SE465 – Course Project

⟨Yusu Zhao⟩ — ⟨20761282⟩ — ⟨y555zhao@uwaterloo.ca⟩

(b)

My line coverage from Jacoco is as follows.

se465

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BugDetector		100%		92%	4	31	0	89	0	6	0	1
Main		100%		84%	5	22	0	48	0	4	0	1
Total	0 of 722	100%	9 of 83	89%	9	53	0	137	0	10	0	2

Although I've achieved 100% line coverage, there is one thing that I want to address. I tested on some try-catch block to catch when there is an I/O exception when the program read or write a file. The test is done based on an actual file, and by massage the permission on it. If my program does not have the permission to modify the permission of the files, then those related tests might fail.

(c)

Generally speaking, I did not get much bugs when I use these two tools.

From pmd, I got:

- Useless paranhesis on (&&)
- Unused import – Path. Hashset

From spotbugs, I got:

- DM_DEFAULT_ENCODING
- ICAST_IDIV_CAST_TO_DOUBLE
- OS_OPEN_STREAM
- VA_FORMAT_STRING_USES_NEWLINE

And all of them are real bugs or code smells, although they are pretty tiny issues. For pmd ones, I simply removed those unused imports, and re-write my logic expression in one if statement. I would doubt the useless parentheses one, since in my own opinion, those parentheses will help code readers to understand the logic expression better and quicker. Maybe pmd should be more loose about parentheses checks.

For spotbugs ones, I added default encoding UTF-8 in each read and write stream, casted integer to floating numbers before doing division, added close clause to close file streams, and change new-line character slash n (\n) to percent n (%n). I would say these reports are pretty useful. For example,

the ICAST_IDIV_CAST_TO_DOUBLE is a but that is easy to be ignored. Our support numbers are all integers, but we are doing a division to get a percentage, which means I will get a huge difference if I directly divide these two numbers which will results in an integer division. And since it is an error that is so tiny that I might will need to spend hours to figure that out. So this tool indeed helped me a lot. However, there are some reports that I do not really understand if it will have an extra effect. The VA_FORMAT_STRING_USES_NEWLINE one, according to their official documentation, they said that "it is generally preferable better to use %n, which will produce the platform-specific line separator.". I will say it is very good to know this, but to be honest I do not know if this is generally true.

One last thing I want to mention is that spotbugs have a very good code documentation that I can quickly figure out the exact bug and relavent explanation from the error code.

(e)

Here is my report for mutation coverage:

Pit Test Coverage Report

Package Summary

se465

Number of Classes	Line Coverage		Mutation Coverage	
2	100%	137/137	97%	67/69

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
BugDetector.java	100%	89/89	96%	43/45
Main.java	100%	48/48	100%	24/24

Report generated by PIT 1.4.11

Specifically, I failed two for my BugDetector class. The first one is "deleting the code":

```
182                 callGraph.put(caller, functionCall);
183             }
184 1           br.close();
185         } catch (IOException e) {
186             throw new IllegalArgumentException("invalid value");
187         }
188     }
```

Unfortunately I cannot find one way to test this case because this is a read stream. Even if we do not close the stream, there will be no change to the input file, and there is not any way to test it (tried Google but no good result). In fact, in some cases Java garbage collector will also close the stream (not guaranteed though). That's why I believe this line is not testable.

The second mutation test is this:

```
101         ArrayList<String> funcList = new ArrayList<>(functionsCalled);
102 1       Collections.sort(funcList);
103 4       for (int i = 0; i < funcList.size() - 1; i++) {
104             String first = funcList.get(i);
105 4           for (int j = i + 1; j < funcList.size(); j++) {
106                 //Our pair will automatically sort
107  //               Pair pair = new Pair(funcList.get(i), funcList.get(j));
108                 String pair = first + Main.splitter + funcList.get(j);
109 1               if (!pairSupport.containsKey(pair)) {
```

The mutation is to make the boundary ¡=. I believe my programm will have the exact same behaviour in this case. Before the mutation, my outer loop will reach index size-2, and the last pair will be (size-2, size-1). After the mutation, my i will be size-1. However, my j will be size, which is out of index boundary, thus the inner loop is not running, so we still end up doing nothing. In other words, this is a equivalent mutant. However, I believe if the mutation changes two boundaries at the same time (i.e, both becomes ¡=), then it will be a different behaviour, and my test will kill that mutation.