ECE 322 Lab Report #5

🅱️enjamin Kong
1573684

December 8, 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 incremental testing (bottom up and top down). We will prepare a test suite for each incremental testing method and attempt to achieve full statement coverage. We will then compile test results, explain failed test cases, and discuss possible fixes to make test cases pass.

Part 1

Q1 Problem definition

The program to be tested today is a simple CLI database system that has seven modules, Module A to module F. The program supports basic functionality such as insertion, deletion, and modification. The program outputs the current state of the data each time an action is performed. The modules perform the following functions:

- Module A: Reads commands and runs the appropriate module to handle that command
- Module B: Opens a data file
- Module C: Sorts records
- Module D: Modify a record
- Module E: Exit
- Module F: Display the database on the CLI
- Module G: Update the file

We will test the program to find flaws in the program. We will explain failed test cases and then discuss possible fixes. We will test the database application using the bottom up and top down testing techniques.

Q2 Summary of each testing strategy

Top down

We will discuss our use of top down testing. The code for the tests have been included with the submission under `Lab5_src/top_down_tests`.

1. First, we created unit tests for the module on the top layer (module A). We tested the functionality with stubs for lower level dependencies.

2. Then, we tested the middle layer (Modules A, B, C, D, and E). We used stubs for the lowest level dependencies.
3. Lastly, we tested the lowest layer to complete the testing (i.e. combining all the modules together and testing them).

Bottom up

We will discuss our use of bottom up testing. The code for the tests have been included with the submission under `Lab5_src/bottom_up_tests`.

1. First, we created unit tests for each module on the bottom layer (modules F and G).
2. We then created unit tests for each module on the middle layer (modules B, C, D, and E). If modules depended on lower modules, they used the real module, not mocks.
3. Then, lastly, we tested the top layer (module A), again using real modules for the submodules.

Q3 Test case tables

The test case tables are displayed below.

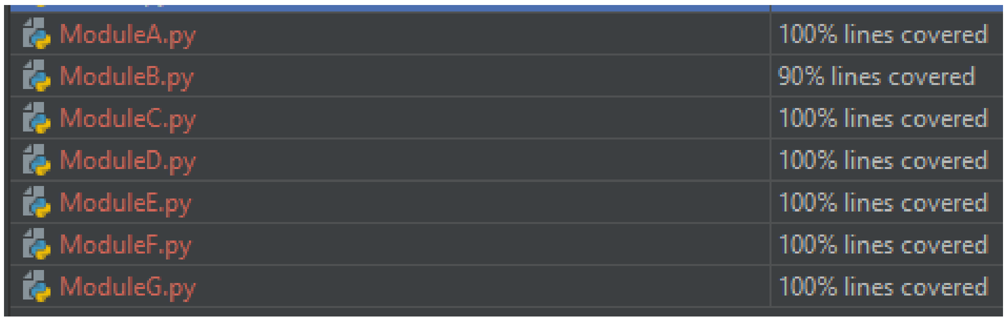
Top down

Table 1: Test results resulting from the top down testing technique.

Test ID	Module	Test Name	Pass
1	TestModuleA	test_commands	Yes
2	TestModuleA	test_commands_index_error	Yes
3	TestModuleA	test_commands_no_file	Yes
4	TestModuleA	test_display_help	Yes
5	TestModuleA	test_empty_command	Yes
6	TestModuleA	test_get_data	Yes
7	TestModuleA	test_invalid_command	Yes
8	TestModuleA	test_parse_add	Yes

9	TestModuleA	test_parse_add_no_data	Yes
10	TestModuleA	test_parse_delete	Yes
11	TestModuleA	test_parse_delete_no_value	Yes
12	TestModuleA	test_parse_load	Yes
13	TestModuleA	test_parse_load_no_data	Yes
14	TestModuleA	test_parse_update	Yes
15	TestModuleA	test_parse_update_no_data	Yes
16	TestModuleA	test_run_exit	Yes
17	TestModuleA	test_run_sort	Yes
18	TestModuleA	test_run_sort_no_data	Yes
19	TestModuleA	test_set_data	Yes
20	TestModuleABCDE	test_delete_data	No
21	TestModuleABCDE	test_file_not_found_error	No
22	TestModuleABCDE	test_getters	Yes
23	TestModuleABCDE	test_insert_data	Yes
24	TestModuleABCDE	test_io_error	Yes
25	TestModuleABCDE	test_load_file	Yes
26	TestModuleABCDE	test_run_exit	Yes
27	TestModuleABCDE	test_setters	Yes
28	TestModuleABCDE	test_sort_data	Yes
29	TestModuleABCDE	test_update_data	No
30	TestModuleABCDEFG	test_add	Yes
31	TestModuleABCDEFG	test_commands_index_error	Yes
32	TestModuleABCDEFG	test_commands_no_file	Yes
33	TestModuleABCDEFG	test_data_update	Yes
34	TestModuleABCDEFG	test_data_update_file_error	Yes
35	TestModuleABCDEFG	test_delete	Yes
36	TestModuleABCDEFG	test_display_data	Yes
37	TestModuleABCDEFG	test_display_help	Yes
38	TestModuleABCDEFG	test_empty_command	Yes
39	TestModuleABCDEFG	test_exit	Yes
40	TestModuleABCDEFG	test_help	Yes
41	TestModuleABCDEFG	test_invalid_command	Yes
42	TestModuleABCDEFG	test_load	Yes
43	TestModuleABCDEFG	test_sort	Yes
44	TestModuleABCDEFG	test_update	Yes

An image of the coverage report is provided below.



ModuleA.py	100% lines covered
ModuleB.py	90% lines covered
ModuleC.py	100% lines covered
ModuleD.py	100% lines covered
ModuleE.py	100% lines covered
ModuleF.py	100% lines covered
ModuleG.py	100% lines covered

Figure 1: Coverage report from top down testing.

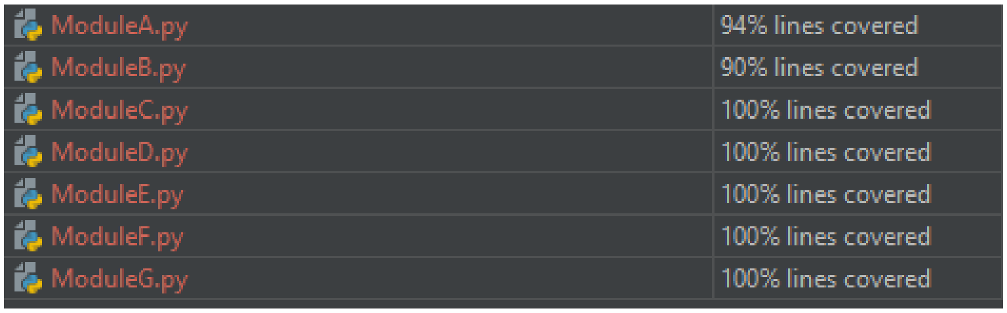
Bottom up

Table 2: Test results resulting from the bottom up testing technique.

Test ID	Module	Test Name	Pass
1	TestModuleF	test_display_data	Yes
2	TestModuleG	test_data_update	Yes
3	TestModuleG	test_data_update_file_error	Yes
4	TestModuleE	test_run_exit	Yes
5	TestModuleDFG	test_delete_data	No
6	TestModuleDFG	test_getters	Yes
7	TestModuleDFG	test_insert_data	Yes
8	TestModuleDFG	test_setters	Yes
9	TestModuleDFG	test_update_data	No
10	TestModuleCF	test_get_f	Yes
11	TestModuleCF	test_set_f	Yes
12	TestModuleCF	test_sort_data	Yes
13	TestModuleBF	test_file_not_found_error	No
14	TestModuleBF	test_get_f	Yes
15	TestModuleBF	test_io_error	Yes
16	TestModuleBF	test_load_file	Yes
17	TestModuleBF	test_set_f	Yes
18	TestModuleA	test_add	Yes
19	TestModuleA	test_commands_index_error	Yes
20	TestModuleA	test_commands_no_file	Yes
21	TestModuleA	test_delete	Yes
22	TestModuleA	test_display_help	Yes
23	TestModuleA	test_empty_command	Yes
24	TestModuleA	test_exit	Yes
25	TestModuleA	test_get_data	Yes

26	TestModuleA	test_help	Yes
27	TestModuleA	test_invalid_command	Yes
28	TestModuleA	test_load	Yes
29	TestModuleA	test_set_data	Yes
30	TestModuleA	test_sort	Yes
31	TestModuleA	test_update	Yes

An image of the coverage report is provided below.



ModuleA.py	94% lines covered
ModuleB.py	90% lines covered
ModuleC.py	100% lines covered
ModuleD.py	100% lines covered
ModuleE.py	100% lines covered
ModuleF.py	100% lines covered
ModuleG.py	100% lines covered

Figure 2: Coverage report from bottom up testing.

Failed test case discussion

- Test "test_delete_data" (test 20 from top down, test 5 from bottom up): 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:

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

```

- Test "test_update_data" (test 29 from top down, test 9 from bottom up): 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:

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

```

- Test "test_file_not_found_error" (test 21 from top down, test 13 from bottom up): 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:

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

```

Q4 Discussion

1. As we saw from our testing, module isolation is effective at detecting hidden errors that may go unseen just from testing the entire system. For example, while updating and deleting from the database appear to work when the whole program is tested, the individual modules have errors. By coincidence, these errors cancel each other out when the whole program is tested, but it isn't always the case that errors will cancel each other out.

2. Top down testing is great when the most important code is in higher level modules. However, bottom up testing is more effective when the important code is in the lower

level modules. Bottom up testing is also better for object oriented systems whereas top down testing is better for procedural oriented systems. Bottom up testing is also better if the system is to be built from the lower levels and up.

3. While integration testing is effective at testing the system as a whole, it does not scale well. As more modules are added, both bottom up and top down testing techniques require more and more effort. Both techniques will require more set up and tests. Bottom up will require more drivers. Top down will require the creation of more stubs.

4. Stubs and drivers are effective ways of isolating the functionality of modules during testing. They allow for easy testing of modules that depend on other modules and are essential for any type of integration testing.

5. In the modern world, systems are typically designed around the object oriented programming paradigm. As such, I believe bottom up integration testing is the most effective. As discussed earlier, bottom up testing is effective for object oriented systems as it allows easier testing of the low level modules (which are the modules that contain the important code in object oriented systems).

6. I believe the bottom up testing technique would be the best for test driven development. This allows the lower level modules (which don't depend on anything) to be implemented first, then allows us to work up after the lower level modules are completed. This allows the verification of modules as we continue development.

7. For testing software libraries, I believe the big bang integration testing technique would be the best choice. Typically, users are not be expected to test libraries; rather, the libraries should be tested by the developers of the library and then the user's system should be tested using the library. As such, it would make more sense to test the library and the user's code separately, then test the user's code combined with the library.

8. I believe the bottom up technique would be easiest to maintain. While all techniques would require substantial effort to maintain, bottom up allows for a gradual build up of tests as low level modules don't depend on other modules. Bottom up also doesn't require the creation of stubs, which are typically harder to create than drivers.

9. As we saw, we were able to effectively uncover some errors in our program by using incremental 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.

Conclusion

The purpose of this lab was to introduce integration white box testing techniques, namely, bottom up and top down testing techniques. We used Python and Python unittest to gain experience with integration testing.

Integration testing is the logical extension of unit testing. In this lab, we focused on incremental testing (top down and bottom up). Incremental testing is where the set of previously tested modules are combined with the next module to be tested before running tests. There are two common types 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.

For bottom up testing, lower level modules typically don't function by themselves. As a result, we need driver code to test these lower level modules. For top down testing, in order to just test a higher level module that may depend on lower level modules, we use stubs. Stubs mock the functionality of lower level modules so that the higher level module can be tested in isolation. 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.

- For bottom up, we created drivers for the lowest level modules. We then created drivers for the middle level modules, then finally, a driver for the top level module. These drivers ran tests. We found three failed test cases.

- For top down, we started at the top level module and stubbed the behavior of the middle level modules. We then tested the middle level modules where the lower level modules were stubbed. Lastly, we tested the system as a whole with nothing stubbed. We also found three failed test cases.

Through both types of incremental testing (top down and bottom up), we were able to discover three errors: there were some errors when indexing the database for updates and deletion and there were areas of code that were unreachable. We investigated these failed test cases and proposed fixes to fix these test cases. We saw that incremental testing is an effective way to test a program, but comes at the cost of requiring a lot of tedious work to complete.