


ECE 322 Lab Report #4

 Benjamin Kong
1573684

November 24, 2021

Introduction

The purpose of this lab was to introduce integration white box testing techniques. We will use Python and Python unittest `unittest.mock` to gain experience with integration testing.

Integration testing is the logical extension of unit testing. There are two common approaches:

- Non-incremental testing (big bang): where each module is tested independently, then the system is tested as a whole, and
- Incremental testing: where the set of previously tested modules are combined with the next module to be tested before running tests. There are two common methods of incremental testing:
 - Bottom up: where the lowest level modules are tested in isolation, then higher level modules are added incrementally, and
 - Top down: where the highest level modules are tested in isolation (stubbing lower level modules), then lower level modules are added incrementally.

In this lab, we will be creating mock objects which are essentially used as stubs in our integration tests. Usually, when integration testing is performed, we need to use stubs and drivers.

- Stubs are used as a stand in for lower level modules that aren't currently being tested. Stubs return dummy values or makes an assertion so that the higher level modules can still run and be tested.
- Drivers are a piece of testing code that make it possible to call the submodule of an application by itself. Often, driver code requires stub setup and object initialization.

We will be testing a simple command line interface (CLI) database system that has seven modules, Module A to module F. We will be focusing on non-incremental testing (big bang integration). We will prepare a series of unit tests for each module and attempt to achieve full statement coverage using our tests, then attempt to test the entire program when it is assembled. We will also compile test results, explain failed test cases, and discuss possible fixes to make test cases pass.

Task

For this lab, we tested the database application using the big bang testing technique. We created unit tests for each module A-F along with test cases for the entire program when it is assembled. For individual module tests, we mocked any dependencies on other modules. The code for the tests have been included with the submission under `Lab4_src/tests`. The table of tests and results is displayed below with failed test cases highlighted in red.

Table 1: Test results resulting from the big bang testing technique.

Test ID	Module	Test Name	Pass
1	All	test_display_help	Yes
2	All	test_empty_command	Yes
3	All	test_invalid_command	Yes
4	All	test_help	Yes
5	All	test_exit	Yes
6	All	test_load	Yes
7	All	test_add	Yes
8	All	test_update	Yes
9	All	test_delete	Yes
10	All	test_sort	Yes
11	All	test_commands_index_error	Yes
12	All	test_commands_no_file	Yes
13	A	test_commands	Yes
14	A	test_commands_index_error	Yes
15	A	test_commands_no_file	Yes
16	A	test_display_help	Yes
17	A	test_empty_command	Yes
18	A	test_get_data	Yes
19	A	test_invalid_command	Yes
20	A	test_parse_add	Yes
21	A	test_parse_add_no_data	Yes
22	A	test_parse_delete	Yes
23	A	test_parse_delete_no_value	Yes
24	A	test_parse_load	Yes
25	A	test_parse_load_no_data	Yes
26	A	test_parse_update	Yes
27	A	test_parse_update_no_data	Yes
28	A	test_run_exit	Yes
29	A	test_run_sort	Yes
30	A	test_run_sort_no_data	Yes

31	A	test_set_data	Yes
32	B	test_get_f	Yes
33	B	test_set_f	Yes
34	B	test_file_not_found_error	No
35	B	test_io_error	Yes
36	B	test_load_file	Yes
37	C	test_get_f	Yes
38	C	test_set_f	Yes
39	C	test_sort_data	Yes
40	D	test_getters	Yes
41	D	test_setters	Yes
42	D	test_insert_data	Yes
43	D	test_update_data	No
44	D	test_delete_data	No
45	E	test_run_exit	Yes
46	F	test_display_data	Yes
47	G	test_data_update	Yes
48	G	test_data_update_file_error	Yes

An image of the coverage report is provided below.

Figure 1: Coverage report resulting from the big bang testing technique.

Coverage report: 97%				
Module	statements	missing	excluded	coverage
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\data\Entry.py	20	2	0	90%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\data__init__.py	0	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleA.py	93	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleB.py	30	3	0	90%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleC.py	15	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleD.py	35	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleE.py	5	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleF.py	7	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules\ModuleG.py	9	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\modules__init__.py	0	0	0	100%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleA.py	144	1	0	99%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleB.py	42	1	0	98%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleC.py	26	1	0	96%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleD.py	51	3	0	94%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleE.py	11	1	0	91%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleF.py	14	1	0	93%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestModuleG.py	27	1	0	96%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab4\Lab4_src\tests\TestRunner.py	19	0	0	100%
Total	548	14	0	97%
coverage.py v6.1.2, created at 2021-11-23 23:51 -0700				

As seen from the previous table and image, we have 48 total tests. Of the 48 tests, we had 3 tests that failed. We also had close to 100% code coverage for the actual code. The reason we could not achieve 100% code coverage for `ModuleB.py` is due to the fact that the error is impossible to trigger: the exception that triggers that statement is caught by an earlier statement. As a result, it is impossible to cover that line of code as it is unreachable. Next, we will explain the failed test cases.

Test 34

In this test, we attempted to trigger a `FileNotFoundError` by having our program attempt to use a file that doesn't exist in the directory. However, as mentioned previously, the line of code that handles this specific exception is not reachable as the error is caught by an earlier exception that is a superclass of the `FileNotFoundError` error. In order to fix this error, we could move this statement above the other statement like this:

```
1  except FileNotFoundError:
2      msg = "FileNotFoundError"
3      print(msg)
4  except IOError as e:
5      print("Could not read file:{0.filename}".format(e))
```

Test 43

In this test, we attempted to test updating a data entry. Due to the ambiguous nature of this function, the tester must make a judgement as to what it is supposed to do due to the lack of comments. Is it supposed to remove based on the printed list's index, or the index of an array where the first element is element 0? For my test, I assumed the former (i.e., attempting to remove index 2 would remove the 2nd item in the list). However, the test case fails in this case: attempting to remove the 2nd item ends up removing the 4th item (or the item at the 3rd index). In any case, this behavior is probably incorrect. A fix would be to change the indexing in the `updateData` function in `ModuleD.py` as follows:

```
1  data[index - 1] = Entry(name, number)
2  self.f.displayData(data)
3  self.g.updateData(filename, data);
4  return data
```

Test 44

In this test, we attempted to test removing a data entry. Similarly to test 43, due to the lack of comments, we are left guessing as to what the correct behavior of this function is. Again, we will assume we are supposed to remove based on the printed list's index. This test case fails, however, as attempting to remove the i th element actually removes the $i + 1$ th element. A fix would be to change the indexing in the `deleteData` function in `ModuleD.py` as follows:

```
1  del data[index - 1]
2  self.f.displayData(data)
3  self.g.updateData(filename, data)
4  return data
```

Discussion

As we saw, we were able to effectively uncover some errors in our program by using non-incremental (big bang) testing techniques. This type of testing can be somewhat effective for large projects, as it not only ensures individual modules are working but also makes sure that all modules are working harmoniously together. After all, it is possible for all individual modules of a system to be working while the program as a whole doesn't work at all. However, it may be impossible in practice for large projects due to the amount of tedious work required to get it set up.

We also saw that creating stubs was an effective way to test individual modules. We were able to mock the functionality of other modules so that we can test each module individually.

Conclusion

The purpose of this lab was to introduce integration white box testing techniques. We used Python and Python unittest `unittest.mock` to gain experience with integration testing.

Integration testing is the logical extension of unit testing. In this lab, we focused on non-incremental testing (big bang). In this type of testing, each module is tested independently,

then the system is tested as a whole. Since modules usually depend on other modules, we use stubs to mock objects such that each module can be tested individually. Stubs are used as a stand in for lower level modules that aren't currently being tested. Stubs return dummy values or makes an assertion so that the higher level modules can still run and be tested.

In the lab, we tested a simple database system that had seven modules, Module A to module F. We focused on non-incremental testing (big bang integration). We first created unit tests for Modules A through F, then created unit tests to test the complete program. We were able to effectively uncover some errors in the program through testing: there were some errors when indexing the database for updates and deletion and there were areas of code that were unreachable. Through completing this lab, we saw that non-incremental (big bang) testing is an effective way to test a program, but comes at the cost of requiring a lot of tedious work to complete.