ECE 322 Lab 4

Integration Testing #1 (Non- Incremental Testing)

Introduction

The objective of this lab was to familiarize ourselves with the is integration white box testing technique using Python and Python unittest and unittest.mock for non-incremental integration testing. Individual software modules are combined and tested as a group in integration testing.

Integration testing occurs after unit testing and before system testing. Thus, integration testing takes as its input modules that have been unit tested, groups them in larger aggregates and then applies tests defined in an integration test plan (i.e., a test suite in our case). About 70% of testing resources are spent on integration testing. There are two main approaches in integration testing – Non-incremental testing (being conducted in this lab) and incremental testing (conducted in the next lab).

Non-incremental testing (Big Bang testing) tests each module independently and combines the modules while incremental testing (Top down/Bottom up testing) combines the next module to be tested with the set of the previously tested modules before it is tested. Non-incremental testing is generally done in either a bottom up or top down method.

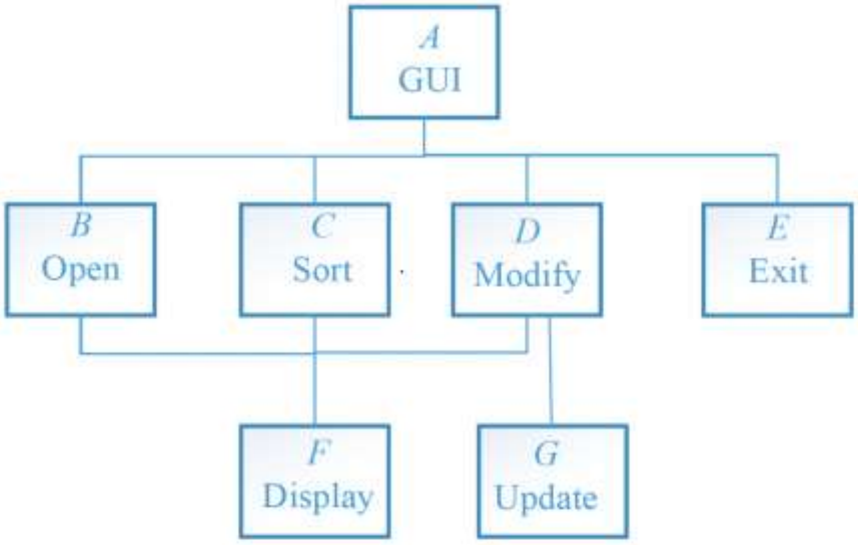
In this lab, we focused on non-incremental testing where we tested modules as a stand-alone entity using stubs and drivers. Stubs are used as a stand in for lower-level modules that are not currently under test and they simply make an assertion so that the test case can ensure it was called or they return a dummy value. On the other hand, a driver is a piece of testing code which allows for the possibility to call a sub-module of an application independently. A driver code may require a stub setup and object initialization.

In this lab, we used mocking frameworks to allow for the easy creation of mock objects. These objects were used as stubs in integration tests in the lab. To use mock objects, the python package - unittest.mock was used. Some unit tests were also written for the modules using the python package – unittest.

Task

1.Description of what the program does:

We were given a simple database system, constructed in a modular fashion: Module A invokes Module B Module C, Module D, and Module E. Module D invokes Module F, and Module G. Modules B and C also invoke Module F. This hierarchy of modules is shown below:



Two elements – String name and String Phone number comprised the entries of this database. The Files for this database can only contain one entry per line where the elements of each entry should be comma separated. The modules in the program do the following:

<u>Module A:</u> It is a command line interface which processes strings from the command. It also delegates functionality to sub-modules B,C,D and E.

<u>Module B:</u> Opens a data file

<u>Module C:</u> Sorts the records based on the first name

<u>Module D:</u> Modifies a record using three different methods – insertData(), updateData() and deleteData(). It also uses the function from Module F – displayData() and function updateData() from Module G.

<u>Module E:</u> Exits the program.

<u>Module F:</u> Displays the current data and is used by Module D.

<u>Module G:</u> Updates the data and is used by Module D.

2. Definition of the problem/task and the testing methods used:

Our task was to test the program (explained above) using non-incremental integration testing (Big Bang testing). Unit testing was used to cover the full functionality of each module. Stubs and drivers were also implemented and finally all the test cases were included in a test suite to implement integration testing.

Mock objects were used from the unittest.mock package to mock the objects from the modules that were invoked by other modules as a mock object substitutes and imitates a real object within a testing environment. It also controls our code's behaviour during testing.Thus, mock objects were created and used for the following modules – Module G, Module F, Module E, Module D, Module C and Module B.

Various assertions were made, both in unit testing and stubs using mock objects. The lower-level modules in the hierarchy were not tested using stubs but only were tested using unit testing. This is simply because they were not invoking any modules and thus, didn't need any mocked object behavior. For the remaining modules, both unit testing and stubs were used for testing.

3. Test Case Tables (Per Module) :

- **Module G:**

Test ID	Test	Description	Expected Result	Actual Result
1	def testUpdateData(self): self.G = ModuleG() openF = "data.txt" dataT = ["Jacob,5656"] self.G.updateData(openF,dataT)	Unit test for testing updateData() from Module G	Ran 1 test in 0.000s OK	'str' object has no attribute 'name'

- **Module F:**

Test ID	Test	Description	Expected Result	Actual Result
1	def testDisplayData(self): self.F = ModuleF() dataT = ["Jacob,5656","Joshua,1324"]	Unit test for testing displayData() from Module F	Ran 1 test in 0.000s OK Current Data:	Ran 1 test in 0.000s OK Current Data:

	self.F.displayData(dataT)		------------- ------------- ------------- ------------- ------ 1 Jacob,5656 2 Joshua,132 4 ----------------------- ----------------------- ------------	------------- ------------- ------------- ------------- ------ 1 Jacob,5656 2 Joshua,132 4 ----------------------- ----------------------- ------------

- **Module E:**

Test ID	Test	Description	Expected Result	Actual Result
1	def testExitProgram(self): self.E = ModuleE() self.E.exitProgram()	Unit test for testing exitProgram () from Module E	Ran 1 test in 0.000s OK	E ======== ======== ======== ERROR: testExitProgram (__main__.TestModuleE) ----------------------- ----------------------- ----------------------- sys.exit() SystemExit ------------- ------------- ------------- ------------- ------------- ----- Ran 1 test in 0.016s

				FAILED (errors=1)

- **Module D:**

Test ID	Test	Description	Expected Result	Actual Result

1	def testInsertData(self):
	n = "Gurbani"
	num = "0909"
	d = ["Gurbani,0900"]
	f = "test.txt"
	
	self.F.displayData()
	self.G.updateData()
	val = self.D.insertData(d, n, num, f)
	
	print(self.F.displayData.assert_called()) #Expected to print None
	
	print(self.G.updateData.assert_called()) #Expected to print None
	
	self.assertEqual(val,d,True)
		Test for insertData() using mock object for Module F and Module G	Ran 1 test in 0.000s
			
			OK
				Ran 1 test in 0.000s
				
				OK

2	def testUpdateData(self):
	n = "Gurbani"
	num = "0909"
	d = ["Gurbani,0900"]
	f = "test.txt"
	i = 0
		Test for updateData( ) using mock object for Module F and Module G	Ran 1 test in 0.000s
			
			OK
				======
				ERROR: testUpdateData (__main__.TestModuleD)
				------------------ 	
				IndexError: list assignment

	self.F.displayData()
	
	self.G.updateData()
	
	val = self.D.updateData(d,i,n,num,f)
	
	print(self.F.displayData.assert_called()) #Expected to print None
	
	print(self.G.updateData.assert_called()) #Expected to print None
	
	self.assertEqual(val,d,True)
				index out of range
				
				-------------------- 	

3	def testDeleteData(self):
	
	d = ["Gurbani,0900"]
	
	f = "test.txt"
	
	i = 0
	
	self.F.displayData()
	
	self.G.updateData()
	
	val = self.D.deleteData(d, i,f)
	
	print(self.F.displayData.assert_called()) # Expected to print None
	
	print(self.G.updateData.assert_called()) # Expected to print None
		Test for deleteData() using mock object for Module F and Module G	Ran 1 test in 0.000s
			
			OK
				Ran 1 test in 0.000s
				
				OK

	self.assertEqual(val, d, True)			
		On running the whole test file	Ran 3 tests in 0.017s OK	Ran 3 tests in 0.003s FAILED (errors=1)

- **Module C:**

Test ID	Test	Description	Expected Result	Actual Result

1	def testSortData(self):
	d = ["Gurbani,1100", "Lincoln,7908"]
	self.F.displayData()
	val = self.C.sortData(d)
	
	print(self.F.displayData.assert_called()) #Expected to print None
	
	self.assertEqual(val,d,True)
		Test for sortData() using mock object for Module F	Ran 1 test in 0.000s
			
			OK
				AttributeError: 'str' object has no attribute 'name'
				
				-- 	
				Ran 1 test in 0.013s
				
				FAILED (errors=1)

- **Module B:**

Test ID	Test	Description	Expected Result	Actual Result

1	def testLoadFile(self):
	f = "data.txt"
	self.F.displayData()
		Test for loadFile() using mock	Ran 1 test in 0.000s	Ran 1 test in 0.000s
				
				OK

	val = self.B.loadFile(f)
	
	self.F.displayData.assert_called() #Expected to print None
	
	self.assertEqual(val,[],True)
		object for Module F	OK	

- **<u>Module A:</u>**

Test ID	Test	Description	Expected Result	Actual Result

1	def testParseDelete(self):
	i = 0
	self.D.deleteData()
	val = self.A.parseDelete(i)
	
	self.D.deleteData.assert_called()
	self.assertTrue(val)
		Test for parseData() using mock object for Module D	Ran 1 test in 0.000s
			
			OK
				Ran 1 test in 0.000s
				
				OK

2	def testDisplayHelp(self):
	val = self.A.displayHelp()
	self.assertTrue(val)
		Unit Test for displayHelp()	Ran 1 test in 0.000s
			
			OK
				Ran 1 test in 0.000s
				
				OK

3	def testParseLoad(self):
	self.B.loadFile()
	f = "data.txt"
		Test for parseLoad() using mock object for Module B	Ran 1 test in 0.000s
			
			OK
				Ran 1 test in 0.000s
				
				OK

	self.B.loadFile.assert_called()
	val = self.A.parseLoad(f)
	self.assertTrue(val)			

4	def testParseAdd(self):
	n = "Paul"
	num = "1234"
	self.D.insertData()
	self.D.insertData.assert_called()
	val = self.A.parseAdd(n,num)
	self.assertTrue(val)
		Test for parseAdd() using mock object for Module D	Ran 1 test in 0.000s
			OK
				Ran 1 test in 0.000s
				OK

5	def testRunSort(self):
	self.C.sortData()
	self.C.sortData.assert_called()
	val = self.A.runSort()
	self.assertTrue(val)
		Test for runSort() using mock object for Module C	Ran 1 test in 0.000s
			OK
				Ran 1 test in 0.000s
				OK

6	def testParseUpdate(self):
	i = 0
	n = "Gurbani"
	num = "8989"
	self.D.updateData()
	self.D.updateData.assert_called()
	val = self.A.parseUpdate(i,n,num)
		Test for parseUpdate() using mock object for Module D	Ran 1 test in 0.000s
			OK
				Ran 1 test in 0.000s
				OK

	self.assertTrue(val)			

7	def testRunExit(self):
	    self.E.exitProgram()
	
	self.E.exitProgram.assert_called()
		Test for runExit() using mock object for Module E	Ran 1 test in 0.000s
			
			OK
				Ran 1 test in 0.000s
				
				OK

		On running the whole test file	Ran 7 tests in 0.004s OK Available Commands: load <filepath> add <name> <number> update <index> <name> <number> delete <index> sort exit Process finished with exit code 0	Ran 7 tests in 0.004s OK Available Commands: load <filepath> add <name> <number> update <index> <name> <number> delete <index> sort exit Process finished with exit code 0

- The test suite is included in the code and gave the following results:

```
C:\Users\gurba\AppData\Local\Microsoft\WindowsApps\python3.9.exe
C:/Users/gurba/Lab4_src/tests/TestRunner.py
Available Commands:
load <filepath>
add <name> <number>
update <index> <name> <number>
delete <index>
sort
exit
testDisplayHelp (TestModuleA.TestModuleA) ... ok
```

```
testParseAdd (TestModuleA.TestModuleA) ... ok
testParseDelete (TestModuleA.TestModuleA) ... ok
testParseLoad (TestModuleA.TestModuleA) ... ok
testParseUpdate (TestModuleA.TestModuleA) ... ok
testRunExit (TestModuleA.TestModuleA) ... ok
testRunSort (TestModuleA.TestModuleA) ... ok
testLoadFile (TestModuleB.TestModuleC) ... ok
testSortData (TestModuleC.TestModuleC) ... ERROR
None
None
None
None
Program Exit !
Current Data:
----------------------------------------------------------
1 Jacob,5656
2 Joshua,1324
----------------------------------------------------------
testDeleteData (TestModuleD.TestModuleD) ... ok
testInsertData (TestModuleD.TestModuleD) ... ok
testUpdateData (TestModuleD.TestModuleD) ... ERROR
testExitProgram (TestModuleE.TestModuleE) ... ERROR
testDisplayData (TestModuleF.TestModuleF) ... ok
testUpdateData (TestModuleG.TestModuleG) ... ERROR

======================================================================
ERROR: testSortData (TestModuleC.TestModuleC)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\gurba\Lab4_src\tests\TestModuleC.py", line 14, in testSortData
    val = self.C.sortData(d)
  File "C:\Users\gurba\Lab4_src\modules\ModuleC.py", line 8, in sortData
    data.sort(key=lambda x: x.name)
  File "C:\Users\gurba\Lab4_src\modules\ModuleC.py", line 8, in <lambda>
    data.sort(key=lambda x: x.name)
AttributeError: 'str' object has no attribute 'name'

======================================================================
ERROR: testUpdateData (TestModuleD.TestModuleD)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\gurba\Lab4_src\tests\TestModuleD.py", line 36, in testUpdateData
    val = self.D.updateData(d,i,n,num,f)
  File "C:\Users\gurba\Lab4_src\modules\ModuleD.py", line 19, in updateData
    data[index+1]=Entry(name, number)
```

```
IndexError: list assignment index out of range

======================================================================
=
ERROR: testExitProgram (TestModuleE.TestModuleE)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\gurba\Lab4_src\tests\TestModuleE.py", line 8, in testExitProgram
    self.E.exitProgram()
  File "C:\Users\gurba\Lab4_src\modules\ModuleE.py", line 5, in exitProgram
    sys.exit()
SystemExit

======================================================================
=
ERROR: testUpdateData (TestModuleG.TestModuleG)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\gurba\Lab4_src\tests\TestModuleG.py", line 10, in testUpdateData
    self.G.updateData(openF,dataT)
  File "C:\Users\gurba\Lab4_src\modules\ModuleG.py", line 7, in updateData
    FileWriter.write(val.name + "," + val.number+ "\n")
AttributeError: 'str' object has no attribute 'name'

----------------------------------------------------------------------
Ran 15 tests in 0.007s

FAILED (errors=4)
```

Discussion

Four failed tests (highlighted in red above) were observed while testing the given program. They are as follows:

1. In Module G: While performing a unit test on Module G, the error - 'str' object has no attribute 'name' was observed. This error occurred because of the way the method – updateData() was accessing the elements of the list data. The method did the following:

```
            for val in data:
    FileWriter.write(val.name + "," + val.number+ "\n")
```

   Here the elements from the list data are being treated like they have two attributes name and number which is not true. A solution to make the test case pass would be that the string attributes are handled correctly, for instance – each element in the list is split into two elements using the “,” and then the first element of the split string can be used as name the latter as number.

2. In Module E: While performing the unit test on Module E, the error - sys.exit() SystemExit was observed. This is not basically an error as whenever sys.exit() is called, it raises an execption SystemExit which basically that our progam had a behavior such that we want had to stop it and raise an error.

3. In Module D: Test Case 2 gives an error - IndexError: list assignment index out of range . This is because the method – updateData() in Module D, is initializing a list element as : data[index+1]=Entry(name, number). The error is raised when we try to assign an item to an index position that does not exist. To solve this, we should use append to add an item to data.

4. In Module C: The same error, like in the case of Module G was observed - AttributeError: 'str' object has no attribute 'name' in method – sortData(). This is because the string element here is being treated like it has an attribute name, but it does not. Thus, the error can be solved by using the same method as explained for the first case.

Integration testing is not an effective way of isolating individual modules and testing them, the best fit for that is unit testing. However, integration testing is a good fit for integrating all units/modules to ensure that our software/program works properly as all units should integrate and perform as they're expected to. Integration testing is the next step after unit testing where we test what happens when units start interacting because if units successfully pass tests via unit testing does not guarantee that they will work correctly together when integrated. Thus, integration testing helps observe any problems that arise when the units/modules are assembles at the early stages to lower the cost of making any mistake. While integration testing allows for simultaneous testing of multiple modules and is very convenient, practical, cost-efficient and can take place at anywhere in the Software Development Life Cycle, the Big Bang type of integration has its own cons. Non-incremental/Big Bang testing is not really helpful for large-scale systems. This is because, in case all the units/modules form a huge system, it can be very difficult to find the source of defects in such a giant system. Also, if there are numerous units in a system, reviewing all of them could be very time consuming. Thus, big bang/non-incremental integration testing is good for testing small systems as it allows to find errors very quickly, however, it is not a great fit for large-scale systems.

Drivers and stubs are effective for isolating modules. This is because drivers and stubs enable us to unit test a module without other dependent code modules being available. Stubs are very helpful in producing an expected output while drivers are use to send a required input to the module. Drivers and stubs help reduce the risk of taking errors to the further stages in development as they catch defects early. Also, they help identify the functional and non-function parts of a module that contains objects of other dependent modules.

Mocking objects is a good way of implementing stubs and drivers in the testing, as done in this lab. The package unittest.mock helps mock objects from modules that depend on other modules. Sometimes, a temporary change in the behavior of external services causes intermittent failures within a test suite. Thus, mock objects are very helpful in such cases, as they help us to test our code in a controlled environment by substituting and imitating real objects within a testing environment.

Conclusion

In this lab, we learned about non-incremental integration testing which occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates and then applies tests defined in an integration test plan (i.e., a test suite in our case).Non-incremental testing (Big Bang testing) tests each module independently and combines the modules while incremental testing (Top down/Bottom up testing) combines the next module to be tested with the set of the previously tested modules before it is tested. We used mocking frameworks for using stubs and drivers, to allow us for the easy creation of mock objects by using the python package - unittest.mock. Unit tests were also written for the modules and finally, every test module was integrated in a test suite.

A few errors were observed due to mishandling of list elements, which could be solved using list operations such as append() and split(). We observed that although, big bang/non-incremental integration testing is good for testing small systems as it allows to find errors very quickly, however, it is not a great fit for large-scale systems. This is because, looking for defects in a large-scale system comprising of numerous units can be very difficult and time consuming. Drivers and stubs are an effective way of isolating modules as they allow the testing of modules with code of other dependent modules not being available. Stubs are very helpful in producing an expected output while drivers are use to send a required input to the module. Implementing stubs and drivers using mock objects is a great way as mock objects are very helpful in testing our code in a controlled environment by substituting and imitating real objects within a testing environment.