

1 Introduction

The purpose of this lab is to become familiar with integration white-box testing. The goal of this lab was to give us experience with using Python and Python unittest unittest.mock for integration testing.

Integration testing serves as a logical extensions of unit testing. There are two general approaches to integration testing – non-incremental testing (Big Bang) and incremental testing (Top down/Bottom up). In non-incremental testing, each module is tested individually and then the whole system is tested as a whole. In integration testing, we combine the next module to be tested with the set of previously tested modules before running tests. This can generally be done in either a bottom up or top down method. A bottom up method involves testing the lowest level modules in isolation and then incrementally adding higher and higher module levels. A top down method involves testing the highest level modules in isolation and then incrementally adding lower modules.

Integration testing usually involves using various stubs and drivers. Stubs, in integration testing, is used as a stand in for lower level modules that are not currently under test. A stub returns a dummy value or makes an assertion so that the test case can ensure it was called. Drivers are a piece of testing code which makes it possible to call a submodule of an application independently.

In this lab, we will be focusing on non-incremental testing

2 Non-incremental testing - Big Bang testing






During the non-incremental testing portion of this lab, we made unit tests for module A-F along with test cases for the system as a whole. For each module, we mocked any dependencies on other modules. You can find the tests in the directory src/tests. In the test case table on the next page, the red highlighted entries are failed test cases.

Table 1: Test results for non-incremental testing

Test #	Module	Test name	Passed?
1	A	test_allCommands	Pass
2	A	test_allCommandsIndexError	Pass
3	A	test_allCommandsNoFile	Pass
4	A	test_dataGetter	Pass
5	A	test_dataSetter	Pass
6	A	test_help	Pass
7	A	test_noCommand	Pass
8	A	test_parseAddWithData	Pass
9	A	test_parseAddWithNoData	Pass
10	A	test_parseDeleteWithValue	Pass
11	A	test_parseDeleteWithoutValue	Pass
12	A	test_parseLoadData	Pass
13	A	test_parseLoadNoData	Pass
14	A	test_parseUpdateWithData	Pass
15	A	test_parseUpdateWithNoData	Pass
16	A	test_runExit	Pass
17	A	test_runSortWithData	Pass
18	A	test_runSortWithNoData	Pass
19	A	test_unknownCmd	Pass
20	B	test_FGetter	Pass
21	B	test_FSetter	Pass
22	B	test_IOError	Pass
23	B	test_fileNotFoundError	Fail
24	B	test_loadFile	Pass
25	C	test_FGetter	Pass
26	C	test_FSetter	Pass
27	C	test_sortData	Pass
28	D	test_FGetter	Pass
29	D	test_FSetter	Pass
30	D	test_GGetter	Pass
31	D	test_GSetter	Pass
32	D	test_deleteData	Pass
33	D	test_insertData	Pass
34	D	test_updateData	Fail
35	E	test_exit	Pass
36	F	test_displayData	Pass
37	G	test_updateData	Pass
38	G	test_updateDataFileNotFound	Pass

Additionally, a coverage report is provided below:

Figure 1: Coverage results for non-incremental testing

 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

Out of the 38 tests, only 2 tests (test 23 and test 34) failed. We had 100% code coverage on all modules except for ModuleB where lines 22 to 26 can not be reached at all. We will explain the failures of the two tests in the following section

3 Failures

3.1 Test 23

Test 23, named test_fileNotFoundError, in module B failed. We tried to trigger a FileNotFoundError, but as mentioned earlier, that section of code is impossible to reach since an error would be caught by the earlier IOError exception as shown below:

```

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

```

As you can see, the FileNotFoundError can not be reached. In order to reach this section of code, we need to switch the IOError and FileNotFoundError statements like the following:

```

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

```

3.2 Test 34

Test 34, name test_updateData, in module D failed. In the test, we used the following data where we were trying to update data at index 2.

<pre>data = [('A', '1'), ('B', '2'), ('C', '3'), ('D', '4'),]</pre>	<pre>expected = [('A', '1'), ('B', '2'), ('Z', '22'), ('D', '4'),]</pre>
---	--

This test failed however due to the following line in updateData in ModuleD where when index used, it is incremented by one which produces unexpected results. It does not update the index passed in and instead updates the index above it.

```
data[index+1]=Entry(name, number)
```

To fix this, we simply remove the +1 in the code like the following:

```
data[index]=Entry(name, number)
```

4 Discussion

Through non-incremental testing, we were able to effectively test the program. We were able to find several errors in our program. This type of testing is good at making sure individual modules are tested and that the modules are also working well together. This type of testing can seem impractical for large projects since it requires a lot of effort to write tests for each module and sometimes there were tests that tested very similar functionality. Creating mocks of modules does seem to be an effective way to mock the functionality of modules to isolate the testing of the module.

5 Conclusion

The purpose of this lab was to become familiar with integration white-box testing. The goal of this lab was to teach us to use Python and Python unittest unittest.mock for integration testing. Integration testing is a form of unit testing. We used non-incremental testing also known as big bang testing where every module is tested individually and then the system is tested as a whole. Usually integration testing involves using stubs and drivers to isolate modules to be tested individually.

In this lab, we were able to test modules A-F using non-incremental testing and wrote tests for each module as well as testing the system as a whole. We were able to find 2 failures in the program and explained possible reasons to how those failures may be fixed. Through this lab, we realized that big bang testing is somewhat effective in testing a program although becomes a lot of work for larger systems.