

# ECE 322

## Lab Report 4

Arun Woosaree  
XXXXXXX

November 24, 2019

### 1 Introduction

The purpose of this lab was to serve as an introduction to integration testing. Integration testing is a logical extension of the previous lab, in which we did unit testing. This also allows us to build on the JUnit skills we learned in the previous lab. We were also introduced to the idea of mocking, using the Mockito library to mock out relationships between classes as we desire. Generally, there are two strategies to integration testing. The first is *Non-incremental* testing, also known as Big Bang testing. With this strategy, each module is tested individually, and then there is one final test where the entire system is tested as a whole. The second integration testing approach is *Incremental* testing, in which we combine the next module to be tested with the set of previously tested modules before running tests. This can be done in either a bottom up or top down approach. In this lab, a simple command-line database program was tested. There are 7 modules which make up the system. Module A is responsible for the GUI, B for opening, C for sorting, D for modifying, E for exiting, F for displaying, and G for updating data. Files in the database contain one entry per line, which are comma separated. Both testing strategies were used to test the application in this lab. Big bang testing was done in the first part, and we chose the Bottom-up incremental testing strategy for the second portion of the lab. Where possible, the tests were crafted to cover the full functionality of the program, including statement coverage. The exceptions are modules A and B, where it was not feasible to cover some lines. The case with Module A was that line 147 cannot be covered because when the program exits, an exception is thrown, which halts execution. With Module B, testing lines 39-42 when an IO Exception is caught does not need to be tested, because there is already code which handles the case when a file is not found. Furthermore, this situation was never encountered in our testing, due to the error handling when a file is not found. Additionally, the only code that gets run if this IO Exception somehow gets thrown are library functions (Printing to stdout, and printing a stacktrace of an error), and these library functions are likely already tested extensively, otherwise programmers would not trust them. The tests were written using

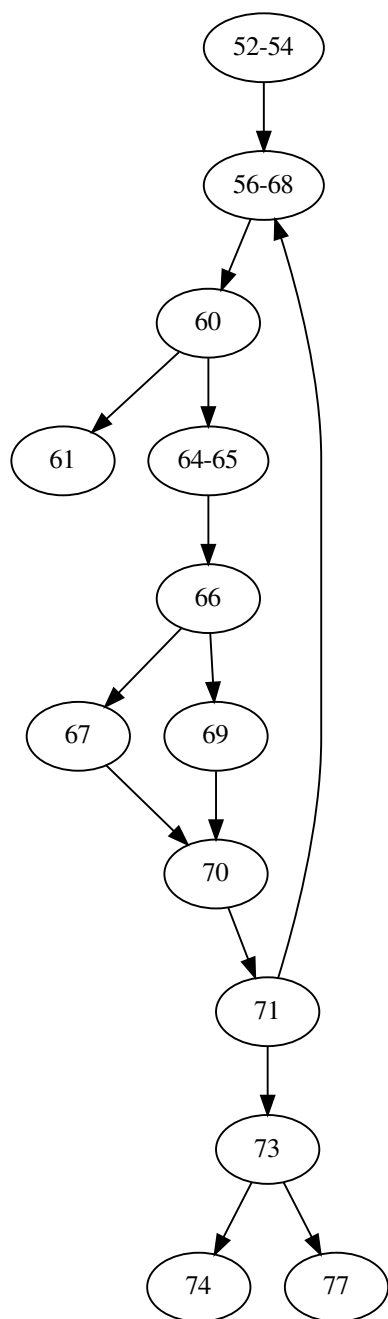
**junit-jupiter:5.5.2**, **mockito-core:3.1.0**, and were run using Java 13 with the command-line argument **-enable-preview**. A **build.gradle** is provided for ease of use, which an IDE like IntelliJ or Eclipse should be able to install dependencies from.

## 2 Task 1

For task one in this lab, the Bisect application was tested using White box testing methods. Bisect is a program written in Java, which uses the well-known bisection algorithm in mathematics to attempt to find the root of any polynomial in a given interval that crosses  $x = 0$  somewhere. That is, wherever the polynomial intersects  $x = 0$ . The algorithm which the program implements is outlined as follows:

1. Calculate  $c$ , the midpoint of the interval:  $c = \frac{a+b}{2}$
2. Calculate the polynomial's value at  $f(c)$
3. If  $|f(c)|$  is within the tolerance, stop
4. Check the sign of  $f(c)$  and replace either  $(a, f(a))$  or  $(b, f(b))$  with  $(c, f(c))$  such that the interval crosses  $x = 0$  somewhere and repeat until step 3 quits

Since this is white box testing, we have access to the source code for this application. By inspecting the source code, the following control flow graph was generated for the application. The numbers in each node represent the line numbers in the source code in **Bisect.java**.



By inspecting the source code and understanding the bisection algorithm, some JUnit test cases were generated. These test cases can be found in Appendix ???. A summary of these test cases is outlined in the table below.

Test Id	Description	tolerance	max iterations	polynomial	$x_1$	$x_2$	Expected	Actual
1	Exception is thrown when both $f(x_1)$ and $f(x_2)$ are $> 0$	0.000001	50	$x^2 - 1$	-5	5	Exception	Exception
2	Exception is thrown when maximum iterations are exceeded	0.000001	1	$x + 100$	-150	10000000000	Exception	Exception
3	Test constructor with tolerance	50.0	50	$x$	-10	10	0	0
4	Test constructor with tolerance and max iterations	0	50	$x$	-10	10	0	0
5	Max iterations getter	0.000001	500	$x$	n/a	n/a	500	500
6	Max iterations setter	0.000001	500	$x$	n/a	n/a	500	500
7	Tolerance getter	0.0	50	$x$	n/a	n/a	0.0	0.0
8	Tolerance setter	0.0	50	$x$	n/a	n/a	0.0	0.0
9	Normal test case, make sure that $f(result) \leq result$ , and it iterates more than once	0.000001	50	$x^2 - 1$	-1.5	0.5	-1	-1

While code coverage is a useful metric for determining the quality of tests, it should not be the only factor in determining the quality of the tests. After all, it is possible to have all tests cover all lines and branches, but the tests may not be checking for the correctness in execution of these statements and branching conditions. What is useful however, is knowing which lines of the code have been executed in the tests. That way, if a line is missed for example, code coverage criterion will bring to light that fact that the line of code was not covered. Furthermore, branch coverage will also ensure that all branches have been executed in the tests. In short, coverage criterion is effective for letting us know when a part of the code *for sure* is not being tested. It does not however, guarantee that the correctness of the covered statements and branches.

Based solely on the coverage criteria, it would be in my opinion foolish, or at best naive to be completely confident in the application being bug-free. Having well-thought-out tests that adequately cover the functionality of the programs *in addition to* the code coverage criterion, however, does increase my confidence in the program working. Tests that are written such that they only cover the code coverage criterion will not adequately cover the correctly. The tests have to

be made with the knowledge of how the Bisection algorithm works, and testing that the program computes the result as expected. With the test cases made in this lab, (available in Appendix ??), I am fairly confident that the portion of the program that does the Bisection algorithm works correctly, since they were crafted with the knowledge of how the Bisection algorithm works, and therefore test that the functionality of the algorithm.

In general, the number of paths available to test is a number between  $n+1 \leq x \leq 2^n$ , where  $n$  is the number of branches. This depends on when the branches merge. Additionally, for this application, there is a branch decision on every loop iteration. So, the number of test cases depends on the number of maximum iterations. Even with the default value of 50 iterations, we can see how quickly the number of tests needed for path coverage can blow up exponentially to ridiculous proportions, which is why path coverage is not realistic.

### 3 Task 2

For the second part of the lab, we are to assume that we have a system with three independent variables: A, B, C. Each variable has three possible values: 0, 1, 2. There is no actual testing for this portion, it is a conceptual exercise.

In total, there are  $3 \times 3 \times 3 = 27$  test cases if we were to do combinatorial testing. We are looking for a standard orthogonal array that can fit  $3^3$ . From <http://neilsloane.com/oadir/>, we find that the  $L_9(3^4)$  standard orthogonal array works. Below is the set of test cases using the orthogonal array mentioned above:

0	0	0	0
0	1	1	2
0	2	2	1
1	0	1	1
1	1	2	0
1	2	0	2
2	0	2	2
2	1	0	1
2	2	1	0

The PICT program (<https://github.com/microsoft/pict>) was used to generate test cases with the following input:

```
A: 0, 1, 2
B: 0, 1, 2
C: 0, 1, 2
```

It should be noted that strangely, there seems to be different outputs depending on the operating system that pict is run on.

On GNU/Linux systems, the output is as follows:

A	B	C
0	0	0
0	2	1
2	0	2
1	0	1
2	1	0
1	1	2
0	1	1
1	2	0
0	2	2
2	2	1

On Windows, the output is as follows:

A	B	C
1	2	0
2	0	2
0	1	0
2	2	1
0	0	1
1	1	2
2	0	0
0	2	2
1	0	1
2	1	1

On macOS, the output is as follows:

A	B	C
0	2	2
2	2	1
0	0	0
2	1	0
2	0	2
1	2	0
0	1	1
1	1	2
1	0	1

What is strange is that on the Windows and macOS systems, 10 test cases are generated, while on GNU/Linux, 11 test cases are generated.

Given the inconsistent outputs of the pict tool depending on the operating system that the program is run on, I am not sure about the effectiveness of the tool for test case generation. Ideally, the test cases generated should be consistent, since this they are not supposed to be chosen randomly. If the program worked consistently, however, it would be quite effective since not much effort would be required to come up with the test cases, and we have the guarantee that every pair of inputs is tested. Compared to the orthogonal array, the

pict tool generated more test cases, which is another consideration to take into account.

Pairwise testing is fairly useful, especially when you consider it versus combinatorial testing. In this toy example, the number of test cases was not reduced that much, (27 for combinatorial testing, versus 9 with an orthogonal table), but one can imagine the reduction in test cases when there are more inputs, and more possible values that these inputs can take on. Compared with random combinations, pairwise testing gives us the guarantee that we are testing every single pair of input factors, whereas random combinations does not have this guarantee, and there are far more combinations to choose from. However, random combinations can by chance reveal errors that pairwise testing would not catch.

Pairwise testing catches errors where two inputs interact with each other. It does not take into account the effect that multiple inputs might have together. For example, imagine an application that requires three inputs to be in a certain state before something gets activated. Using pairwise testing there is a possibility that we would miss testing the part of the app, where three inputs are in the state for something to happen, since the pairwise tests would only focus on the combination of pairs of the inputs. In this way, core functionality of the application can be missed in testing. Furthermore, with pairwise testing it is possible to miss the more probable combinations when selecting the test data. However, given the reduction in the number of test cases versus exhaustive testing, not to mention the cost of running so many tests, these weaknesses may be justifiable to some testers, given that pairwise testing usually results yields higher defect detection rates, increases the test coverage, all while using significantly less test cases. As mentioned above, to cover all possible inputs for the application in this example, a total of 27 test cases are required.

## 4 Conclusion

In this lab, we were introduced to the white-box testing strategy. There was a focus on control flow testing, and coverage criterion: Statement coverage, branch coverage, condition coverage, and path coverage. A control flow graph was generated, and test cases were made knowing the implementation of the program, and by inspecting to source code. Knowing the implementation details of the program allowed for the creation of test cases that not only check for correctness of the implementation of the algorithm, but also that all lines and branches in the code were covered by the tests. Statement and branch coverage on their own do not indicate good tests, but they can at a glance tell us which parts of an application for sure have not been tested yet. In this part of the lab, we also discovered the infeasability of path coverage, due to a while loop causing the number of potential paths taken to increase exponentially. In the second part of the lab, the focus was on pairwise testing, where we analyzed a hypothetical testing scenario and used a standard orthogonal array to generate test cases, and contrasted that with using a tool made by Microsoft named PICT which

automatically generates test cases, and compared and contrasted these methods versus exhaustive testing. In general, it seems like pairwise testing is great for reducing the number of test cases, while still keeping detection rate of failures high, but it also has some weaknesses, like when more than two inputs have some important relation in the application, when pairwise testing by definition only looks at the relationships between each pair of inputs.



## A Big Bang Testing Strategy

### A.1 Module A

```
1 package bigbang;
2
3 import data.Entry;
4 import modules.*;
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.Test;
7 import org.mockito.Mockito;
8
9 import java.io.ByteArrayOutputStream;
10 import java.io.PrintStream;
11 import java.util.ArrayList;
12
13 import static org.junit.jupiter.api.Assertions.assertEquals;
14 import static org.junit.jupiter.api.Assertions.assertThrows;
15 import static org.mockito.ArgumentMatchers.*;
16 import static org.mockito.Mockito.*;
17
18
19 public class TestA {
20
21     ModuleA ma;
22     ModuleB mb;
23     ModuleC mc;
24     ModuleD md;
25     ModuleE me;
26     final static String TEST_FILENAME = "testFileName";
27
28     ByteArrayOutputStream stdout;
29
30     @BeforeEach
31     public void setUp(){
32         mb = Mockito.mock(ModuleB.class);
33         mc = Mockito.mock(ModuleC.class);
34         md = Mockito.mock(ModuleD.class);
35         me = Mockito.mock(ModuleE.class);
36
37         ma = new ModuleA(mb, mc, md, me);
38
39         newStdout();
40     }
41
42     public void newStdout(){
```

```

43         stdout = new ByteArrayOutputStream();
44         ma.setOutputStream(new PrintStream(stdout));
45     }
46
47     @Test
48     public void testHelp() throws ModuleE.DataBaseExitException {
49         ma.run(new String[]{"help"});
50
51         final String help = "Available Commands: \n" +
52             "load <filepath>\n" +
53             "add <name> <number>\n" +
54             "update <index> <name> <number>\n" +
55             "delete <index>\n" +
56             "sort\n" +
57             "exit";
58
59         assertEquals(help + "\n", stdout.toString());
60     }
61
62     public void load() throws ModuleE.DataBaseExitException {
63         Mockito.when(md.insertData(any(), anyString()),
64             ↪ anyString(), anyString()).thenReturn(new
65             ↪ ArrayList<Entry>());
66         ma.run(new String[]{"load", TEST_FILENAME});
67     }
68
69     @Test
70     public void testLoad() throws ModuleE.DataBaseExitException {
71         load();
72         verify(mb, times(1)).loadFile(anyString());
73     }
74
75     @Test
76     public void testLoadNoArgument() throws
77         ↪ ModuleE.DataBaseExitException {
78         ma.run(new String[]{"load"});
79         assertEquals("Malformed command!\n",
80             ↪ stdout.toString());
81         verify(mb, never()).loadFile(anyString());
82     }
83
84     @Test
85     public void testLoadBReturnsNull() throws
86         ↪ ModuleE.DataBaseExitException {
87         Mockito.when(mb.loadFile(anyString())).thenReturn(null);

```

```

84         ma.run(new String[]{"load", TEST_FILENAME});
85         verify(mb, times(1)).loadFile(anyString());
86     }
87
88     @Test
89     public void testAddNoData() throws
90     ↪ ModuleE.DataBaseExitException {
91         ma.run(new String[]{"add", "name", "number"});
92         assertEquals("No file loaded!\n", stdout.toString());
93         verify(md, never()).insertData(any(), anyString(),
94         ↪ anyString(), anyString());
95     }
96
97     @Test
98     public void testAdd() throws ModuleE.DataBaseExitException {
99         Mockito.when(md.insertData(any(), anyString(),
100         ↪ anyString(), anyString())).thenReturn(new
101         ↪ ArrayList<Entry>());
102         ma.run(new String[]{"load", TEST_FILENAME});
103         ma.run(new String[]{"add", "name", "number"});
104         verify(md, times(1)).insertData(any(), anyString(),
105         ↪ anyString(), anyString());
106     }
107
108     @Test
109     public void testAddDreturnsNull() throws
110     ↪ ModuleE.DataBaseExitException {
111         Mockito.when(md.insertData(any(), anyString(),
112         ↪ anyString(), anyString())).thenReturn(null);
113         ma.run(new String[]{"load", TEST_FILENAME});
114         ma.run(new String[]{"add", "name", "number"});
115         verify(md, times(1)).insertData(any(), anyString(),
116         ↪ anyString(), anyString());
117     }
118
119     @Test
120     public void testAddNoArgument() throws
121     ↪ ModuleE.DataBaseExitException {
122         ma.run(new String[]{"load", TEST_FILENAME});
123         ma.run(new String[]{"add"});
124         assertEquals("Malformed command!\n", stdout.toString());
125         verify(md, never()).insertData(any(), anyString(),
126         ↪ anyString(), anyString());
127     }
128
129     @Test

```

```

120     public void testSort() throws ModuleE.DataBaseExitException {
121         ma.run(new String[]{"load", TEST_FILENAME});
122         ma.run(new String[]{"sort"});
123         verify(mc, times(1)).sortData(any());
124     }
125
126     @Test
127     public void testSortNoData() throws
128         ↪ ModuleE.DataBaseExitException {
129         ma.run(new String[]{"sort"});
130         assertEquals("No file loaded!\n", stdout.toString());
131         verify(mc, never()).sortData(any());
132     }
133
134     @Test
135     public void testSortCReturnsNull() throws
136         ↪ ModuleE.DataBaseExitException {
137         Mockito.when(mc.sortData(any())).thenReturn(null);
138         ma.run(new String[]{"load", TEST_FILENAME});
139         ma.run(new String[]{"sort"});
140         verify(mc, times(1)).sortData(any());
141     }
142
143     @Test
144     public void testUpdate() throws ModuleE.DataBaseExitException
145         ↪ {
146         ma.run(new String[]{"load", TEST_FILENAME});
147         ma.run(new String[]{"update", "1", "2", "3"});
148         verify(md, times(1)).updateData(any(), anyInt(),
149             ↪ anyString(), anyString(), anyString());
150     }
151
152     @Test
153     public void testUpdateInvalidArguments() throws
154         ↪ ModuleE.DataBaseExitException {
155         ma.run(new String[]{"load", TEST_FILENAME});
156         ma.run(new String[]{"update", "arg1", "arg2", "arg3"});
157         verify(md, never()).updateData(any(), anyInt(),
158             ↪ anyString(), anyString(), anyString());
159     }
160
161     @Test
162     public void testUpdateNoData() throws
163         ↪ ModuleE.DataBaseExitException {
164         ma.run(new String[]{"update", "1", "2", "3"});

```

```

159         assertEquals("No file loaded!\n", stdout.toString());
160         verify(md, never()).updateData(any(), anyInt(),
            ↪ anyString(), anyString(), anyString());
161     }
162
163     @Test
164     public void testUpdateCReturnsNull() throws
        ↪ ModuleE.DataBaseExitException {
165         Mockito.when(md.updateData(any(), anyInt(), anyString(),
            ↪ anyString(), anyString())).thenReturn(null);
166         ma.run(new String[]{"load", TEST_FILENAME});
167         ma.run(new String[]{"update", "1", "2", "3"});
168         verify(md, times(1)).updateData(any(), anyInt(),
            ↪ anyString(), anyString(), anyString());
169     }
170
171     @Test
172     public void testUpdateNoArgument() throws
        ↪ ModuleE.DataBaseExitException {
173         ma.run(new String[]{"load", TEST_FILENAME});
174         ma.run(new String[]{"update"});
175         assertEquals("Malformed command!\n", stdout.toString());
176         verify(md, never()).updateData(any(), anyInt(),
            ↪ anyString(), anyString(), anyString());
177     }
178
179     @Test
180     public void testDelete() throws ModuleE.DataBaseExitException
        ↪ {
181         ma.run(new String[]{"load", TEST_FILENAME});
182         ma.run(new String[]{"delete", "1"});
183         verify(md, times(1)).deleteData(any(), anyInt(),
            ↪ anyString());
184     }
185
186     @Test
187     public void testDeleteInvalidArguments() throws
        ↪ ModuleE.DataBaseExitException {
188         ma.run(new String[]{"load", TEST_FILENAME});
189         ma.run(new String[]{"delete", "arg1"});
190         verify(md, never()).deleteData(any(), anyInt(),
            ↪ anyString());
191     }
192
193     @Test

```

```

194     public void testDeleteNoData() throws
        ↳ ModuleE.DataBaseExitException {
195         ma.run(new String[]{"delete"});
196         assertEquals("No file loaded!\n", stdout.toString());
197         verify(md, never()).deleteData(any(), anyInt(),
            ↳ anyString());
198     }
199
200     @Test
201     public void testDeleteDReturnsNull() throws
        ↳ ModuleE.DataBaseExitException {
202         Mockito.when(md.deleteData(any(), anyInt(),
            ↳ anyString())).thenReturn(null);
203         ma.run(new String[]{"load", TEST_FILENAME});
204         ma.run(new String[]{"delete", "1"});
205         verify(md, times(1)).deleteData(any(), anyInt(),
            ↳ anyString());
206     }
207
208     @Test
209     public void testDeleteNoArgument() throws
        ↳ ModuleE.DataBaseExitException {
210         ma.run(new String[]{"load", TEST_FILENAME});
211         ma.run(new String[]{"delete"});
212         assertEquals("Malformed command!\n", stdout.toString());
213         verify(md, never()).deleteData(any(), anyInt(),
            ↳ anyString());
214     }
215
216     @Test
217     public void testExit() throws ModuleE.DataBaseExitException {
218         ↳ Mockito.doThrow(ModuleE.DataBaseExitException.class).when(me).exitProgram();
219         assertThrows(ModuleE.DataBaseExitException.class, ()
            ↳ -> ma.run(new String[]{"exit"}));
220         // line 147 does not get covered, because the program
            ↳ exits
221     }
222
223     @Test
224     public void unknownCommand() throws
        ↳ ModuleE.DataBaseExitException {
225         ma.run(new String[]{"thiscommanddoesntexist"});
226     }
227
228

```

229 }

## A.2 Module B

```
1 package bigbang;
2
3 import TestUtil.TestUtil;
4 import modules.ModuleB;
5 import modules.ModuleF;
6 import org.junit.jupiter.api.AfterEach;
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9
10 import java.io.File;
11 import java.io.IOException;
12 import java.nio.file.Files;
13 import data.Entry;
14 import java.nio.file.Paths;
15 import java.util.ArrayList;
16
17 import static org.junit.jupiter.api.Assertions.assertArrayEquals;
18 import static org.junit.jupiter.api.Assertions.assertEquals;
19 import static org.mockito.Mockito.mock;
20
21 // IO Exception catching is not tested
22 // Explain in the report that it's just calling a library
23 → function
24 // Also, the scenario was never encountered e.g. reading
25 → /etc/shadow
26
27 public class TestB {
28
29     ModuleB mb;
30     ModuleF mf;
31     final static String TEST_FILENAME = "BTEST_FILE";
32     static File f;
33
34     @BeforeEach
35     public void setUp() throws IOException {
36         mf = mock(ModuleF.class);
37         mb = new ModuleB(mf);
38
39         f = new File(TEST_FILENAME);
40         f.createNewFile();
41         Files.writeString(Paths.get(TEST_FILENAME), "")
42     }
43 }
```

This

```

41     is,some
42     test
43     data""");
44     }
45
46     @AfterEach
47     public void tearDown(){
48         f.delete();
49     }
50
51     @Test
52     public void loadFileTestValidFile() {
53         ArrayList<Entry> ret = mb.loadFile(TEST_FILENAME);
54
55         ArrayList<Entry> expected = new ArrayList<>() {{
56             add(new Entry("is", "some"));
57         }};
58
59         TestUtil.compareArrayOfEntries(expected, ret);
60     }
61
62     @Test
63     public void loadFileTestInvalidFile() {
64         mb.loadFile("/");
65     }
66
67     @Test
68     public void loadFileTestFileNotFound(){
69         mb.loadFile("");
70     }
71
72     @Test
73     public void setFTest(){
74         ModuleF newF = mock(ModuleF.class);
75         mb.setF(newF);
76     }
77 }

```

### A.3 Module C

```

1 package bigbang;
2
3 import TestUtil.TestUtil;
4 import data.Entry;
5 import modules.ModuleC;
6 import modules.ModuleF;

```



```

7  import org.junit.jupiter.api.BeforeEach;
8  import org.junit.jupiter.api.Test;
9
10 import java.util.ArrayList;
11
12 import static org.mockito.Mockito.mock;
13
14 public class TestC {
15
16     ModuleF mf;
17     ModuleC mc;
18
19     @BeforeEach
20     public void setUp(){
21         mf = mock(ModuleF.class);
22         mc = new ModuleC(mf);
23     }
24
25     @Test
26     public void sortDataTest(){
27
28         final String TEST_NAME = "testName";
29         final String TEST_NUMBER = "testNumber";
30
31         ArrayList<Entry> unsorted = new ArrayList<>() {{
32             add(new Entry("ddd", "aaa"));
33             add(new Entry("bbb", "bbb"));
34             add(new Entry("ccc", "ccc"));
35             add(new Entry("aaa", "aaa"));
36             add(new Entry("ccc", "aaa"));
37             add(new Entry("bbb", "aaa"));
38         }};
39
40         ArrayList<Entry> sorted = new ArrayList<>() {{
41             add(new Entry("aaa", "aaa"));
42             add(new Entry("bbb", "aaa"));
43             add(new Entry("bbb", "bbb"));
44             add(new Entry("ccc", "aaa"));
45             add(new Entry("ccc", "ccc"));
46             add(new Entry("ddd", "aaa"));
47         }};
48         ArrayList<Entry> ret = mc.sortData(unsorted);
49
50         TestUtil.compareArrayOfEntries(sorted, ret);
51     }
52

```

```

53     @Test
54     public void setFTest(){
55         ModuleF newF = mock(ModuleF.class);
56         mc.setF(newF);
57     }
58
59     // to cover line 28 in ModuleC
60     @Test
61     public void sortFourElementsTest(){
62         ArrayList<Entry> unsorted = new ArrayList<>() {{
63             add(new Entry("ccc", "ccc"));
64             add(new Entry("aaa", "aaa"));
65             add(new Entry("bbb", "ddd"));
66             add(new Entry("bbb", "aaa"));
67         }};
68
69         ArrayList<Entry> sorted = new ArrayList<>() {{
70             add(new Entry("aaa", "aaa"));
71             add(new Entry("bbb", "aaa"));
72             add(new Entry("bbb", "ddd"));
73             add(new Entry("ccc", "ccc"));
74         }};
75
76         ArrayList<Entry> ret = mc.sortData(unsorted);
77
78         TestUtil.compareArrayOfEntries(sorted, ret);
79     }
80
81 }

```

## A.4 Module D

```

1  package bigbang;
2
3  import TestUtil.TestUtil;
4  import data.Entry;
5  import modules.ModuleD;
6  import modules.ModuleF;
7  import modules.ModuleG;
8  import org.junit.jupiter.api.*;
9
10 import java.util.ArrayList;
11
12 import static org.mockito.Mockito.*;
13
14

```

```

15 public class TestD {
16
17     ModuleF mf;
18     ModuleG mg;
19     ModuleD md;
20
21     final static String TEST_NAME = "testName";
22     final static String TEST_NUMBER = "testNumber";
23     final static String TEST_FILENAME = "testFilename";
24     final static int TEST_INDEX = 5;
25
26     ArrayList<Entry> expected;
27
28     @BeforeEach
29     public void setUp(){
30         mf = mock(ModuleF.class);
31         mg = mock(ModuleG.class);
32
33         md = new ModuleD(mf, mg);
34
35         expected = new ArrayList<>() {{
36             for (int i = 0; i < 10; i += 1)
37                 add(new Entry(TEST_NAME + i, TEST_NUMBER + i));
38         }};
39     }
40
41     @AfterEach
42     public void after(TestInfo testInfo){
43         if(testInfo.getTags().contains("SkipAfter")) {
44             return;
45         }
46         verify(mf, times(1)).displayData(any());
47         verify(mg, times(1)).updateData(anyString(), any());
48     }
49
50     @Test
51     public void insertDataTest(){
52         ArrayList<Entry> ret=
53             ↪ md.insertData((ArrayList<Entry>)expected.clone(),
54             ↪ TEST_NAME, TEST_NUMBER, TEST_FILENAME);
55
56         expected.add(new Entry(TEST_NAME, TEST_NUMBER));
57
58         TestUtil.compareArrayOfEntries(expected, ret);
59     }
60 }

```

```

59     @Test
60     public void updateDataTest(){
61         ArrayList<Entry> ret = md.updateData((ArrayList<Entry>)
        ↪ expected.clone(), TEST_INDEX, TEST_NAME, TEST_NUMBER,
        ↪ TEST_FILENAME);
62
63         expected.set(TEST_INDEX, new Entry(TEST_NAME,
        ↪ TEST_NUMBER));
64
65         TestUtils.compareArrayOfEntries(expected, ret);
66     }
67
68     @Test
69     public void deleteDataTest(){
70         ArrayList<Entry> ret = md.deleteData((ArrayList<Entry>)
        ↪ expected.clone(), TEST_INDEX, TEST_FILENAME);
71
72         expected.remove(TEST_INDEX);
73
74         TestUtils.compareArrayOfEntries(expected, ret);
75     }
76
77     @Tag("SkipAfter")
78     @Test
79     public void setFTest(){
80         ModuleF newF = mock(ModuleF.class);
81         md.setF(newF);
82     }
83
84     @Tag("SkipAfter")
85     @Test
86     public void setGTest(){
87         ModuleG newG = mock(ModuleG.class);
88         md.setG(newG);
89     }
90
91 }

```

## A.5 Module E

```

1  package bigbang;
2  import static org.junit.jupiter.api.Assertions.*;
3
4  import modules.ModuleE;
5  import org.junit.jupiter.api.Test;
6

```

```

7 public class TestE {
8
9     @Test
10    public void testE(){
11        assertThrows(ModuleE.DataBaseExitException.class , ()->
12            ↪ new ModuleE().exitProgram());
13    }
14 }

```

## A.6 Module F

```

1 package bigbang;
2
3 import data.Entry;
4 import modules.ModuleF;
5 import org.junit.jupiter.api.Test;
6 import static org.junit.jupiter.api.Assertions.*;
7
8 import java.io.ByteArrayOutputStream;
9 import java.io.PrintStream;
10 import java.util.ArrayList;
11
12 public class TestF {
13
14
15     @Test
16    public void testModuleF(){
17        ByteArrayOutputStream stdout = new
18            ↪ ByteArrayOutputStream();
19
20        ModuleF mf = new ModuleF();
21        mf.setOutputStream(new PrintStream(stdout));
22
23        ArrayList<Entry> entries = new ArrayList<>();
24        entries.add(new Entry("name1", "number1"));
25        entries.add(new Entry("name2", "number2"));
26        entries.add(new Entry("name3", "number3"));
27        entries.add(new Entry("name4", "number4"));
28        entries.add(new Entry("name5", "number5"));
29
30        mf.displayData(entries);
31
32        assertEquals("""
33 Current Data:
34 1 name1, number1

```

```

34     2 name2, number2
35     3 name3, number3
36     4 name4, number4
37     5 name5, number5
38     "", stdout.toString());
39     }
40 }

```

## A.7 Module G

```

1  package bigbang;
2
3  import data.Entry;
4  import modules.ModuleG;
5  import org.junit.jupiter.api.*;
6  import static org.junit.jupiter.api.Assertions.*;
7
8  import java.io.ByteArrayOutputStream;
9  import java.io.File;
10 import java.io.IOException;
11 import java.io.PrintStream;
12 import java.nio.file.Files;
13 import java.nio.file.Paths;
14 import java.util.ArrayList;
15
16 public class TestG {
17
18     static String FILENAME = "GTEST_FILE";
19     static File f;
20     static ModuleG mg;
21
22     @BeforeEach
23     public void createFile(){
24         f = new File(FILENAME);
25         mg = new ModuleG();
26     }
27
28     @AfterEach
29     public void deleteFile(){
30         f.delete();
31     }
32
33     @Test
34     public void testModuleG() throws IOException {
35
36         ArrayList<Entry> entries = new ArrayList<>();

```

```

37         entries.add(new Entry("name1", "number1"));
38         entries.add(new Entry("name2", "number2"));
39         entries.add(new Entry("name3", "number3"));
40         entries.add(new Entry("name4", "number4"));
41         entries.add(new Entry("name5", "number5"));
42
43         mg.updateData(FILENAME, entries);
44
45         // todo test output
46         assertEquals("",
47 name1,number1
48 name2,number2
49 name3,number3
50 name4,number4
51 name5,number5
52 "", Files.readString(Paths.get(FILENAME)));
53     }
54
55     @Test
56     public void testModuleGFail() {
57         ByteArrayOutputStream stdout= new ByteArrayOutputStream();
58         System.setOut(new PrintStream(stdout));
59         mg.updateData("", new ArrayList<Entry>());
60
61         assertEquals("Error updating DB File.\n", stdout.toString());
62     }
63 }

```

## A.8 Test Everything

```

1 package bigbang;
2
3 import modules.*;
4 import org.junit.jupiter.api.AfterEach;
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.Test;
7 import org.mockito.Mockito;
8
9 import java.io.ByteArrayOutputStream;
10 import java.io.File;
11 import java.io.IOException;
12 import java.io.PrintStream;
13 import java.nio.file.Files;
14 import java.nio.file.Paths;
15
16 import static org.junit.jupiter.api.Assertions.assertEquals;

```

```

17 import static org.junit.jupiter.api.Assertions.assertThrows;
18 import static org.mockito.ArgumentMatchers.*;
19 import static org.mockito.Mockito.*;
20
21
22 public class Test_Everything {
23
24     ModuleA ma;
25     ModuleB mb;
26     ModuleC mc;
27     ModuleD md;
28     ModuleE me;
29     ModuleF mf;
30     ModuleG mg;
31     final static String TEST_FILENAME = "testFileName";
32     final static String NONEXISTENT_FILE = "nonExistentFile";
33
34     File f;
35
36     ByteArrayOutputStream stdout;
37
38     @BeforeEach
39     public void setUp() throws IOException {
40         me = Mockito.spy(new ModuleE());
41         mf = Mockito.spy(new ModuleF());
42         mg = Mockito.spy(new ModuleG());
43
44         mb = Mockito.spy(new ModuleB(mf));
45         mc = Mockito.spy(new ModuleC(mf));
46         md = Mockito.spy(new ModuleD(mf, mg));
47
48         ma = new ModuleA(mb, mc, md, me);
49
50         newStdout();
51
52         f = new File(TEST_FILENAME);
53         f.createNewFile();
54     }
55
56     @AfterEach
57     public void deleteFile() throws IOException {
58
59         ↪ System.out.println(Files.readString(Paths.get(TEST_FILENAME)));
60         f.delete();
61     }

```



```

62     public void newStdout(){
63         stdout = new ByteArrayOutputStream();
64         ma.setOutputStream(new PrintStream(stdout));
65     }
66
67     @Test
68     public void testHelp() throws ModuleE.DataBaseExitException {
69         ma.run(new String[]{"help"});
70
71         final String help = "Available Commands: \n" +
72             "load <filepath>\n" +
73             "add <name> <number>\n" +
74             "update <index> <name> <number>\n" +
75             "delete <index>\n" +
76             "sort\n" +
77             "exit";
78
79         assertEquals(help + "\n", stdout.toString());
80     }
81
82     public void load() throws ModuleE.DataBaseExitException {
83         // Mockito.when(md.insertData(any(), anyString()),
84         ↪ anyString(), anyString())).thenReturn(new
85         ↪ ArrayList<Entry>());
86         ma.run(new String[]{"load", TEST_FILENAME});
87     }
88
89     @Test
90     public void testLoad() throws ModuleE.DataBaseExitException {
91         load();
92         verify(mb, times(1)).loadFile(anyString());
93     }
94
95     @Test
96     public void testLoadNoArgument() throws
97         ↪ ModuleE.DataBaseExitException {
98         ma.run(new String[]{"load"});
99         assertEquals("Malformed command!\n", stdout.toString());
100         verify(mb, never()).loadFile(anyString());
101     }
102
103     @Test
104     public void testLoadBReturnsNull() throws
105         ↪ ModuleE.DataBaseExitException {

```

```

103 //
104 ↪ Mockito.when(mb.loadFile(anyString())).thenReturn(null);
105     ma.run(new String[]{"load", NONEXISTENT_FILE});
106     verify(mb, times(1)).loadFile(anyString());
107 }
108
109 @Test
110 public void testAddNoData() throws
111     ↪ ModuleE.DataBaseExitException {
112     ma.run(new String[]{"add", "name", "number"});
113     assertEquals("No file loaded!\n", stdout.toString());
114     verify(md, never()).insertData(any(), anyString(),
115     ↪ anyString(), anyString());
116 }
117
118 @Test
119 public void testAdd() throws ModuleE.DataBaseExitException,
120     ↪ IOException {
121     // Mockito.when(md.insertData(any(), anyString(),
122     ↪ anyString(), anyString())).thenReturn(new
123     ↪ ArrayList<Entry>());
124     ma.run(new String[]{"load", TEST_FILENAME});
125     ma.run(new String[]{"add", "name", "number"});
126     verify(md, times(1)).insertData(any(), anyString(),
127     ↪ anyString(), anyString());
128
129     assertEquals("name,number\n",
130     ↪ Files.readString(Paths.get(TEST_FILENAME)));
131 }
132
133 @Test
134 public void testAddDreturnsNull() throws
135     ↪ ModuleE.DataBaseExitException {
136
137     // through static analysis, we can see that d.insertdata
138     ↪ will never return null in practice
139
140     // Mockito.when(md.insertData(any(), anyString(),
141     ↪ anyString(), anyString())).thenReturn(null);
142     // ma.run(new String[]{"load", NONEXISTENT_FILE});
143     // ma.run(new String[]{"add", "name", "number"});
144     // verify(md, times(1)).insertData(any(), anyString(),
145     ↪ anyString(), anyString());
146 }
147
148 @Test

```

```

137     public void testAddNoArgument() throws
        ↪ ModuleE.DataBaseExitException {
138         ma.run(new String[]{"load", TEST_FILENAME});
139         ma.run(new String[]{"add"});
140         assertEquals("Malformed command!\n", stdout.toString());
141         verify(md, never()).insertData(any(), anyString(),
            ↪ anyString(), anyString());
142     }
143
144     @Test
145     public void testSort() throws ModuleE.DataBaseExitException,
        ↪ IOException {
146         ma.run(new String[]{"load", TEST_FILENAME});
147         ma.run(new String[]{"add", "ddd", "aaa"});
148         ma.run(new String[]{"add", "bbb", "bbb"});
149         ma.run(new String[]{"add", "ccc", "ccc"});
150         ma.run(new String[]{"add", "aaa", "aaa"});
151         ma.run(new String[]{"add", "ccc", "aaa"});
152         ma.run(new String[]{"add", "bbb", "aaa"});
153
154
155         ma.run(new String[]{"sort"});
156         verify(mc, times(1)).sortData(any());
157
158         assertEquals("""
159         aaa,aaa
160         bbb,aaa
161         bbb,bbb
162         ccc,aaa,
163         ccc,ccc
164         ddd,aaa""", Files.readString(Paths.get(TEST_FILENAME)));
165     }
166
167     @Test
168     public void testSortNoData() throws
        ↪ ModuleE.DataBaseExitException {
169         ma.run(new String[]{"sort"});
170         assertEquals("No file loaded!\n", stdout.toString());
171         verify(mc, never()).sortData(any());
172     }
173
174     @Test
175     public void testSortCReturnsNull() throws
        ↪ ModuleE.DataBaseExitException {
176         // in practice, moduleC never returns null

```

```

177         // it can throw a NullPointerException if the input data
178         ↪ is null
179         // but the program does a null check on line 56 in module
180         ↪ A
181         // so it will never return null
182
183     // Mockito.when(mc.sortData(any())).thenReturn(null);
184     // ma.run(new String[]{"load", TEST_FILENAME});
185     // ma.run(new String[]{"sort"});
186     // verify(mc, times(1)).sortData(any());
187 }
188
189 @Test
190 public void testUpdate() throws
191     ↪ ModuleE.DataBaseExitException, IOException {
192     ma.run(new String[]{"load", TEST_FILENAME});
193     ma.run(new String[]{"add", "aaa", "aaa"});
194     ma.run(new String[]{"add", "bbb", "aaa"});
195     ma.run(new String[]{"add", "bbb", "bbb"});
196     ma.run(new String[]{"add", "ccc", "aaa"});
197     ma.run(new String[]{"add", "ccc", "ccc"});
198     ma.run(new String[]{"add", "ddd", "aaa"});
199
200     ma.run(new String[]{"update", "5", "new", "data"});
201
202     verify(md, times(1)).updateData(any(), anyInt(),
203         ↪ anyString(), anyString(), anyString());
204
205     assertEquals("
206     aaa,aaa
207     bbb,aaa
208     bbb,bbb
209     ccc,aaa
210     new,data
211     ddd,aaa
212     """, Files.readString(Paths.get(TEST_FILENAME)));
213
214     // huh this seems to pass
215     // talk about in the report how on line 138 they do
216     ↪ index-2
217     // and how the error cancels out that way to be correct
218     ↪ in the end
219 }
220
221 @Test

```

```

216     public void testUpdateInvalidArguments() throws
        ↳ ModuleE.DataBaseExitException {
217         ma.run(new String[]{"load", TEST_FILENAME});
218         ma.run(new String[]{"update", "arg1", "arg2", "arg3"});
219         verify(md, never()).updateData(any(), anyInt(),
            ↳ anyString(), anyString(), anyString());
220     }
221
222
223     @Test
224     public void testUpdateNoData() throws
        ↳ ModuleE.DataBaseExitException {
225         ma.run(new String[]{"update", "1", "2", "3"});
226         assertEquals("No file loaded!\n", stdout.toString());
227         verify(md, never()).updateData(any(), anyInt(),
            ↳ anyString(), anyString(), anyString());
228     }
229
230     @Test
231     public void testUpdateCReturnsNull() throws
        ↳ ModuleE.DataBaseExitException {
232
233         //      in practice, we see using static analysis that moduleC
        ↳ can never return null
234
235         //      Mockito.when(md.updateData(any(), anyInt(),
        ↳ anyString(), anyString(), anyString())).thenReturn(null);
236         //      ma.run(new String[]{"load", NONEXISTENT_FILE});
237         //      ma.run(new String[]{"update", "1", "2", "3"});
238         //      verify(md, times(1)).updateData(any(), anyInt(),
        ↳ anyString(), anyString(), anyString());
239     }
240
241     @Test
242     public void testUpdateNoArgument() throws
        ↳ ModuleE.DataBaseExitException {
243         ma.run(new String[]{"load", TEST_FILENAME});
244         ma.run(new String[]{"update"});
245         assertEquals("Malformed command!\n", stdout.toString());
246         verify(md, never()).updateData(any(), anyInt(),
            ↳ anyString(), anyString(), anyString());
247     }
248
249     @Test
250     public void testDelete() throws
        ↳ ModuleE.DataBaseExitException, IOException {

```

```

251     ma.run(new String[]{"load", TEST_FILENAME});
252     ma.run(new String[]{"add", "aaa", "aaa"});
253     ma.run(new String[]{"add", "bbb", "aaa"});
254     ma.run(new String[]{"add", "bbb", "bbb"});
255     ma.run(new String[]{"add", "ccc", "aaa"});
256     ma.run(new String[]{"add", "ccc", "ccc"});
257     ma.run(new String[]{"add", "ddd", "aaa"});
258
259     ma.run(new String[]{"delete", "5"});
260     verify(md, times(1)).deleteData(any(), anyInt(),
        ↪ anyString());
261
262     assertEquals("""
263     aaa,aaa
264     bbb,aaa
265     bbb,bbb
266     ccc,aaa
267     ddd,aaa
268     """, Files.readString(Paths.get(TEST_FILENAME)));
269 }
270
271 @Test
272 public void testDeleteInvalidArguments() throws
    ↪ ModuleE.DataBaseExitException {
273     ma.run(new String[]{"load", TEST_FILENAME});
274     ma.run(new String[]{"delete", "arg1"});
275     verify(md, never()).deleteData(any(), anyInt(),
        ↪ anyString());
276 }
277
278 @Test
279 public void testDeleteNoData() throws
    ↪ ModuleE.DataBaseExitException {
280     ma.run(new String[]{"delete"});
281     assertEquals("No file loaded!\n", stdout.toString());
282     verify(md, never()).deleteData(any(), anyInt(),
        ↪ anyString());
283 }
284
285 @Test
286 public void testDeleteDReturnsNull() throws
    ↪ ModuleE.DataBaseExitException {
287
288     // using static analysis, we see that in practice,
        ↪ deleteData will never return null

```

```

289 //      Mockito.when(md.deleteData(any(), anyInt(),
    ↪ anyString())).thenReturn(null);
290 //      ma.run(new String[]{"load", NONEXISTENT_FILE});
291 //      ma.run(new String[]{"delete", "1"});
292 //      verify(md, times(1)).deleteData(any(), anyInt(),
    ↪ anyString());
293 }
294
295 @Test
296 public void testDeleteNoArgument() throws
    ↪ ModuleE.DataBaseExitException {
297     ma.run(new String[]{"load", TEST_FILENAME});
298     ma.run(new String[]{"delete"});
299     assertEquals("Malformed command!\n", stdout.toString());
300     verify(md, never()).deleteData(any(), anyInt(),
    ↪ anyString());
301 }
302
303 @Test
304 public void testExit() throws ModuleE.DataBaseExitException {
305 //      Mockito.doThrow(ModuleE.DataBaseExitException.class).when(me).exitProgram();
    ↪ assertThrows(ModuleE.DataBaseExitException.class, ()
    ↪ ->ma.run(new String[]{"exit"}));
306 // line 147 does not get covered, because the program
    ↪ exits
307 }
308
309 @Test
310 public void unknownCommand() throws
    ↪ ModuleE.DataBaseExitException {
311     ma.run(new String[]{"thiscommanddoesntexist"});
312 }
313 }
314 }

```

## B Bottom Up Testing Strategy

### B.1 Test F

```

1 package bottumUp;
2
3 import data.Entry;
4 import modules.ModuleF;
5 import org.junit.jupiter.api.Test;
6

```

```

7  import java.io.ByteArrayOutputStream;
8  import java.io.PrintStream;
9  import java.util.ArrayList;
10
11  import static org.junit.jupiter.api.Assertions.assertEquals;
12
13  public class Test00_F {
14
15
16      @Test
17      public void testModuleF(){
18          ByteArrayOutputStream stdout = new
19              ↳ ByteArrayOutputStream();
19
20          ModuleF mf = new ModuleF();
21          mf.setOutputStream(new PrintStream(stdout));
22
23          ArrayList<Entry> entries = new ArrayList<>();
24          entries.add(new Entry("name1", "number1"));
25          entries.add(new Entry("name2", "number2"));
26          entries.add(new Entry("name3", "number3"));
27          entries.add(new Entry("name4", "number4"));
28          entries.add(new Entry("name5", "number5"));
29
30          mf.displayData(entries);
31
32          assertEquals("""
33  Current Data:
34  1 name1, number1
35  2 name2, number2
36  3 name3, number3
37  4 name4, number4
38  5 name5, number5
39  """, stdout.toString());
40      }
41  }

```

## B.2 Test G

```

1  package bottumUp;
2
3  import data.Entry;
4  import modules.ModuleG;
5  import org.junit.jupiter.api.AfterEach;
6  import org.junit.jupiter.api.BeforeEach;
7  import org.junit.jupiter.api.Test;

```



```

8
9  import java.io.ByteArrayOutputStream;
10 import java.io.File;
11 import java.io.IOException;
12 import java.io.PrintStream;
13 import java.nio.file.Files;
14 import java.nio.file.Paths;
15 import java.util.ArrayList;
16
17 import static org.junit.jupiter.api.Assertions.assertEquals;
18
19 public class Test01_G {
20
21     static String FILENAME = "GTEST_FILE";
22     static File f;
23     static ModuleG mg;
24
25     @BeforeEach
26     public void createFile(){
27         f = new File(FILENAME);
28         mg = new ModuleG();
29     }
30
31     @AfterEach
32     public void deleteFile(){
33         f.delete();
34     }
35
36     @Test
37     public void testModuleG() throws IOException {
38
39         ArrayList<Entry> entries = new ArrayList<>();
40         entries.add(new Entry("name1", "number1"));
41         entries.add(new Entry("name2", "number2"));
42         entries.add(new Entry("name3", "number3"));
43         entries.add(new Entry("name4", "number4"));
44         entries.add(new Entry("name5", "number5"));
45
46         mg.updateData(FILENAME, entries);
47
48         // todo test output
49         assertEquals("""
50 name1,number1
51 name2,number2
52 name3,number3
53 name4,number4

```

```

54     name5,number5
55     """, Files.readString(Paths.get(FILENAME)));
56     }
57
58     @Test
59     public void testModuleGFail() {
60         ByteArrayOutputStream stdout= new
        ↳ ByteArrayOutputStream();
61         System.setOut(new PrintStream(stdout));
62         mg.updateData("", new ArrayList<Entry>());
63
64         assertEquals("Error updating DB File.\n",
        ↳ stdout.toString());
65     }
66 }

```

### B.3 Test BF

```

1  package bottumUp;
2
3  import TestUtil.TestUtil;
4  import modules.ModuleB;
5  import modules.ModuleF;
6  import org.junit.jupiter.api.AfterEach;
7  import org.junit.jupiter.api.BeforeEach;
8  import org.junit.jupiter.api.Test;
9
10 import java.io.File;
11 import java.io.IOException;
12 import java.nio.file.Files;
13 import data.Entry;
14 import java.nio.file.Paths;
15 import java.util.ArrayList;
16
17 import static org.mockito.Mockito.mock;
18
19 // IO Exception catching is not tested
20 // Explain in the report that it's just calling a library
   ↳ function
21 // Also, the scenario was never encountered e.g. reading
   ↳ /etc/shadow
22
23 public class Test02_BF {
24
25     ModuleB mb;
26     ModuleF mf;

```

```

27     final static String TEST_FILENAME = "BFTEST_FILE";
28     static File f;
29
30     @BeforeEach
31     public void setUp() throws IOException {
32         mf = new ModuleF();
33         mb = new ModuleB(mf);
34
35         f = new File(TEST_FILENAME);
36         f.createNewFile();
37         Files.writeString(Paths.get(TEST_FILENAME), "")
38         This
39         is,some
40         test
41         data""");
42     }
43
44     @AfterEach
45     public void tearDown(){
46         f.delete();
47     }
48
49     @Test
50     public void loadFileTestValidFile() {
51         ArrayList<Entry> ret = mb.loadFile(TEST_FILENAME);
52
53         ArrayList<Entry> expected = new ArrayList<>() {{
54             add(new Entry("is", "some"));
55         }};
56
57         TestUtil.compareArrayOfEntries(expected, ret);
58     }
59
60     @Test
61     public void loadFileTestInvalidFile() {
62         mb.loadFile("/");
63     }
64
65     @Test
66     public void loadFileTestFileNotFound(){
67         mb.loadFile("");
68     }
69
70     @Test
71     public void setFTest(){
72         ModuleF newF = new ModuleF();

```

```

73         mb.setF(newF);
74     }
75 }

```

## B.4 Test CF

```

1  package bottumUp;
2
3  import TestUtil.TestUtil;
4  import data.Entry;
5  import modules.ModuleC;
6  import modules.ModuleF;
7  import org.junit.jupiter.api.BeforeEach;
8  import org.junit.jupiter.api.Test;
9
10 import java.util.ArrayList;
11
12 public class Test03_CF {
13
14     ModuleF mf;
15     ModuleC mc;
16
17     @BeforeEach
18     public void setUp(){
19         mf = new ModuleF();
20         mc = new ModuleC(mf);
21     }
22
23     @Test
24     public void sortDataTest(){
25
26         final String TEST_NAME = "testName";
27         final String TEST_NUMBER = "testNumber";
28
29         ArrayList<Entry> unsorted = new ArrayList<>() {{
30             add(new Entry("ddd", "aaa"));
31             add(new Entry("bbb", "bbb"));
32             add(new Entry("ccc", "ccc"));
33             add(new Entry("aaa", "aaa"));
34             add(new Entry("ccc", "aaa"));
35             add(new Entry("bbb", "aaa"));
36         }};
37
38         ArrayList<Entry> sorted = new ArrayList<>() {{
39             add(new Entry("aaa", "aaa"));
40             add(new Entry("bbb", "aaa"));

```

```

41         add(new Entry("bbb", "bbb"));
42         add(new Entry("ccc", "aaa"));
43         add(new Entry("ccc", "ccc"));
44         add(new Entry("ddd", "aaa"));
45     });
46     ArrayList<Entry> ret = mc.sortData(unsorted);
47
48     TestUtil.compareArrayOfEntries(sorted, ret);
49 }
50
51 @Test
52 public void setFTest(){
53     ModuleF newF = new ModuleF();
54     mc.setF(newF);
55 }
56
57 // to cover line 28 in ModuleC
58 @Test
59 public void sortFourElementsTest(){
60     ArrayList<Entry> unsorted = new ArrayList<>() {{
61         add(new Entry("ccc", "ccc"));
62         add(new Entry("aaa", "aaa"));
63         add(new Entry("bbb", "ddd"));
64         add(new Entry("bbb", "aaa"));
65     }};
66
67     ArrayList<Entry> sorted = new ArrayList<>() {{
68         add(new Entry("aaa", "aaa"));
69         add(new Entry("bbb", "aaa"));
70         add(new Entry("bbb", "ddd"));
71         add(new Entry("ccc", "ccc"));
72     }};
73
74     ArrayList<Entry> ret = mc.sortData(unsorted);
75
76     TestUtil.compareArrayOfEntries(sorted, ret);
77 }
78
79 }

```

## B.5 Test DFG

```

1 package bottumUp;
2
3 import TestUtil.TestUtil;
4 import data.Entry;

```

```

5  import modules.ModuleD;
6  import modules.ModuleF;
7  import modules.ModuleG;
8  import org.junit.jupiter.api.*;
9
10 import java.util.ArrayList;
11
12 import static org.mockito.Mockito.*;
13
14
15 public class Test04_DFG {
16
17     ModuleF mf;
18     ModuleG mg;
19     ModuleD md;
20
21     final static String TEST_NAME = "testName";
22     final static String TEST_NUMBER = "testNumber";
23     final static String TEST_FILENAME = "testFilename";
24     final static int TEST_INDEX = 5;
25
26     ArrayList<Entry> expected;
27
28     @BeforeEach
29     public void setUp(){
30         mf = spy(new ModuleF());
31         mg = spy(new ModuleG());
32
33         md = new ModuleD(mf, mg);
34
35         expected = new ArrayList<>() {{
36             for (int i = 0; i < 10; i += 1)
37                 add(new Entry(TEST_NAME + i, TEST_NUMBER + i));
38         }};
39     }
40
41     @AfterEach
42     public void after(TestInfo testInfo){
43         if(testInfo.getTags().contains("SkipAfter")) {
44             return;
45         }
46         verify(mf, times(1)).displayData(any());
47         verify(mg, times(1)).updateData(anyString(), any());
48     }
49
50     @Test

```

```

51     public void insertDataTest(){
52         ArrayList<Entry> ret=
53             ↳ md.insertData((ArrayList<Entry>)expected.clone(),
54             ↳ TEST_NAME, TEST_NUMBER, TEST_FILENAME);
55
56         expected.add(new Entry(TEST_NAME, TEST_NUMBER));
57
58         TestUtils.compareArrayOfEntries(expected, ret);
59     }
60
61     @Test
62     public void updateDataTest(){
63         ArrayList<Entry> ret = md.updateData((ArrayList<Entry>)
64             ↳ expected.clone(), TEST_INDEX, TEST_NAME, TEST_NUMBER,
65             ↳ TEST_FILENAME);
66
67         expected.set(TEST_INDEX, new Entry(TEST_NAME,
68             ↳ TEST_NUMBER));
69
70         TestUtils.compareArrayOfEntries(expected, ret);
71     }
72
73     @Test
74     public void deleteDataTest(){
75         ArrayList<Entry> ret = md.deleteData((ArrayList<Entry>)
76             ↳ expected.clone(), TEST_INDEX, TEST_FILENAME);
77
78         expected.remove(TEST_INDEX);
79
80         TestUtils.compareArrayOfEntries(expected, ret);
81     }
82
83     @Tag("SkipAfter")
84     @Test
85     public void setFTest(){
86         ModuleF newF = new ModuleF();
87         md.setF(newF);
88     }
89
90     @Tag("SkipAfter")
91     @Test
92     public void setGTest(){
93         ModuleG newG = new ModuleG();
94         md.setG(newG);
95     }

```

```
91 }
```

## B.6 Test E

```
1 package bottumUp;
2
3 import modules.ModuleE;
4 import org.junit.jupiter.api.Test;
5
6 import static org.junit.jupiter.api.Assertions.assertThrows;
7
8 public class Test05_E {
9
10     @Test
11     public void testE(){
12         assertThrows(ModuleE.DataBaseExitException.class , ()->
13             ↪ new ModuleE().exitProgram());
14     }
15 }
```

## B.7 Test Everything

```
1 package bottumUp;
2
3 import modules.*;
4 import org.junit.jupiter.api.AfterEach;
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.Test;
7 import org.mockito.Mockito;
8
9 import java.io.ByteArrayOutputStream;
10 import java.io.File;
11 import java.io.IOException;
12 import java.io.PrintStream;
13 import java.nio.file.Files;
14 import java.nio.file.Paths;
15
16 import static org.junit.jupiter.api.Assertions.assertEquals;
17 import static org.junit.jupiter.api.Assertions.assertThrows;
18 import static org.mockito.ArgumentMatchers.*;
19 import static org.mockito.Mockito.*;
20
21
22 public class Test06_Everything {
23 }
```



```

24     ModuleA ma;
25     ModuleB mb;
26     ModuleC mc;
27     ModuleD md;
28     ModuleE me;
29     ModuleF mf;
30     ModuleG mg;
31     final static String TEST_FILENAME = "testFileName";
32     final static String NONEXISTENT_FILE = "nonExistentFile";
33
34     File f;
35
36     ByteArrayOutputStream stdout;
37
38     @BeforeEach
39     public void setUp() throws IOException {
40         me = Mockito.spy(new ModuleE());
41         mf = Mockito.spy(new ModuleF());
42         mg = Mockito.spy(new ModuleG());
43
44         mb = Mockito.spy(new ModuleB(mf));
45         mc = Mockito.spy(new ModuleC(mf));
46         md = Mockito.spy(new ModuleD(mf, mg));
47
48         ma = new ModuleA(mb, mc, md, me);
49
50         newStdout();
51
52         f = new File(TEST_FILENAME);
53         f.createNewFile();
54     }
55
56     @AfterEach
57     public void deleteFile() throws IOException {
58
59         ↪ System.out.println(Files.readString(Paths.get(TEST_FILENAME)));
60         f.delete();
61     }
62
63     public void newStdout(){
64         stdout = new ByteArrayOutputStream();
65         ma.setOutputStream(new PrintStream(stdout));
66     }
67
68     @Test
69     public void testHelp() throws ModuleE.DataBaseExitException {

```

```

69         ma.run(new String[]{"help"});
70
71         final String help = "Available Commands: \n" +
72             "load <filepath>\n" +
73             "add <name> <number>\n" +
74             "update <index> <name> <number>\n" +
75             "delete <index>\n" +
76             "sort\n" +
77             "exit";
78
79         assertEquals(help + "\n", stdout.toString());
80
81     }
82
83     public void load() throws ModuleE.DataBaseExitException {
84         // Mockito.when(md.insertData(any(), anyString()),
85         ↪ anyString(), anyString())).thenReturn(new
86         ↪ ArrayList<Entry>());
87         ma.run(new String[]{"load", TEST_FILENAME});
88     }
89
90     @Test
91     public void testLoad() throws ModuleE.DataBaseExitException {
92         load();
93         verify(mb, times(1)).loadFile(anyString());
94     }
95
96     @Test
97     public void testLoadNoArgument() throws
98         ↪ ModuleE.DataBaseExitException {
99         ma.run(new String[]{"load"});
100         assertEquals("Malformed command!\n", stdout.toString());
101         verify(mb, never()).loadFile(anyString());
102     }
103
104     @Test
105     public void testLoadBReturnsNull() throws
106         ↪ ModuleE.DataBaseExitException {
107         // Mockito.when(mb.loadFile(anyString())).thenReturn(null);
108         ↪ ma.run(new String[]{"load", NONEXISTENT_FILE});
109         verify(mb, times(1)).loadFile(anyString());
110     }
111
112     @Test

```



```

142     }
143
144     @Test
145     public void testSort() throws ModuleE.DataBaseExitException,
146         ↳ IOException {
147         ma.run(new String[]{"load", TEST_FILENAME});
148         ma.run(new String[]{"add", "ddd", "aaa"});
149         ma.run(new String[]{"add", "bbb", "bbb"});
150         ma.run(new String[]{"add", "ccc", "ccc"});
151         ma.run(new String[]{"add", "aaa", "aaa"});
152         ma.run(new String[]{"add", "ccc", "aaa"});
153         ma.run(new String[]{"add", "bbb", "aaa"});
154
155         ma.run(new String[]{"sort"});
156         verify(mc, times(1)).sortData(any());
157
158         assertEquals("",
159             aaa,aaa
160             bbb,aaa
161             bbb,bbb
162             ccc,aaa,
163             ccc,ccc
164             ddd,aaa"", Files.readString(Paths.get(TEST_FILENAME)));
165     }
166
167     @Test
168     public void testSortNoData() throws
169         ↳ ModuleE.DataBaseExitException {
170         ma.run(new String[]{"sort"});
171         assertEquals("No file loaded!\n", stdout.toString());
172         verify(mc, never()).sortData(any());
173     }
174
175     @Test
176     public void testSortCReturnsNull() throws
177         ↳ ModuleE.DataBaseExitException {
178         // in practice, moduleC never returns null
179         // it can throw a NullPointerException if the input data
180         ↳ is null
181         // but the program does a null check on line 56 in module
182         ↳ A
183         // so it will never return null
184
185         // Mockito.when(mc.sortData(any())).thenReturn(null);
186         // ma.run(new String[]{"load", TEST_FILENAME});

```

```

183 //      ma.run(new String[]{"sort"});
184 //      verify(mc, times(1)).sortData(any());
185 }
186
187 @Test
188 public void testUpdate() throws
    ↪ ModuleE.DataBaseExitException, IOException {
189     ma.run(new String[]{"load", TEST_FILENAME});
190     ma.run(new String[]{"add", "aaa", "aaa"});
191     ma.run(new String[]{"add", "bbb", "aaa"});
192     ma.run(new String[]{"add", "bbb", "bbb"});
193     ma.run(new String[]{"add", "ccc", "aaa"});
194     ma.run(new String[]{"add", "ccc", "ccc"});
195     ma.run(new String[]{"add", "ddd", "aaa"});
196
197     ma.run(new String[]{"update", "5", "new", "data"});
198
199     verify(md, times(1)).updateData(any(), anyInt(),
    ↪ anyString(), anyString(), anyString());
200
201     assertEquals("""
202     aaa,aaa
203     bbb,aaa
204     bbb,bbb
205     ccc,aaa
206     new,data
207     ddd,aaa
208     """, Files.readString(Paths.get(TEST_FILENAME)));
209
210     // huh this seems to pass
211     // talk about in the report how on line 138 they do
    ↪ index-2
212     // and how the error cancels out that way to be correct
    ↪ in the end
213 }
214
215 @Test
216 public void testUpdateInvalidArguments() throws
    ↪ ModuleE.DataBaseExitException {
217     ma.run(new String[]{"load", TEST_FILENAME});
218     ma.run(new String[]{"update", "arg1", "arg2", "arg3"});
219     verify(md, never()).updateData(any(), anyInt(),
    ↪ anyString(), anyString(), anyString());
220 }
221
222

```

```

223     @Test
224     public void testUpdateNoData() throws
        ↳ ModuleE.DataBaseExitException {
225         ma.run(new String[]{"update", "1", "2", "3"});
226         assertEquals("No file loaded!\n", stdout.toString());
227         verify(md, never()).updateData(any(), anyInt(),
            ↳ anyString(), anyString(), anyString());
228     }
229
230     @Test
231     public void testUpdateCReturnsNull() throws
        ↳ ModuleE.DataBaseExitException {
232
233         //      in practce, we see using static analysis that moduleC
        ↳ can never return null
234
235         //      Mockito.when(md.updateData(any(), anyInt(),
        ↳ anyString(), anyString(), anyString())).thenReturn(null);
236         //      ma.run(new String[]{"load", NONEXISTENT_FILE});
237         //      ma.run(new String[]{"update", "1", "2", "3"});
238         //      verify(md, times(1)).updateData(any(), anyInt(),
        ↳ anyString(), anyString(), anyString());
239     }
240
241     @Test
242     public void testUpdateNoArgument() throws
        ↳ ModuleE.DataBaseExitException {
243         ma.run(new String[]{"load", TEST_FILENAME});
244         ma.run(new String[]{"update"});
245         assertEquals("Malformed command!\n", stdout.toString());
246         verify(md, never()).updateData(any(), anyInt(),
            ↳ anyString(), anyString(), anyString());
247     }
248
249     @Test
250     public void testDelete() throws
        ↳ ModuleE.DataBaseExitException, IOException {
251         ma.run(new String[]{"load", TEST_FILENAME});
252         ma.run(new String[]{"add", "aaa", "aaa"});
253         ma.run(new String[]{"add", "bbb", "aaa"});
254         ma.run(new String[]{"add", "bbb", "bbb"});
255         ma.run(new String[]{"add", "ccc", "aaa"});
256         ma.run(new String[]{"add", "ccc", "ccc"});
257         ma.run(new String[]{"add", "ddd", "aaa"});
258
259         ma.run(new String[]{"delete", "5"});

```

```

260         verify(md, times(1)).deleteData(any(), anyInt(),
           ↪ anyString());
261
262         assertEquals("""
263     aaa,aaa
264     bbb,aaa
265     bbb,bbb
266     ccc,aaa
267     ddd,aaa
268     """, Files.readString(Paths.get(TEST_FILENAME)));
269     }
270
271     @Test
272     public void testDeleteInvalidArguments() throws
           ↪ ModuleE.DataBaseExitException {
273         ma.run(new String[]{"load", TEST_FILENAME});
274         ma.run(new String[]{"delete", "arg1"});
275         verify(md, never()).deleteData(any(), anyInt(),
           ↪ anyString());
276     }
277
278     @Test
279     public void testDeleteNoData() throws
           ↪ ModuleE.DataBaseExitException {
280         ma.run(new String[]{"delete"});
281         assertEquals("No file loaded!\n", stdout.toString());
282         verify(md, never()).deleteData(any(), anyInt(),
           ↪ anyString());
283     }
284
285     @Test
286     public void testDeleteDReturnsNull() throws
           ↪ ModuleE.DataBaseExitException {
287
288         // using static analysis, we see that in practice,
           ↪ deleteData will never return null
289         // Mockito.when(md.deleteData(any(), anyInt(),
           ↪ anyString())).thenReturn(null);
290         // ma.run(new String[]{"load", NONEXISTENT_FILE});
291         // ma.run(new String[]{"delete", "1"});
292         // verify(md, times(1)).deleteData(any(), anyInt(),
           ↪ anyString());
293     }
294
295     @Test

```

```

296     public void testDeleteNoArgument() throws
        ↳ ModuleE.DataBaseExitException {
297         ma.run(new String[]{"load", TEST_FILENAME});
298         ma.run(new String[]{"delete"});
299         assertEquals("Malformed command!\n", stdout.toString());
300         verify(md, never()).deleteData(any(), anyInt(),
        ↳ anyString());
301     }
302
303     @Test
304     public void testExit() throws ModuleE.DataBaseExitException {
305         //
        ↳ Mockito.doThrow(ModuleE.DataBaseExitException.class).when(me).exitProgram();
306         assertThrows(ModuleE.DataBaseExitException.class, ()
        ↳ ->ma.run(new String[]{"exit"}));
307         // line 147 does not get covered, because the program
        ↳ exits
308     }
309
310     @Test
311     public void unknownCommand() throws
        ↳ ModuleE.DataBaseExitException {
312         ma.run(new String[]{"thiscommanddoesntexist"});
313     }
314 }

```