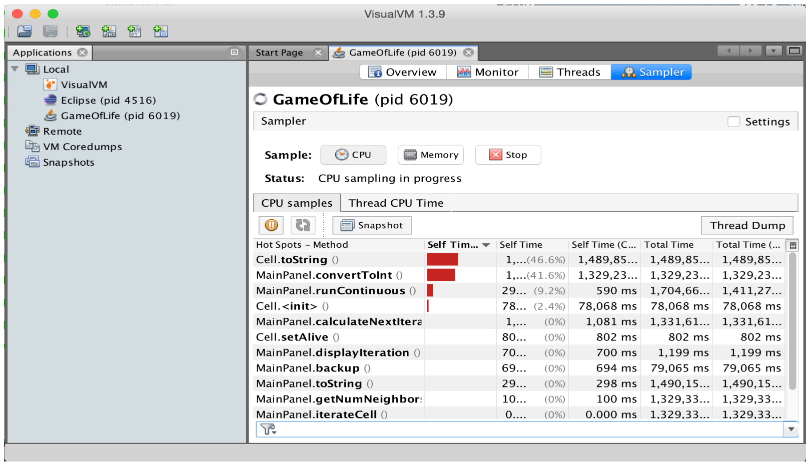
CPU samples | Thread CPU Time

Snapshot | Thread Dump

Hot Spots – Method	**Self Tim...**	Self Time	Self Time (C...	Total Time	Total Time (...
MainPanel.**convertToInt** ()	🟥	13...(67.5%)	135,082 ...	135,082 ...	135,082 ...
MainPanel.**runContinuous** ()	🟥	51...(25.7%)	97.0 ms	200,205 ...	148,848 ...
Cell.**<init>** ()	🟥	13... (6.6%)	13,265 ms	13,265 ms	13,265 ms
MainPanel.**backup** ()		30... (0.2%)	303 ms	13,568 ms	13,568 ms
MainPanel.**calculateNextItera**		10... (0.1%)	100 ms	135,182 ...	135,182 ...
MainPanel.**iterateCell** ()		0.... (0%)	0.000 ms	135,082 ...	135,082 ...
MainPanel.**getNumNeighbor:**		0.... (0%)	0.000 ms	135,082 ...	135,082 ...
RunContinuousButton$GameR		0.... (0%)	0.000 ms	200,205 ...	148,848 ...

We can see that the method convertToInt(), runContinuous() cost too much time .

After press the write button for many times, we can see that the cell.toString() method cost too much time.


VisualVM 1.3.9

Applications | Start Page | GameOfLife (pid 6019)

Local
- VisualVM
- Eclipse (pid 4516)
- GameOfLife (pid 6019)

Remote

VM Coredumps

Snapshots

Overview | Monitor | Threads | Sampler

GameOfLife (pid 6019)

Sampler

Sample: CPU Memory Stop

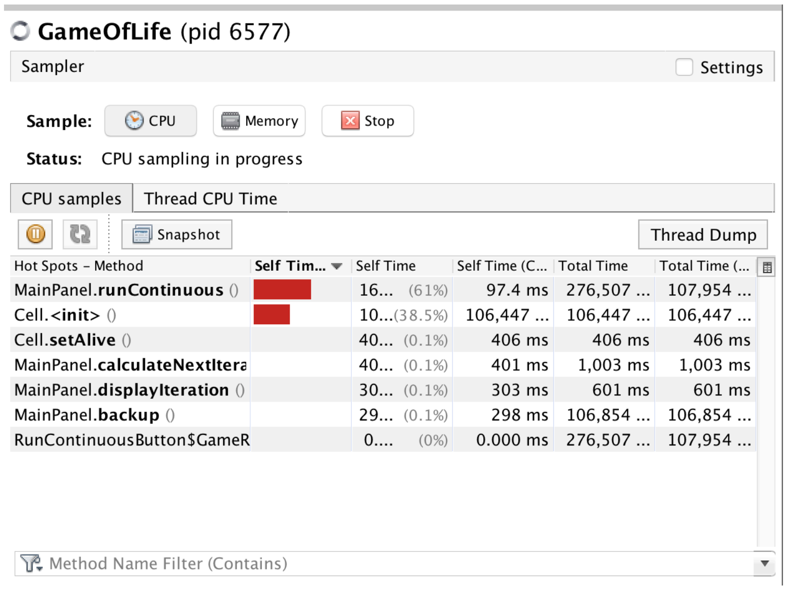
Status: CPU sampling in progress

CPU samples | Thread CPU Time

Snapshot | Thread Dump

Hot Spots – Method	**Self Tim...**	Self Time	Self Time (C...	Total Time	Total Time (...
Cell.**toString** ()	🟥	1,...(46.6%)	1,489,85...	1,489,85...	1,489,85...
MainPanel.**convertToInt** ()	🟥	1,...(41.6%)	1,329,23...	1,329,23...	1,329,23...
MainPanel.**runContinuous** ()	🟥	29... (9.2%)	590 ms	1,704,66...	1,411,27...
Cell.**<init>** ()	🟥	78... (2.4%)	78,068 ms	78,068 ms	78,068 ms
MainPanel.**calculateNextItera**		1,... (0%)	1,081 ms	1,331,61...	1,331,61...
Cell.**setAlive** ()		80... (0%)	802 ms	802 ms	802 ms
MainPanel.**displayIteration** ()		70... (0%)	700 ms	1,199 ms	1,199 ms
MainPanel.**backup** ()		69... (0%)	694 ms	79,065 ms	79,065 ms
MainPanel.**toString** ()		29... (0%)	298 ms	1,490,15...	1,490,15...
MainPanel.**getNumNeighbor:**		10... (0%)	100 ms	1,329,33...	1,329,33...
MainPanel.**iterateCell** ()		0.... (0%)	0.000 ms	1,329,33...	1,329,33...

After optimizing the 3 methods, we can see that only run Continuous cost a little bit more time than before. Maybe because if we do not press the stop button, the game is running all the way. And at the same time, even we press write button for many times, the time consuming ranking is not changed.



GameOfLife (pid 6577)

Sampler — Settings

Sample: CPU | Memory | Stop

Status: CPU sampling in progress

CPU samples | Thread CPU Time

Snapshot — Thread Dump

Hot Spots – Method	Self Tim...	Self Time		Self Time (C...	Total Time	Total Time (...
MainPanel.runContinuous ()	🟥	16...	(61%)	97.4 ms	276,507 ...	107,954 ...
Cell.<init> ()	🟥	10...	(38.5%)	106,447 ...	106,447 ...	106,447 ...
Cell.setAlive ()		40...	(0.1%)	406 ms	406 ms	406 ms
MainPanel.calculateNextItera		40...	(0.1%)	401 ms	1,003 ms	1,003 ms
MainPanel.displayIteration ()		30...	(0.1%)	303 ms	601 ms	601 ms
MainPanel.backup ()		29...	(0.1%)	298 ms	106,854 ...	106,854 ...
RunContinuousButton$GameR		0....	(0%)	0.000 ms	276,507 ...	107,954 ...

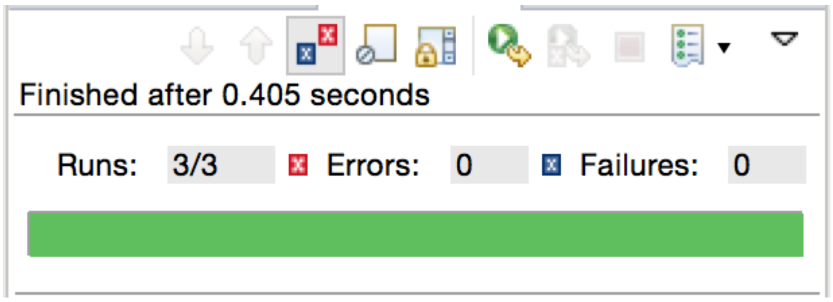
Method Name Filter (Contains)

(The modified code has already uploaded to the github, which are in the modifiedCode files.)

Junit Test

For each optimized method, we give 3 test cases respectively, and all the tests have passed successfully.

For the convertToInt() method, after modifying,
We test the maximum integer, 0 and an arbitary number, all of them are passed.



Finished after 0.405 seconds

Runs: 3/3 Errors: 0 Failures: 0

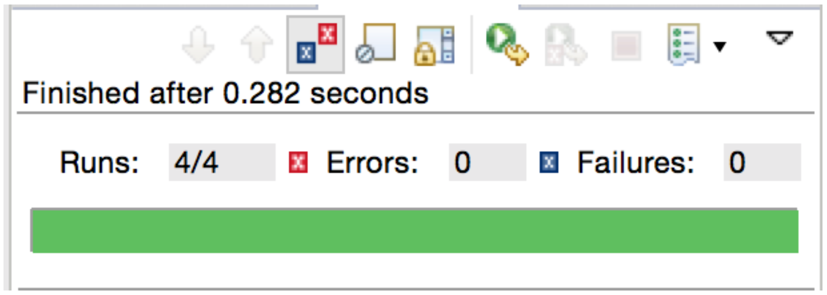
For the write execution, we can test whether the cell alive or dead executes correctly.

Each state gives 2 test cases:

1) The cell is originally dead, after setting it alive then it should return 'X'.
2) The cell is originally alive, then it should return 'X' directly.

3) The cell is originally alive, after setting it dead, then it should return '.'
4) The cell is originally dead, then it should return '.' directly.

All cases passed successfully.



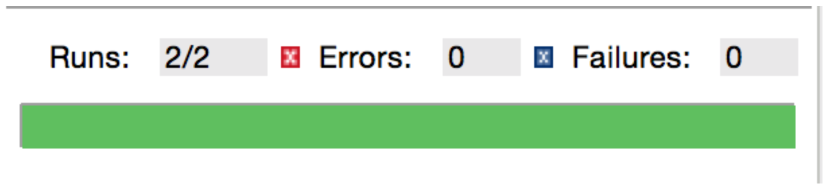
```
Finished after 0.282 seconds

Runs:  4/4    Errors:  0    Failures:  0
```

For the runContinous method, we can test whether the global variables have changed:

1)Test the initial_r
2)Test the panel size

All tests have passed successfully.



```
Runs:  2/2    Errors:  0    Failures:  0
```