ECE 322
Lab Report 5

Arun Woosaree
1514457

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 written in Java 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 Bottum-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 handing 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, from which an IDE like Intellij or Eclipse should be able to install dependencies from and run the tests.

2 Task 1

For part one in this lab, the database application was tested using the big-bang testing strategy. Unit tests were made for each module A-F, and any time a module depended on another, the other module was mocked using mockito. The actual tests can be found in Appendix A, and are also in the zip archive in which this report was submitted. They can be found in *"Lab 5/Lab 5/src/test/java/bigbang"*. The test results are found on the next page.

bigbang in Lab_5:
62 total, 4 error, 5 failed, 53 passed

941 ms

Collapse | Expand

Test_Everything		876 ms
testSortNoData()	passed	769 ms
testLoadBReturnsNull()	passed	15 ms
testAdd()	passed	17 ms
testExit()	passed	4 ms
testHelp()	passed	2 ms
testLoad()	passed	3 ms
testSort()	failed	17 ms
testSortCReturnsNull()	passed	2 ms
testLoadNoArgument()	passed	2 ms
testUpdateNoData()	passed	3 ms
unknownCommand()	passed	2 ms
testDeleteNoData()	passed	3 ms
testAddDreturnsNull()	passed	1 ms
testAddNoData()	passed	3 ms
testDelete()	passed	7 ms
testDeleteNoArgument()	passed	3 ms
testAddNoArgument()	passed	3 ms
testUpdateNoArgument()	passed	3 ms
testDeleteDReturnsNull()	passed	2 ms
testUpdateCReturnsNull()	passed	2 ms
testUpdate()	passed	8 ms
testDeleteInvalidArguments()	error	3 ms
testUpdateInvalidArguments()	error	2 ms
TestF		1 ms
testModuleF()	failed	1 ms
TestG		3 ms
testModuleGFail()	passed	2 ms

bigbang in Lab_5:
62 total, 4 error, 5 failed, 53 passed

941 ms

Collapse | Expand

deleteDataTest()	passed	3 ms
setFTest()	passed	1 ms
setGTest()	passed	1 ms
TestE		1 ms
testE()	passed	1 ms
TestB		6 ms
loadFileTestInValidFile()	passed	2 ms
loadFileTestFileNotFound()	passed	1 ms
loadFileTestValidFile()	passed	2 ms
setFTest()	passed	1 ms
TestC		3 ms
sortFourElementsTest()	failed	1 ms
setFTest()	passed	1 ms
sortDataTest()	failed	1 ms
TestA		37 ms
testSortNoData()	passed	1 ms
testLoadBReturnsNull()	passed	6 ms
testAdd()	passed	2 ms
testExit()	passed	4 ms
testHelp()	passed	1 ms
testLoad()	passed	2 ms
testSort()	passed	1 ms
testSortCReturnsNull()	passed	2 ms
testLoadNoArgument()	passed	1 ms
testUpdateNoData()	passed	2 ms
unknownCommand()	passed	1 ms
testDeleteNoData()	passed	1 ms
testAddDreturnsNull()	passed	2 ms

bigbang in Lab_5:
62 total, 4 error, 5 failed, 53 passed

941 ms

Collapse | Expand

testDeleteDReturnsNull()	passed	2 ms
testUpdateCReturnsNull()	passed	1 ms
testUpdate()	passed	1 ms
testDeleteInvalidArguments()	error	1 ms
testUpdateInvalidArguments()	error	1 ms

The failing tests are explained as follows:

2.1 Test Everything

2.1.1 testSort()

This test fails because the file itself is not sorted. Instead, we have a case where Module A is calling Module C to sort the data, but Module A never tells another module to update the file with the sorted data, so the database file is left unmodified and therefore unsorted.

```
Input:
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
Expected :
aaa,aaa
bbb,aaa
bbb,bbb
ccc,aaa,
ccc,ccc
ddd,aaa
Actual :
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
```

2.1.2 testDeleteInvalidArguments()

The application is not equipped to handle arguments after the delete command which are non numeric. In such a case where a letter or word is inputted after the word 'delete', the program will ungracefully panic, instead of handling the error.

2.1.3 testUpdateInvalidArguments()

This test fails for the exact same reason for which testDeleteInvalidArguments() fails.

2.2 Module F

2.2.1 testModuleF()

When Module F displays data, it skips the first line, or the first entry. This is because the for loop on line 15 in Module F starts from $i = 1$, instead of $i = 0$, which means it starts from indexing the second element in the ArrayList, instead of the first element.

```
Expected:
Current Data:
1 name1, number1
2 name2, number2
3 name3, number3
4 name4, number4
5 name5, number5

Actual:
Current Data:
2 (name2, number2)
3 (name3, number3)
4 (name4, number4)
5 (name5, number5)
```

2.3 Module D

2.3.1 updateDataTest()

The updateData function takes in an index, but then it adds one to the index. This results in a test failure when the test expects index 5, for example to updated, the application actually ends up updating the entry at index 6. It is interesting how this unit test fails yet when testing everything at once, the update command in module A works as expected. Curiously, in module A, the index is subtracted by 2, so this is probably how the functionality ends up still working in the end.

```
Expected :
[(testName0, testNumber0),
 (testName1, testNumber1),
 (testName2, testNumber2),
 (testName3, testNumber3),
 (testName4, testNumber4),
 (testName, testNumber),
 (testName6, testNumber6),
 (testName7, testNumber7),
 (testName8, testNumber8),
 (testName9, testNumber9)]
Actual :
```

```
[(testName0,
 testNumber0),
 (testName1, testNumber1),
 (testName2, testNumber2),
 (testName3, testNumber3),
 (testName4, testNumber4),
 (testName5, testNumber5),
 (testName, testNumber),
 (testName7, testNumber7),
 (testName8, testNumber8),
 (testName9, testNumber9)]
```

2.4 Module C

In general, the sorting code appears smelly. It is not using a library function when one is available for ArrayLists, and the increment variable may not be an integer number, even though it is being used to help calculate indicies to manipulate the data. Both of the errors below can be fixed by using the **Collections.sort()** method available in java, since **compareTo()** for the Entry class is already overridden.

2.4.1 sortFourElementsTest()

The application treats the sorting differently for some reason when there are four elements to sort, because it does a check for when the size of the collection divided by 2 is equal to 2. In this case, it looks like with four elements, the application returns an unsorted ArrayList

```
Input:
[(ccc, ccc), (aaa, aaa), (bbb, ddd), (bbb, aaa)]
Expected :
[(aaa, aaa), (bbb, aaa), (bbb, ddd), (ccc, ccc)]
Actual :
[(ccc, ccc), (aaa, aaa), (bbb, ddd), (bbb, aaa)]
```

2.4.2 sortDataTest()

The application seems to mostly sort the data ok, but it forgets about sorting the first element in the data, and leaves it untouched.

```
Input    :[(ddd, aaa), (bbb, bbb), (ccc, ccc), (aaa, aaa), (ccc, aaa), (bbb, aaa)]
Expected :[(aaa, aaa), (bbb, aaa), (bbb, bbb), (ccc, aaa), (ccc, ccc), (ddd, aaa)]
Actual   :[(ddd, aaa), (aaa, aaa), (bbb, bbb), (bbb, aaa), (ccc, aaa), (ccc, ccc)]
```

2.5 Module A

2.5.1 testSort()

This test fails because the file itself is not sorted. Instead, we have a case where Module A is calling Module C to sort the data, but Module A never tells another module to update the file with the sorted data, so the database file is left unmodified and therefore unsorted.

```
Input:
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
Expected :
aaa,aaa
bbb,aaa
bbb,bbb
ccc,aaa,
ccc,ccc
ddd,aaa
Actual :
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
```

2.5.2 testDeleteInvalidArguments()

The application is not equipped to handle arguments after the delete command which are non numeric. In such a case where a letter or word is inputted after the word 'delete', the program will ungracefully panic, instead of handling the error.

3 Task 2

For part two in this lab, the database application was tested using the bottom up incremental testing strategy. Unit tests were made starting at the lower level modules, building our way up to the higher level modules so that nothing was mocked. The actual tests can be found in Appendix B, and are also in the zip archive in which this report was submitted. They can be found in *"Lab 5/Lab 5/src/test/java/bottumup"* The test results are found on the next page.

bottumUp in Lab_5:
39 total, 2 error, 5 failed, 32 passed

504 ms

Collapse | Expand

Test03_CF		28 ms
sortFourElementsTest()	failed	25 ms
setFTest()	passed	1 ms
sortDataTest()	failed	2 ms
Test04_DFG		328 ms
updateDataTest()	failed	320 ms
insertDataTest()	passed	3 ms
deleteDataTest()	passed	3 ms
setFTest()	passed	1 ms
setGTest()	passed	1 ms
Test00_F		2 ms
testModuleF()	failed	2 ms
Test05_E		1 ms
testE()	passed	1 ms
Test01_G		5 ms
testModuleGFail()	passed	2 ms
testModuleG()	passed	3 ms
Test06_Everything		136 ms
testSortNoData()	passed	83 ms
testLoadBReturnsNull()	passed	2 ms
testAdd()	passed	5 ms
testExit()	passed	1 ms
testHelp()	passed	2 ms
testLoad()	passed	2 ms
testSort()	failed	7 ms
testSortCReturnsNull()	passed	2 ms

bottumUp in Lab_5:
39 total, 2 error, 5 failed, 32 passed

504 ms

Collapse | Expand

testAddNoData()	passed	2 ms
testDelete()	passed	6 ms
testDeleteNoArgument()	passed	3 ms
testAddNoArgument()	passed	2 ms
testUpdateNoArgument()	passed	2 ms
testDeleteDReturnsNull()	passed	2 ms
testUpdateCReturnsNull()	passed	1 ms
testUpdate()	passed	4 ms
testDeleteInvalidArguments()	error	1 ms
testUpdateInvalidArguments()	error	2 ms

Test02_BF

4 ms

loadFileTestInValidFile()	passed	1 ms
loadFileTestFileNotFound()	passed	1 ms
loadFileTestValidFile()	passed	1 ms
setFTest()	passed	1 ms

The failing tests are explained as follows: It should be noted that the tests were run in the order F, G, BF, CF, DFG, E, Everything.

3.1 Test CF

In general, the sorting code appears smelly. It is not using a library function when one is available for ArrayLists, and the increment variable may not be an integer number, even though it is being used to help calculate indicies to manipulate the data. Both of the errors below can be fixed by using the **Collections.sort()** method available in java, since **compareTo()** for the Entry class is already overridden.

3.1.1 sortFourElementsTest()

The application treats the sorting differently for some reason when there are four elements to sort, because it does a check for when the size of the collection divided by 2 is equal to 2. In this case, it looks like with four elements, the application returns an unsorted ArrayList

```
Input:
[(ccc, ccc), (aaa, aaa), (bbb, ddd), (bbb, aaa)]
Expected :
[(aaa, aaa), (bbb, aaa), (bbb, ddd), (ccc, ccc)]
Actual :
[(ccc, ccc), (aaa, aaa), (bbb, ddd), (bbb, aaa)]
```

3.1.2 sortDataTest()

The application seems to mostly sort the data ok, but it forgets about sorting the first element in the data, and leaves it untouched.

```
Input    :[(ddd, aaa), (bbb, bbb), (ccc, ccc), (aaa, aaa), (ccc, aaa), (bbb, aaa)]
Expected :[(aaa, aaa), (bbb, aaa), (bbb, bbb), (ccc, aaa), (ccc, ccc), (ddd, aaa)]
Actual   :[(ddd, aaa), (aaa, aaa), (bbb, bbb), (bbb, aaa), (ccc, aaa), (ccc, ccc)]
```

3.2 Test DFG

3.2.1 updateDataTest()

The updateData function takes in an index, but then it adds one to the index. This results in a test failure when the test expects index 5, for example to be updated, the application actually ends up updating the entry at index 6. It is interesting how this unit test fails yet when testing everything at once, the update command in module A works as expected. Curiously, in module A, the index is subtracted by 2, so this is probably how the functionality ends up still working in the end.

```
Expected :
[(testName0, testNumber0),
 (testName1, testNumber1),
 (testName2, testNumber2),
 (testName3, testNumber3),
 (testName4, testNumber4),
 (testName, testNumber),
 (testName6, testNumber6),
 (testName7, testNumber7),
 (testName8, testNumber8),
 (testName9, testNumber9)]
Actual :
[(testName0,
 testNumber0),
 (testName1, testNumber1),
 (testName2, testNumber2),
 (testName3, testNumber3),
 (testName4, testNumber4),
 (testName5, testNumber5),
 (testName, testNumber),
 (testName7, testNumber7),
 (testName8, testNumber8),
 (testName9, testNumber9)]
```

3.3 Test F

3.3.1 testModuleF()

When Module F displays data, it skips the first line, or the first entry. This is because the for loop on line 15 in Module F starts from $i = 1$, instead of $i = 0$, which means it starts from indexing the second element in the ArrayList, instead of the first element.

```
Expected:
Current Data:
1 name1, number1
2 name2, number2
3 name3, number3
4 name4, number4
5 name5, number5

Actual:
Current Data:
2 (name2, number2)
3 (name3, number3)
4 (name4, number4)
5 (name5, number5)
```

3.4 Test Everything

3.4.1 testSort()

This test fails because the file itself is not sorted. Instead, we have a case where Module A is calling Module C to sort the data, but Module A never tells another module to update the file with the sorted data, so the database file is left unmodified and therefore unsorted.

```
Input:
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
Expected :
aaa,aaa
bbb,aaa
bbb,bbb
ccc,aaa,
ccc,ccc
ddd,aaa
Actual :
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
```

3.4.2 testDeleteInvalidArguments()

The application is not equipped to handle arguments after the delete command which are non numeric. In such a case where a letter or word is inputted after the word 'delete', the program will ungracefully panic, instead of handling the error.

3.4.3 testUpdateInvalidArguments()

This test fails for the exact same reason for which testDeleteInvalidArguments() fails.

4 Conclusion

In this lab, we were introduced to integration testing. We tested a simple command-line database application using two integration testing strategies:

non-incremental testing, and the bottom up incremental testing strategy.

In general, integration testing seems beneficial over testing individual modules. It allows us to verify that the system as a whole is working as expected. For example, we noticed that as a whole, functions like updateData worked as expected, even though they failed when testing the modules individually.

This type of testing might be tedious for a large scale system, but it would definitely be effective, since it would ensure that each module is communicating with other modules as expected. If the modules were tested individually for a large scale system without integration, we might find that the system as a whole does not work, even though the unit tests are passing. Furthermore, using integration testing strategies allows for the errors to be found and traced to the modules where the error is happening fairly easily. Imagine trying to debug a large scale system without knowing where the error is originating when two unknown modules communicate with each other.

Drivers and stubs are an effective method of isolating modules. They allow for a module which depends on other modules to be tested independently of the other modules it may depend on or communicate with. In other words, isolating the modules.

I think the bottom up strategy would be best for test-driven development. That way, we can start at the lower level modules which do not depend on anything, and work our way up. As we get to higher level modules, we have continuous integration of the newer code, which means the testing process is less painful. As the icing on top, we get validation that the system is working as intended every step of the way.

However, for testing software libraries, the big bang method might make more sense, since libraries are isolated from the code that a user writes. It would not make sense for the burden of testing to be passed on to the user of a library. The user expects the library to work when they use it, so in this way we end up with something resembling the big bang strategy, where the library and the users' code is tested individually, and then all together at once in the end.

I think that the The big bang strategy would be the easiest testing strategy to maintain over time since if an API changes with any module, we just need to update the tests for one module, and the one integration test that tests everything.

Personally, I like the bottom-up strategy for incremental testing, since no stubs are needed, so we know that the actual code is always being tested. Plus, each test builds off of the previous one, so if we encounter an error during this process, it is very easy to find where it is originating. Compared to big-bang testing, what if there is a situation where each module individually passes, but the one big integration test fails? Where would one even start to look for the error, and determine the offending modules?

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
63      public void load() throws ModuleE.DataBaseExitException {
64          Mockito.when(md.insertData(any(), anyString(),
            ↪  anyString(), anyString())).thenReturn(new
            ↪  ArrayList<Entry>());
65          ma.run(new String[]{"load", TEST_FILENAME});
66      }
67
68      @Test
69      public void testLoad() throws ModuleE.DataBaseExitException {
70          load();
71          verify(mb, times(1)).loadFile(anyString());
72      }
73
74      @Test
75      public void testLoadNoArgument() throws
        ↪  ModuleE.DataBaseExitException {
76              ma.run(new String[]{"load"});
77              assertEquals("Malformed command!\n",
                ↪  stdout.toString());
78              verify(mb, never()).loadFile(anyString());
79      }
80
81      @Test
82      public void testLoadBReturnsNull() throws
        ↪  ModuleE.DataBaseExitException {
83          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
        ↪  ModuleE.DataBaseExitException {
90          ma.run(new String[]{"add", "name", "number"});
91          assertEquals("No file loaded!\n", stdout.toString());
92          verify(md, never()).insertData(any(), anyString(),
            ↪  anyString(), anyString());
93      }
94
95      @Test
96      public void testAdd() throws ModuleE.DataBaseExitException {
97          Mockito.when(md.insertData(any(), anyString(),
            ↪  anyString(), anyString())).thenReturn(new
            ↪  ArrayList<Entry>());
98          ma.run(new String[]{"load", TEST_FILENAME});
99          ma.run(new String[]{"add", "name", "number"});
100         verify(md, times(1)).insertData(any(), anyString(),
            ↪  anyString(), anyString());
101     }
102
103     @Test
104     public void testAddDreturnsNull() throws
        ↪  ModuleE.DataBaseExitException {
105         Mockito.when(md.insertData(any(), anyString(),
            ↪  anyString(), anyString())).thenReturn(null);
106         ma.run(new String[]{"load", TEST_FILENAME});
107         ma.run(new String[]{"add", "name", "number"});
108         verify(md, times(1)).insertData(any(), anyString(),
            ↪  anyString(), anyString());
109     }
110
111     @Test
112     public void testAddNoArgument() throws
        ↪  ModuleE.DataBaseExitException {
113         ma.run(new String[]{"load", TEST_FILENAME});
114         ma.run(new String[]{"add"});
115         assertEquals("Malformed command!\n", stdout.toString());
116         verify(md, never()).insertData(any(), anyString(),
            ↪  anyString(), anyString());
117     }
118
119     @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
        ↪  ModuleE.DataBaseExitException {
128         ma.run(new String[]{"sort"});
129         assertEquals("No file loaded!\n", stdout.toString());
130         verify(mc, never()).sortData(any());
131     }
132 
133     @Test
134     public void testSortCReturnsNull() throws
        ↪  ModuleE.DataBaseExitException {
135         Mockito.when(mc.sortData(any())).thenReturn(null);
136         ma.run(new String[]{"load", TEST_FILENAME});
137         ma.run(new String[]{"sort"});
138         verify(mc, times(1)).sortData(any());
139     }
140 
141     @Test
142     public void testUpdate() throws ModuleE.DataBaseExitException
        ↪  {
143         ma.run(new String[]{"load", TEST_FILENAME});
144         ma.run(new String[]{"update", "1", "2", "3"});
145         verify(md, times(1)).updateData(any(), anyInt(),
            ↪  anyString(), anyString(), anyString());
146     }
147 
148     @Test
149     public void testUpdateInvalidArguments() throws
        ↪  ModuleE.DataBaseExitException {
150         ma.run(new String[]{"load", TEST_FILENAME});
151         ma.run(new String[]{"update", "arg1", "arg2", "arg3"});
152         verify(md, never()).updateData(any(), anyInt(),
            ↪  anyString(), anyString(), anyString());
153     }
154 
155 
156     @Test
157     public void testUpdateNoData() throws
        ↪  ModuleE.DataBaseExitException {
158         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[]{"thiscommanddoesn'texist"});
226     }
227
228
```

```
}
```

A.2 Module B

```
package bigbang;

import TestUtil.TestUtil;
import modules.ModuleB;
import modules.ModuleF;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import data.Entry;
import java.nio.file.Paths;
import java.util.ArrayList;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.mock;

// IO Exception catching is not tested
// Explain in the report that it's just calling a library
↪  function
// Also, the scenario was never encountered e.g. reading
↪  /etc/shadow

public class TestB {

    ModuleB mb;
    ModuleF mf;
    final static String TEST_FILENAME = "BTEST_FILE";
    static File f;

    @BeforeEach
    public void setUp() throws IOException {
        mf = mock(ModuleF.class);
        mb = new ModuleB(mf);

        f = new File(TEST_FILENAME);
        f.createNewFile();
        Files.writeString(Paths.get(TEST_FILENAME), """
This
```

```
is,some
test
data""");
    }

    @AfterEach
    public void tearDown(){
        f.delete();
    }

    @Test
    public void loadFileTestValidFile() {
       ArrayList<Entry> ret =  mb.loadFile(TEST_FILENAME);

       ArrayList<Entry> expected = new ArrayList<>() {{
           add(new Entry("is", "some"));
       }};

       TestUtil.compareArrayOfEntries(expected, ret);
    }

    @Test
    public void loadFileTestInValidFile()  {
       mb.loadFile("/");
    }

    @Test
    public void loadFileTestFileNotFound(){
        mb.loadFile("");
    }

    @Test
    public void setFTest(){
        ModuleF newF = mock(ModuleF.class);
        mb.setF(newF);
    }
}
```

A.3 Module C

```
package bigbang;

import TestUtil.TestUtil;
import data.Entry;
import modules.ModuleC;
import modules.ModuleF;
```

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;

import static org.mockito.Mockito.mock;

public class TestC {

    ModuleF mf;
    ModuleC mc;

    @BeforeEach
    public void setUp(){
        mf = mock(ModuleF.class);
        mc = new ModuleC(mf);
    }

    @Test
    public void sortDataTest(){

        final String TEST_NAME = "testName";
        final String TEST_NUMBER = "testNumber";

        ArrayList<Entry> unsorted = new ArrayList<>() {{
            add(new Entry("ddd", "aaa"));
            add(new Entry("bbb", "bbb"));
            add(new Entry("ccc", "ccc"));
            add(new Entry("aaa", "aaa"));
            add(new Entry("ccc", "aaa"));
            add(new Entry("bbb", "aaa"));
        }};

        ArrayList<Entry> sorted = new ArrayList<>() {{
            add(new Entry("aaa", "aaa"));
            add(new Entry("bbb", "aaa"));
            add(new Entry("bbb", "bbb"));
            add(new Entry("ccc", "aaa"));
            add(new Entry("ccc", "ccc"));
            add(new Entry("ddd", "aaa"));
        }};
        ArrayList<Entry> ret = mc.sortData(unsorted);

        TestUtil.compareArrayOfEntries(sorted, ret);
    }

```

```
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
```

```java
public class TestD {

    ModuleF mf;
    ModuleG mg;
    ModuleD md;

    final static String TEST_NAME = "testName";
    final static String TEST_NUMBER = "testNumber";
    final static String TEST_FILENAME = "testFilename";
    final static int TEST_INDEX = 5;

    ArrayList<Entry> expected;

    @BeforeEach
    public void setUp(){
        mf = mock(ModuleF.class);
        mg = mock(ModuleG.class);

        md = new ModuleD(mf, mg);

        expected = new ArrayList<>() {{
            for (int i = 0; i < 10; i += 1)
                add(new Entry(TEST_NAME + i, TEST_NUMBER + i));
        }};
    }

    @AfterEach
    public void after(TestInfo testInfo){
        if(testInfo.getTags().contains("SkipAfter")) {
            return;
        }
        verify(mf, times(1)).displayData(any());
        verify(mg, times(1)).updateData(anyString(), any());
    }

    @Test
    public void insertDataTest(){
        ArrayList<Entry> ret=
        ↪  md.insertData((ArrayList<Entry>)expected.clone(),
        ↪  TEST_NAME, TEST_NUMBER, TEST_FILENAME);

        expected.add(new Entry(TEST_NAME, TEST_NUMBER));

        TestUtil.compareArrayOfEntries(expected, ret);
    }

```

```
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          TestUtil.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          TestUtil.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
```

```
public class TestE {

    @Test
    public void testE(){
        assertThrows(ModuleE.DataBaseExitException.class , ()->
        ↪  new ModuleE().exitProgram());
    }

}
```

A.6 Module F

```
package bigbang;

import data.Entry;
import modules.ModuleF;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.util.ArrayList;

public class TestF {


    @Test
    public void testModuleF(){
        ByteArrayOutputStream stdout = new
        ↪  ByteArrayOutputStream();

        ModuleF mf = new ModuleF();
        mf.setOutputStream(new PrintStream(stdout));

        ArrayList<Entry> entries = new ArrayList<>();
        entries.add(new Entry("name1", "number1"));
        entries.add(new Entry("name2", "number2"));
        entries.add(new Entry("name3", "number3"));
        entries.add(new Entry("name4", "number4"));
        entries.add(new Entry("name5", "number5"));

        mf.displayData(entries);

        assertEquals("""
Current Data:
1 name1, number1
```

```
2 name2, number2
3 name3, number3
4 name4, number4
5 name5, number5
""", stdout.toString());
    }
}
```

A.7 Module G

```
package bigbang;

import data.Entry;
import modules.ModuleG;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public class TestG {

    static String FILENAME = "GTEST_FILE";
    static File f;
    static ModuleG mg;

    @BeforeEach
    public void createFile(){
        f = new File(FILENAME);
        mg = new ModuleG();
    }

    @AfterEach
    public void deleteFile(){
        f.delete();
    }

    @Test
    public void testModuleG() throws IOException {

        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
         ↪  System.out.println(Files.readString(Paths.get(TEST_FILENAME)));
59         f.delete();
60     }
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
83      public void load() throws ModuleE.DataBaseExitException {
84  //          Mockito.when(md.insertData(any(), anyString(),
    ↪  anyString(), anyString())).thenReturn(new
    ↪  ArrayList<Entry>());
85          ma.run(new String[]{"load", TEST_FILENAME});
86      }
87
88      @Test
89      public void testLoad() throws ModuleE.DataBaseExitException {
90          load();
91          verify(mb, times(1)).loadFile(anyString());
92      }
93
94      @Test
95      public void testLoadNoArgument() throws
        ↪  ModuleE.DataBaseExitException {
96          ma.run(new String[]{"load"});
97          assertEquals("Malformed command!\n", stdout.toString());
98          verify(mb, never()).loadFile(anyString());
99      }
100
101     @Test
102     public void testLoadBReturnsNull() throws
        ↪  ModuleE.DataBaseExitException {
```

```
103 //
    ↪ Mockito.when(mb.loadFile(anyString())).thenReturn(null);
104         ma.run(new String[]{"load", NONEXISTENT_FILE});
105         verify(mb, times(1)).loadFile(anyString());
106     }
107
108     @Test
109     public void testAddNoData() throws
        ↪ ModuleE.DataBaseExitException {
110         ma.run(new String[]{"add", "name", "number"});
111         assertEquals("No file loaded!\n", stdout.toString());
112         verify(md, never()).insertData(any(), anyString(),
            ↪ anyString(), anyString());
113     }
114
115     @Test
116     public void testAdd() throws ModuleE.DataBaseExitException,
        ↪ IOException {
117 //        Mockito.when(md.insertData(any(), anyString(),
    ↪ anyString(), anyString())).thenReturn(new
    ↪ ArrayList<Entry>());
118         ma.run(new String[]{"load", TEST_FILENAME});
119         ma.run(new String[]{"add", "name", "number"});
120         verify(md, times(1)).insertData(any(), anyString(),
            ↪ anyString(), anyString());
121
122         assertEquals("name,number\n",
            ↪ Files.readString(Paths.get(TEST_FILENAME)));
123     }
124
125     @Test
126     public void testAddDreturnsNull() throws
        ↪ ModuleE.DataBaseExitException {
127
128         // through static analysis, we can see that d.insertdata
            ↪ will never return null in practice
129
130 //        Mockito.when(md.insertData(any(), anyString(),
    ↪ anyString(), anyString())).thenReturn(null);
131 //        ma.run(new String[]{"load", NONEXISTENT_FILE});
132 //        ma.run(new String[]{"add", "name", "number"});
133 //        verify(md, times(1)).insertData(any(), anyString(),
    ↪ anyString(), anyString());
134     }
135
136     @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
                ↪  is null
178             // but the program does a null check on line 56 in module
                ↪  A
179             // so it will never return null
180
181     //        Mockito.when(mc.sortData(any())).thenReturn(null);
182     //        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 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();
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[]{"thiscommanddoesn'texist"});
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
```

```java
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.util.ArrayList;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class Test00_F {


    @Test
    public void testModuleF(){
        ByteArrayOutputStream stdout = new
        ↪  ByteArrayOutputStream();

        ModuleF mf = new ModuleF();
        mf.setOutputStream(new PrintStream(stdout));

        ArrayList<Entry> entries = new ArrayList<>();
        entries.add(new Entry("name1", "number1"));
        entries.add(new Entry("name2", "number2"));
        entries.add(new Entry("name3", "number3"));
        entries.add(new Entry("name4", "number4"));
        entries.add(new Entry("name5", "number5"));

        mf.displayData(entries);

        assertEquals("""
Current Data:
1 name1, number1
2 name2, number2
3 name3, number3
4 name4, number4
5 name5, number5
""", stdout.toString());
    }
}
```

B.2 Test G

```java
package bottumUp;

import data.Entry;
import modules.ModuleG;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
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
```

```
name5,number5
""", Files.readString(Paths.get(FILENAME)));
    }

    @Test
    public void testModuleGFail() {
        ByteArrayOutputStream stdout= new
        ↪  ByteArrayOutputStream();
        System.setOut(new PrintStream(stdout));
        mg.updateData("", new ArrayList<Entry>());

        assertEquals("Error updating DB File.\n",
        ↪  stdout.toString());
    }
}
```

B.3 Test BF

```
package bottumUp;

import TestUtil.TestUtil;
import modules.ModuleB;
import modules.ModuleF;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import data.Entry;
import java.nio.file.Paths;
import java.util.ArrayList;

import static org.mockito.Mockito.mock;

// IO Exception catching is not tested
// Explain in the report that it's just calling a library
↪  function
// Also, the scenario was never encountered e.g. reading
↪  /etc/shadow

public class Test02_BF {

    ModuleB mb;
    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"));
```

```
            add(new Entry("bbb", "bbb"));
            add(new Entry("ccc", "aaa"));
            add(new Entry("ccc", "ccc"));
            add(new Entry("ddd", "aaa"));
        }};
        ArrayList<Entry> ret = mc.sortData(unsorted);

        TestUtil.compareArrayOfEntries(sorted, ret);
    }

    @Test
    public void setFTest(){
        ModuleF newF = new ModuleF();
        mc.setF(newF);
    }

    // to cover line 28 in ModuleC
    @Test
    public void sortFourElementsTest(){
        ArrayList<Entry> unsorted = new ArrayList<>() {{
            add(new Entry("ccc", "ccc"));
            add(new Entry("aaa", "aaa"));
            add(new Entry("bbb", "ddd"));
            add(new Entry("bbb", "aaa"));
        }};

        ArrayList<Entry> sorted = new ArrayList<>() {{
            add(new Entry("aaa", "aaa"));
            add(new Entry("bbb", "aaa"));
            add(new Entry("bbb", "ddd"));
            add(new Entry("ccc", "ccc"));
        }};

        ArrayList<Entry> ret = mc.sortData(unsorted);

        TestUtil.compareArrayOfEntries(sorted, ret);
    }

}
```

B.5 Test DFG

```
package bottumUp;

import TestUtil.TestUtil;
import data.Entry;
```

```java
import modules.ModuleD;
import modules.ModuleF;
import modules.ModuleG;
import org.junit.jupiter.api.*;

import java.util.ArrayList;

import static org.mockito.Mockito.*;


public class Test04_DFG {

    ModuleF mf;
    ModuleG mg;
    ModuleD md;

    final static String TEST_NAME = "testName";
    final static String TEST_NUMBER = "testNumber";
    final static String TEST_FILENAME = "testFilename";
    final static int TEST_INDEX = 5;

    ArrayList<Entry> expected;

    @BeforeEach
    public void setUp(){
        mf = spy(new ModuleF());
        mg = spy(new ModuleG());

        md = new ModuleD(mf, mg);

        expected = new ArrayList<>() {{
            for (int i = 0; i < 10; i += 1)
                add(new Entry(TEST_NAME + i, TEST_NUMBER + i));
        }};
    }

    @AfterEach
    public void after(TestInfo testInfo){
        if(testInfo.getTags().contains("SkipAfter")) {
            return;
        }
        verify(mf, times(1)).displayData(any());
        verify(mg, times(1)).updateData(anyString(), any());
    }

    @Test
```

```
51      public void insertDataTest(){
52          ArrayList<Entry> ret=
            ↪  md.insertData((ArrayList<Entry>)expected.clone(),
            ↪  TEST_NAME, TEST_NUMBER, TEST_FILENAME);
53
54          expected.add(new Entry(TEST_NAME, TEST_NUMBER));
55
56          TestUtil.compareArrayOfEntries(expected, ret);
57      }
58
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          TestUtil.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          TestUtil.compareArrayOfEntries(expected, ret);
75      }
76
77      @Tag("SkipAfter")
78      @Test
79      public void setFTest(){
80          ModuleF newF = new ModuleF();
81          md.setF(newF);
82      }
83
84      @Tag("SkipAfter")
85      @Test
86      public void setGTest(){
87          ModuleG newG = new ModuleG();
88          md.setG(newG);
89      }
90
```

```
}
```

B.6 Test E

```
package bottumUp;

import modules.ModuleE;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertThrows;

public class Test05_E {

    @Test
    public void testE(){
        assertThrows(ModuleE.DataBaseExitException.class , ()->
        ↪  new ModuleE().exitProgram());
    }

}
```

B.7 Test Everything

```
package bottumUp;

import modules.*;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.nio.file.Files;
import java.nio.file.Paths;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.*;


public class Test06_Everything {

```

```
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
            ↪  System.out.println(Files.readString(Paths.get(TEST_FILENAME)));
59          f.delete();
60      }
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
83      public void load() throws ModuleE.DataBaseExitException {
84  //          Mockito.when(md.insertData(any(), anyString(),
    ↪  anyString(), anyString())).thenReturn(new
    ↪  ArrayList<Entry>());
85          ma.run(new String[]{"load", TEST_FILENAME});
86      }
87
88      @Test
89      public void testLoad() throws ModuleE.DataBaseExitException {
90          load();
91          verify(mb, times(1)).loadFile(anyString());
92      }
93
94      @Test
95      public void testLoadNoArgument() throws
        ↪  ModuleE.DataBaseExitException {
96          ma.run(new String[]{"load"});
97          assertEquals("Malformed command!\n", stdout.toString());
98          verify(mb, never()).loadFile(anyString());
99      }
100
101     @Test
102     public void testLoadBReturnsNull() throws
        ↪  ModuleE.DataBaseExitException {
103 //
    ↪  Mockito.when(mb.loadFile(anyString())).thenReturn(null);
104         ma.run(new String[]{"load", NONEXISTENT_FILE});
105         verify(mb, times(1)).loadFile(anyString());
106     }
107
108     @Test
```

```
109     public void testAddNoData() throws
        ↪  ModuleE.DataBaseExitException {
110         ma.run(new String[]{"add", "name", "number"});
111         assertEquals("No file loaded!\n", stdout.toString());
112         verify(md, never()).insertData(any(), anyString(),
            ↪  anyString(), anyString());
113     }
114
115     @Test
116     public void testAdd() throws ModuleE.DataBaseExitException,
        ↪  IOException {
117 //        Mockito.when(md.insertData(any(), anyString(),
    ↪  anyString(), anyString())).thenReturn(new
    ↪  ArrayList<Entry>());
118         ma.run(new String[]{"load", TEST_FILENAME});
119         ma.run(new String[]{"add", "name", "number"});
120         verify(md, times(1)).insertData(any(), anyString(),
            ↪  anyString(), anyString());
121
122         assertEquals("name,number\n",
            ↪  Files.readString(Paths.get(TEST_FILENAME)));
123     }
124
125     @Test
126     public void testAddDreturnsNull() throws
        ↪  ModuleE.DataBaseExitException {
127
128         // through static analysis, we can see that d.insertdata
            ↪  will never return null in practice
129
130 //        Mockito.when(md.insertData(any(), anyString(),
    ↪  anyString(), anyString())).thenReturn(null);
131 //        ma.run(new String[]{"load", NONEXISTENT_FILE});
132 //        ma.run(new String[]{"add", "name", "number"});
133 //        verify(md, times(1)).insertData(any(), anyString(),
    ↪  anyString(), anyString());
134     }
135
136     @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
            ↪  is null
178         // but the program does a null check on line 56 in module
            ↪  A
179         // so it will never return null
180
181 //        Mockito.when(mc.sortData(any())).thenReturn(null);
182 //        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[]{"thiscommanddoesn'texist"});
313     }
314 }
```