IS2545 - DELIVERABLE 4: Performance Testing

Conway's Game of Life

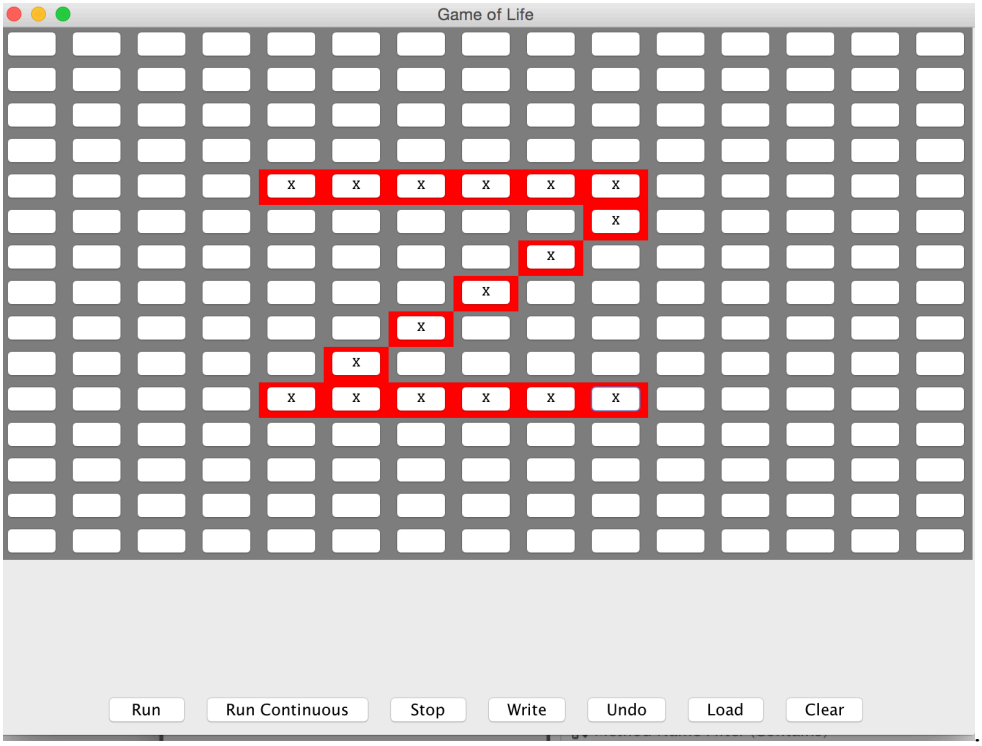
TAOLI(tal88)

Summary

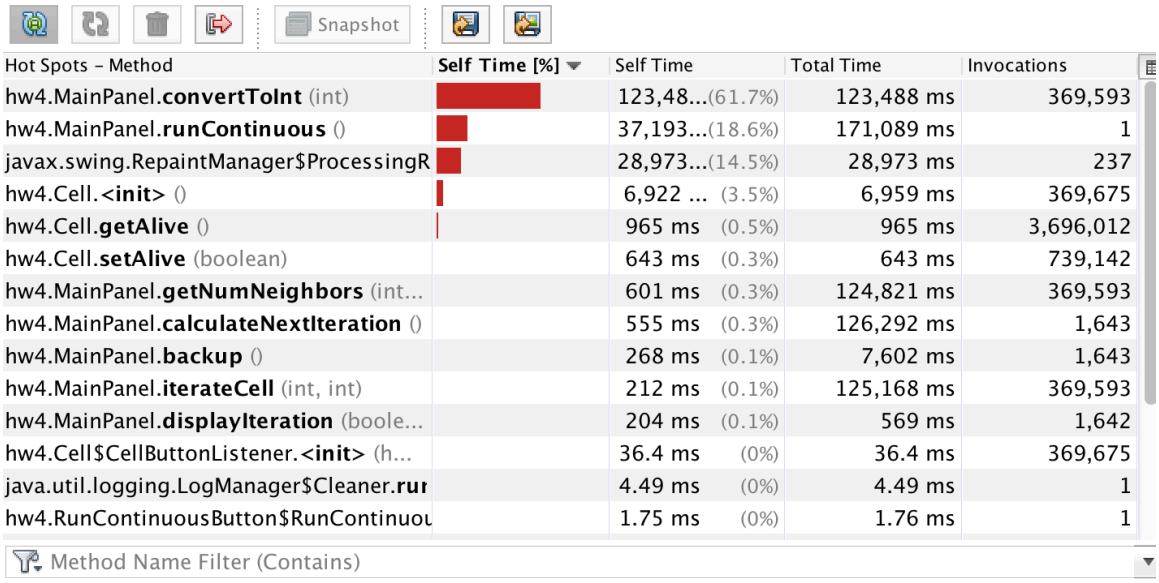
In this deliverable I use VisualVM as a profiling tool to find out which method in the application is the most CPU intensive. And then try to refactor the method, improve their performance. Finally, create some test cases to make sure that the application works as the same before.

Find methods to be refactored

First, I initialized the program by setting arguments 15. Second, I opened hw4.GameOfLife in VisualVM and clicked "Profiler" and then clicked "Run Continuous" to check different methods



From Profiler in VisualVm, I founded MainPanel.convertToInt(), MainPanel.runContinuous() and Cell.<init>() are the most CPU intensive. The picture below shows time cost of different methods.



Hot Spots – Method	Self Time [%]	Self Time	Total Time	Invocations
hw4.MainPanel.**convertToInt** (int)		123,48...(61.7%)	123,488 ms	369,593
hw4.MainPanel.**runContinuous** ()		37,193...(18.6%)	171,089 ms	1
javax.swing.RepaintManager$ProcessingR		28,973...(14.5%)	28,973 ms	237
hw4.Cell.**<init>** ()		6,922 ... (3.5%)	6,959 ms	369,675
hw4.Cell.**getAlive** ()		965 ms (0.5%)	965 ms	3,696,012
hw4.Cell.**setAlive** (boolean)		643 ms (0.3%)	643 ms	739,142
hw4.MainPanel.**getNumNeighbors** (int...		601 ms (0.3%)	124,821 ms	369,593
hw4.MainPanel.**calculateNextIteration** ()		555 ms (0.3%)	126,292 ms	1,643
hw4.MainPanel.**backup** ()		268 ms (0.1%)	7,602 ms	1,643
hw4.MainPanel.**iterateCell** (int, int)		212 ms (0.1%)	125,168 ms	369,593
hw4.MainPanel.**displayIteration** (boole...		204 ms (0.1%)	569 ms	1,642
hw4.Cell$CellButtonListener.**<init>** (h...		36.4 ms (0%)	36.4 ms	369,675
java.util.logging.LogManager$Cleaner.**run**		4.49 ms (0%)	4.49 ms	1
hw4.RunContinuousButton$RunContinuou		1.75 ms (0%)	1.76 ms	1

Refactoring

According to the result, the three methods should be modified to improve their performance

(1) MainPanel.convertToInt()

Code(before)

```
private int convertToInt(int x) {
   int c = 0;
   String padding = "0";
   while (c < _r) {
         String l = new String("0");
         padding += l;
         c++;
   }

   String n = padding + String.valueOf(x);
   int q = Integer.parseInt(n);
   return q;
    }
```

Code(after)

```
private int convertToInt(int x) {
            if (x < 0) throw new NumberFormatException();
            return x;
            }
```

Because the padding is unnecessary, and the CPU intensive will increase if we use the while loop.

(2) MainPanel.runContinuous()

Code(before)

```
public void runContinuous() {
   _running = true;
   while (_running) {
         System.out.println("Running...");
         int origR = _r;
         try {
```

```
            Thread.sleep(20);
            } catch (InterruptedException iex) { }
            for (int j=0; j < _maxCount; j++) {
                _r += (j % _size) % _maxCount;
            _r += _maxCount;
            }
            _r = origR;
            backup();
            calculateNextIteration();
        }
        }
```

Code(after)

```
public void runContinuous() {
                _running = true;
                while (_running) {
                    System.out.println("Running...");
                    backup();
                    calculateNextIteration();
                }
                }
```

The calculation here is meaningless, so wen can move the origR = _r to reduce CPU intensive.

(3) **Cell.toString()**

Code(before)

```
public String toString() {
    String toReturn = new String("");
    String currentState = getText();
    for (int j = 0; j < _maxSize; j++) {
        toReturn += currentState;
    }
    if (toReturn.substring(0,1).equals("X")) {
        return toReturn.substring(0,1);
    } else {
        return ".";
```

```
    }
```

Code(after)

```
public String toString() {
    String toReturn = getText();
    if (toReturn.substring(0,1).equals("X")) {
        return toReturn.substring(0,1);
    } else {
        return ".";
    }

    }
```

The reason why change the method is that the for loop in the is unnessary and causes CPU-intensive.

Profiling(after)

After I modified the method and run the program again, I still got the same result as the picture shows below

And then I went back to VisualVM check methods again. This time, time cost of the CPU is reduced.

Snapshot

Hot Spots – Method	Self Time [%]	Self Time	Total Time	Invocations
javax.swing.RepaintManager$ProcessingR		10,088...(14.2%)	10,088 ms	82
hw4.Cell.**setAlive** (boolean)		3,828 ... (5.4%)	3,828 ms	5,717,944
hw4.Cell.**getAlive** ()		3,432 ... (4.8%)	3,432 ms	28,588,500
hw4.MainPanel.**getNumNeighbors** (int...		3,211 ... (4.5%)	5,959 ms	2,858,850
hw4.MainPanel.**backup** ()		1,737 ... (2.4%)	48,490 ms	12,706
hw4.MainPanel.**calculateNextIteration** ()		1,287 ... (1.8%)	11,792 ms	12,706
hw4.MainPanel.**displayIteration** (boole...		1,198 ... (1.7%)	3,492 ms	12,706
hw4.MainPanel.**iterateCell** (int, int)		815 ms (1.1%)	7,012 ms	2,858,850
hw4.MainPanel.**runContinuous** ()		656 ms (0.9%)	60,939 ms	1
hw4.Cell$CellButtonListener.**<init>** (h...		219 ms (0.3%)	219 ms	2,858,850
hw4.MainPanel.**convertToInt** (int)		178 ms (0.3%)	178 ms	2,858,850
hw4.RunContinuousButton$RunContinuou		2.41 ms (0%)	2.44 ms	1
sun.java2d.opengl.OGLSurfaceData$1.**rur**		0.926 ... (0%)	0.926 ms	8
hw4.Cell$CellButtonListener.**actionPerfo**		0.528 ... (0%)	3.81 ms	19

Method Name Filter (Contains)

Pining Test

I designed three tests for MainPanel.convertToInt() to test different value when we input and three tests for Cell.toString().

Tests passed: 100.00 %

All 6 tests passed.(1.292 s)

- hw4.CellTest passed
  - hw4.CellTest.CellTest passed (0.433 s)
  - hw4.CellTest.CellTestDead passed (0.001 s)
  - hw4.CellTest.CellTestLive passed (0.001 s)
- hw4.ConvertToIntTest passed
  - TestConvertToIntZero passed (0.565 s)
  - TestConvertToIntMax passed (0.054 s)
  - testConvertToIntCorrect passed (0.067 s)

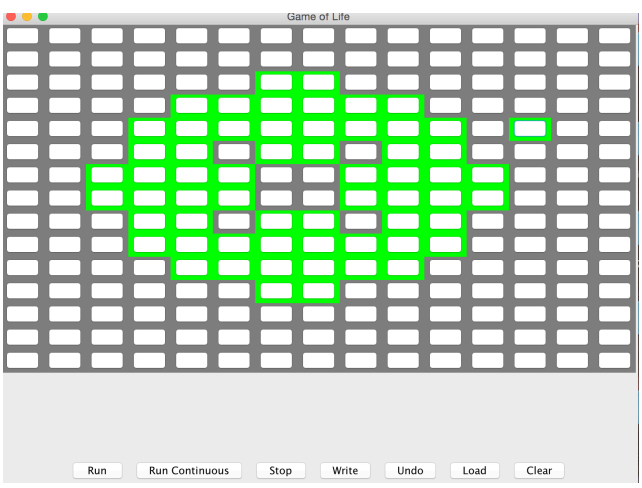
For MainPanel.runContinuous(), cause it is difficult to unit test, so I designed three manual test cases.

TEST CASE 1:
Test whether the original and modified method runCountinous() return the same result based on 15*15 world.

PRECONDITIONS:
Create a 15*15 world

EXECUTION STEPS:
1. click on the panel to choose cells

2. Click Run Continuous button
3. Wait until all the blocks don't change
4. Take a screenshot of the result



5. Exit program
6. Implement the modified runContinous() method.
7. Create a 15*15 world
8. Repeat the same process from 1-3(using same input picture)
9. Take a screenshot of the result
10. Exit program

POSTCONDITIONS:
The results of original and modified runContinous() methods are the same.

TEST CASE 2:
Test whether the original and modified method runCountinous() return the same result based on 15*15 world.
.
PRECONDITIONS:
Create a 15*15 world

EXECUTION STEPS:
1. click on the panel to choose cells

2. Click Run Continuous button
3. Wait until all the blocks don’t change
4. Take a screenshot of the result

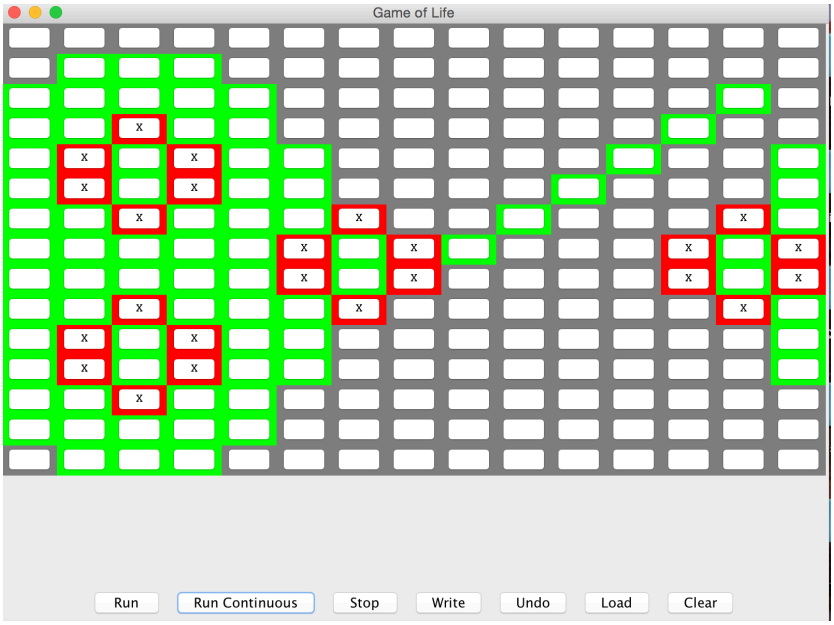
5. Exit program
6. Implement the modified runContinous() method.
7. Create a 15*15 world
8. Repeat the same process from 1-3(using same input picture)
9. Take a screenshot of the result
10. Exit program

POSTCONDITIONS:
The results of original and modified runContinous() methods are the same.

TEST CASE 3:
Test whether the original and modified method runCountinous() return the same result based on 15*15 world.
.
PRECONDITIONS:
Create a 15*15 world

EXECUTION STEPS:
1. click on the panel to choose cells

2. Click Run Continuous button
3. Wait until all the blocks don't change
4. Take a screenshot of the result



5. Exit program
6. Implement the modified runContinous() method.
7. Create a 15*15 world
8. Repeat the same process from 1-3(using same input picture)
9. Take a screenshot of the result
10. Exit program

POSTCONDITIONS:

The results of original and modified runContinous() methods are the same.