IS2545_DELIVERABLE_4

Qi Chen

1. Introduction:

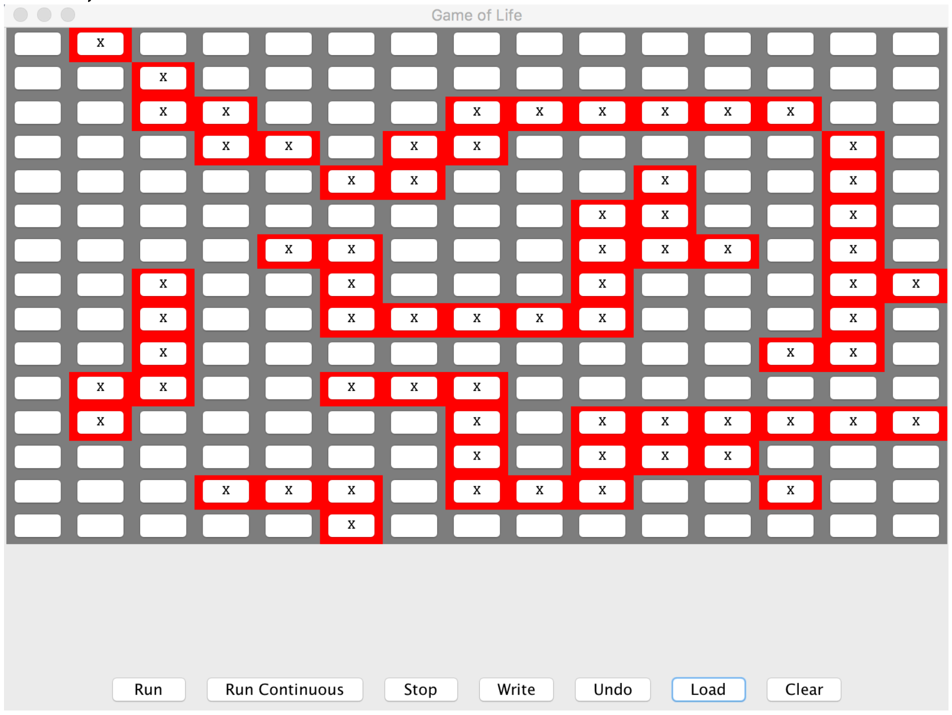
Profiling Tool: Visual VM

Basic ideas: After going through the whole java code in Eclipse, thanks to the Modular programming by the designer, I can quickly understand how each part of code work together. Before doing profiling, I have noticed there are some redundancy code in MainPanel.java. For example, in convertToInt(int x) method, the concatenation by using "+" in string is not a good way, the Thread.sleep(20) in runContinuous method seems to be able to removed, etc.

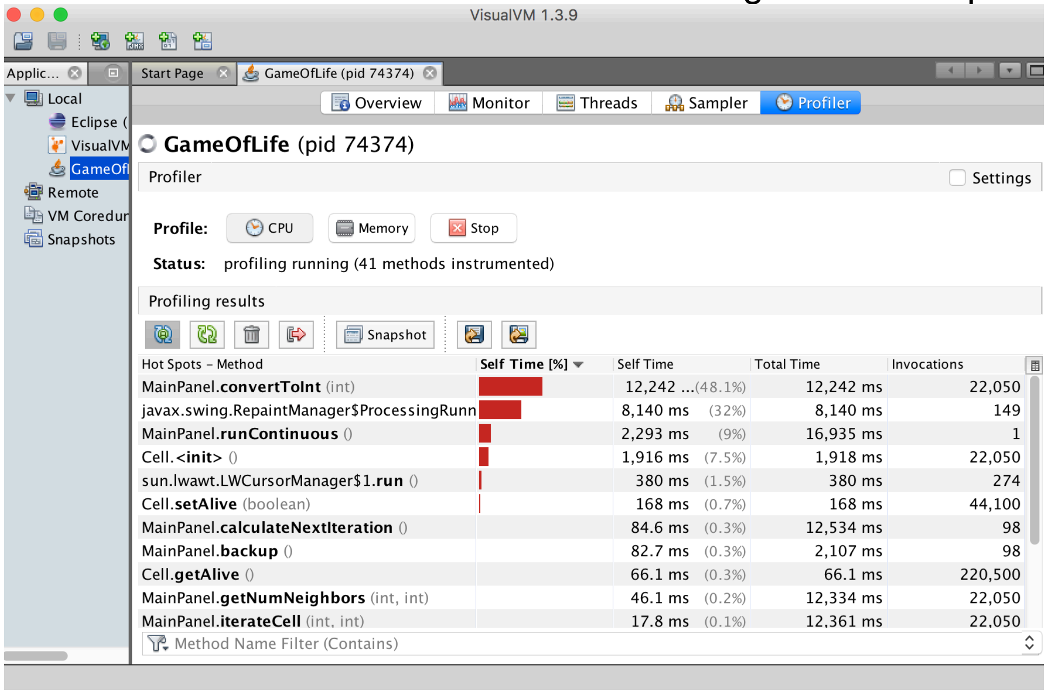
2. Profiling (before)

Visual VM for Mac is a very simple tool for users to determine which method is most CPU-intensive.

First, I initialized the project which argument is 15, and clicked some cells shown below. Also, in order to comparable the result after I revised the code, I click "Write" button to save this initialization.



Then, I went to Visual Vm, chose “GameOfLive” and click “Profiler”, and then clicked “Run Continuous” button in the “game of live” panel.



GameOfLife (pid 74374)

Profiler

Profile:

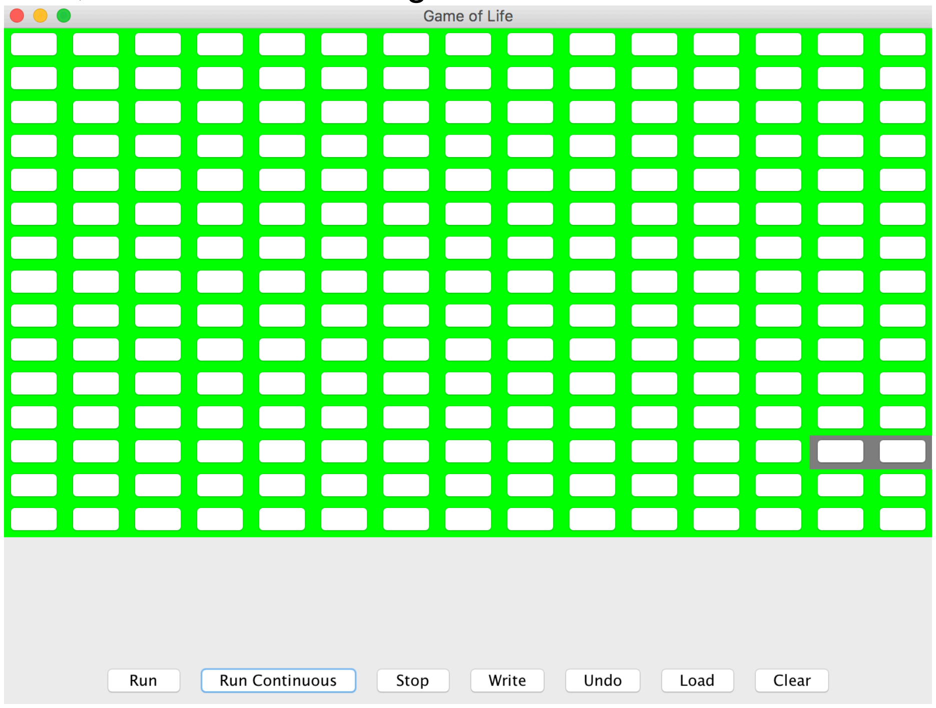
Status: profiling running (41 methods instrumented)

Profiling results

Hot Spots – Method	Self Time [%]	Self Time	Total Time	Invocations
MainPanel.**convertToInt** (int)		12,242 ...(48.1%)	12,242 ms	22,050
javax.swing.RepaintManager$ProcessingRunn		8,140 ms (32%)	8,140 ms	149
MainPanel.**runContinuous** ()		2,293 ms (9%)	16,935 ms	1
Cell.**<init>** ()		1,916 ms (7.5%)	1,918 ms	22,050
sun.lwawt.LWCursorManager$1.**run** ()		380 ms (1.5%)	380 ms	274
Cell.**setAlive** (boolean)		168 ms (0.7%)	168 ms	44,100
MainPanel.**calculateNextIteration** ()		84.6 ms (0.3%)	12,534 ms	98
MainPanel.**backup** ()		82.7 ms (0.3%)	2,107 ms	98
Cell.**getAlive** ()		66.1 ms (0.3%)	66.1 ms	220,500
MainPanel.**getNumNeighbors** (int, int)		46.1 ms (0.2%)	12,334 ms	22,050
MainPanel.**iterateCell** (int, int)		17.8 ms (0.1%)	12,361 ms	22,050

From the picture above, we can know that convertToInt(), runContinuous and Cell.<init>() cost most time.

Also, the result of running “Game of Live” is shown below.



3. Refactoring

According to the profiling results, I decided to modify the codes in convertToInt(), runContinuous and Cell.<init>.

3.1 Replacing String with StringBuilder in convertToInt() in MainPanel.java

Code before change:

```java
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;
}
```

convertToInt(), which is used in the function getNumNeighbors(int x, int y), can concatenate several 0s before a certain integer, making sure to keep the same format of number. but using "+" to do concatenation on string is much time-comsuming, the StringBuilder will be much better.

Code after change:

```java
private int convertToInt(int x) {
	int c = 0;
	StringBuilder padding = new StringBuilder();
	padding.append("0");
	while (c < _r) {
		padding.append("0");
		c++;
	}
	padding.append(String.valueOf(x));
	int q = Integer.parseInt(padding.toString());
	return q;
}
```

3.2 Removing for loop and Thread.sleep() in runContinuous() in MainPanel.java

We do a lot of calculation on the variable _r, but after that, we assign origR to _r at end, which means the all the calculation on _r is meaningless. plus, Thread.sleep(20) can be removed too.

Code before change.

```
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 change:

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

3.3 Some changes in Cell.java

Code before change:

```
public Cell() {
	super(" ");
	setFont(new Font("Courier", Font.PLAIN, 12));
	addActionListener(new CellButtonListener());
}

public Cell(boolean alive) {
	super(" ");
	setFont(new Font("Courier", Font.PLAIN, 12));
	addActionListener(new CellButtonListener());
	setAlive(alive);
}
```

super(" ") in constructor methods is aimed to create a button with an empty text " ", since this cell class extends JButton. This kind of code can be remove, for the JButton (Cell) is created by its default name is an empty text

```java
public Cell() {
	setFont(new Font("Courier", Font.PLAIN, 12));
	addActionListener(new CellButtonListener());
}

public Cell(boolean alive) {
	setFont(new Font("Courier", Font.PLAIN, 12));
	addActionListener(new CellButtonListener());
	setAlive(alive);
}
```

Plus, I found in toString(), in MainPanel.java, it uses cell's toString() several times and just like the problem mentioned before, using "+" to do concatenation on string is much time-comsuming, the StringBuilder will be much better. Of course, similar codes in toString() in MainPanel.java should be changed too.

Code before change:

```java
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 change:

```java
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 ".";
	}
}
```

4.Pining Test

4.1 For convertToInt method

I designed three test cases here. (refer to TestConvert2Int.java)

(1) testGetNumNeighbors1() is used to test what if all cells around a certain cell (x,y) are all dead, it is supposed to return 0;

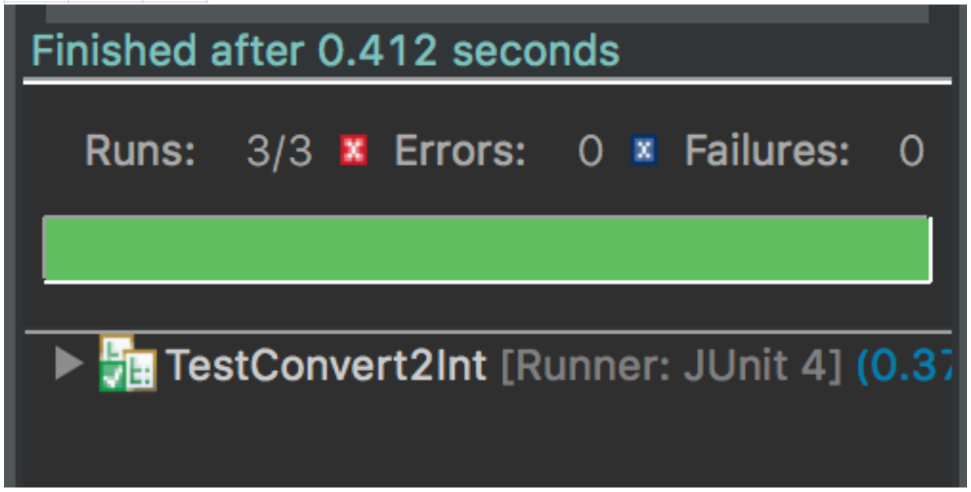
dead	dead	dead
dead	(x,y)	dead
dead	dead	dead

(2) testGetNumNeighbors2() is used to test what if all cells around a certain cell are all dead, except the left, right, top and bottom cells around this certain cell (x,y) is alive, it is supposed to return 4;

dead	live	dead
live	(x,y)	live
dead	live	dead

(3) testGetNumNeighbors3() is used to test what if all cells around a certain cell are all dead, except the cells around this certain cell (x,y) is alive, it is supposed to return 8;

live	live	live
live	(x,y)	live
live	live	live



```
Finished after 0.412 seconds

Runs:   3/3   Errors:   0   Failures:   0

▶ TestConvert2Int [Runner: JUnit 4] (0.37
```

4.2 For runContinuous() method

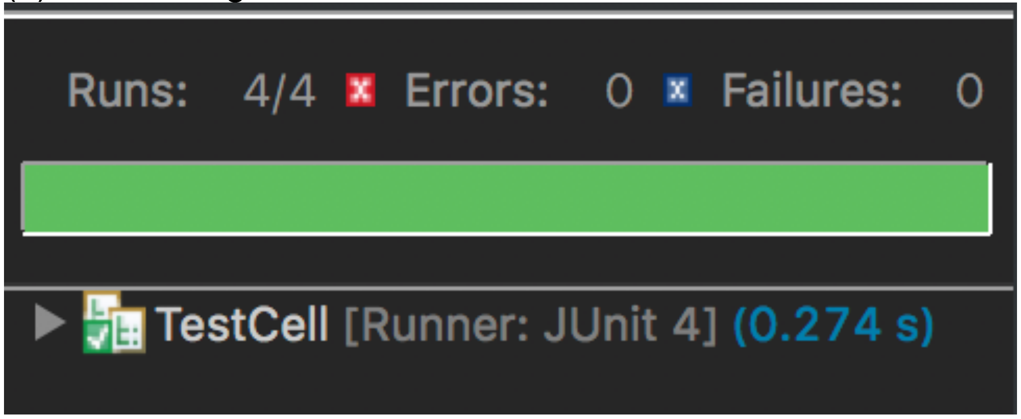
Beacuse runContinuous() method's type is void, which is hard to test using Junit, I decide to test it manually, making sure the cost of CPU in this part is less than before.

sel(ms) for **runContinuous**()	Before revising code	After revising code
1	2239	38.4
2	3186	111.3
3	2579	56
4	2653	78
mean	2664.25	70.925
median	2616	67
max value	3186	111.3
min value	2239	38.4

4.3 For Cell.java

I designed four test cases here. (refer to TestCell.java)

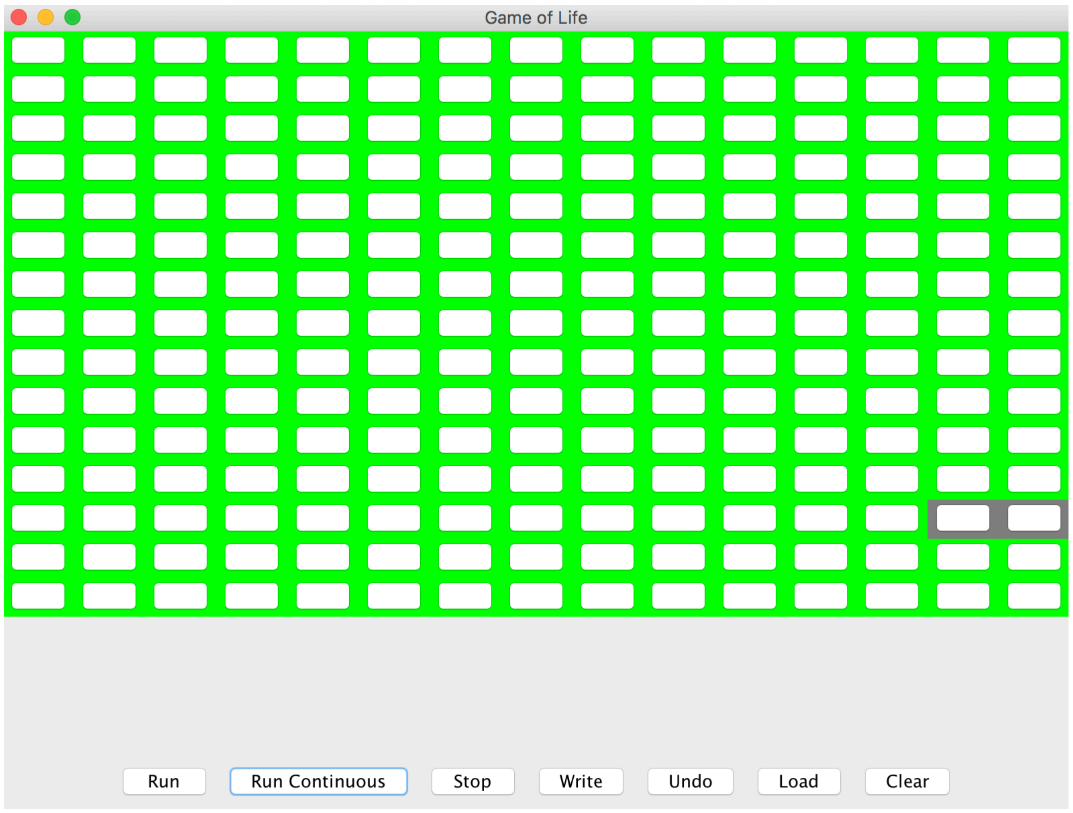
(1)testFont is used to test Font
(2)testListener is used to test listener.
(3)testToString is used to test the value the cell should return if it is dead
(4)testToString2 is used to test the value the cell should return if it is alive



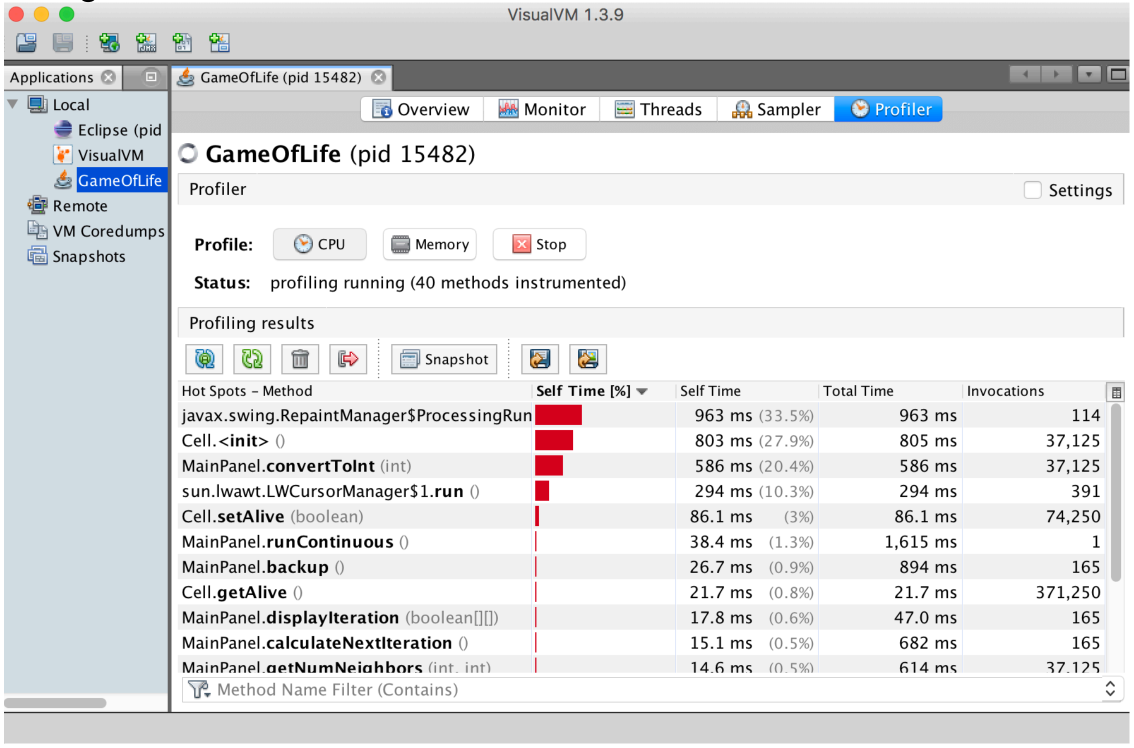
```
Runs:  4/4   Errors:  0   Failures:  0
TestCell [Runner: JUnit 4] (0.274 s)
```

5. Profiling (after)

After I revising the code and re-running the project, the result shown below is exactly same as the original project.



Using Visual VM, we can know that the cost of CPU is reduced a lot.



GameOfLife (pid 15482)

Profiler

Profile:

Status: profiling running (40 methods instrumented)

Profiling results

Hot Spots – Method	**Self Time [%]**	Self Time	Total Time	Invocations
javax.swing.RepaintManager$ProcessingRun	🟥	963 ms (33.5%)	963 ms	114
Cell.**<init>** ()	🟥	803 ms (27.9%)	805 ms	37,125
MainPanel.**convertToInt** (int)	🟥	586 ms (20.4%)	586 ms	37,125
sun.lwawt.LWCursorManager$1.**run** ()	🟥	294 ms (10.3%)	294 ms	391
Cell.**setAlive** (boolean)	🟥	86.1 ms (3%)	86.1 ms	74,250
MainPanel.**runContinuous** ()		38.4 ms (1.3%)	1,615 ms	1
MainPanel.**backup** ()		26.7 ms (0.9%)	894 ms	165
Cell.**getAlive** ()		21.7 ms (0.8%)	21.7 ms	371,250
MainPanel.**displayIteration** (boolean[][])		17.8 ms (0.6%)	47.0 ms	165
MainPanel.**calculateNextIteration** ()		15.1 ms (0.5%)	682 ms	165
MainPanel.**getNumNeighbors** (int, int)		14.6 ms (0.5%)	614 ms	37,125