ECE 322 Lab 5

Integration Testing #2 (incremental testing)

Introduction

The objective of this lab was to familiarize ourselves with the integration white box testing technique using Python and Python unittest and unittest.mock for incremental integration testing – Bottom up and Top Down. 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 the previous lab) and incremental testing (conducted in this lab).

Incremental Testing combines the next module to be tested with the set of previously tested modules before running tests. It is done, either in a bottom up or top-down method. In a bottom-up method, the lowest level modules are tested in isolation, then higher- and higher-level modules are incrementally added. In top-down testing, the highest-level modules are tested in isolation, stubbing out lower-level functionality, and then lower modules are added incrementally. In this lab, we tested a simple database system consisting of several modules using both bottom-up and top-down incremental testing.

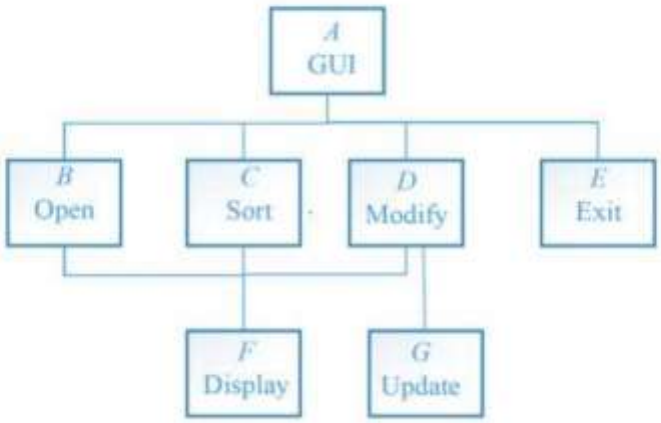
Stubs were used in this lab as a stand in for lower-level modules that were not currently under test. Stubs simply make an assertion so that the test case can ensure it was called or they return a dummy value. In this lab, we used mocking frameworks to allow for the easy creation of mock objects. These 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.

Part 1

Q1: Definition of the problem, including what the program does, what testing method is used:

Answer:

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:

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

Module B: Opens a data file

Module C: Sorts the records based on the first name

Module D: 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.

Module E: Exits the program.

Module F: Displays the current data and is used by Module D.

Module G: Updates the data and is used by Module D.

Our task was to test the program (explained above) using two incremental integration testing techniques – bottom-up and top-down. Unit testing was used to cover the full functionality of a few modules – F,G and E. Mock objects were used from the unittest.mock package to mock the objects from certain lower-level modules that were treated as stubs. In top-down testing, mock objects were used for modules – Module G, Module F, Module E, Module D, Module C and Module B while none were used in bottom-up testing. Various assertions were made, both in unit testing and stubs using mock objects. Finally, all the test cases were included in two separate test suites for bottom-up and top-down testing respectively.

Q 2: List module/modules that you will be test in order according to test strategy:

Answer:

Top Down:

- Module A (using stubs for Modules B,C,D,E)

- Modules AB, AC, AD, AE (using stubs for Modules F,G). These modules can be tested altogether as well, however, it was recommended that the tests be conducted separately.

- Modules ABCDEFG altogether

Bottom Up:

- Module F and Module G

- Module BF, CF, DFG and E

- Modules ABCDE altogether

Q3 Test Case Tables (Top Down and Bottom Up):

Answer:

Top-Down Testing

TestModuleA:

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)	Delete an entry from the file	IndexError: list assignment index out of range
2	def testDisplayHelp(self): val = self.A.displayHelp() self.assertTrue(val)	Pass	Pass
3	def testParseLoad(self): self.B.loadFile() f = "data.txt"	Load data into the file	Loaded data into the file

	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)	Insert data into the file	Inserted data into the file
5	def testRunSort(self): self.C.sortData() self.C.sortData.assert_called() val = self.A.runSort() self.assertTrue(val)	Sort by name	Sorted by name
6	def testParseUpdate(self): i = 0 n = "Gurbani" num = "8989" self.D.updateData()	Update data in the file	IndexError: list assignment index out of range

	self.D.updateData.assert_called() val = self.A.parseUpdate(i,n,num) self.assertTrue(val)		
7	def testRunExit(self): self.E.exitProgram() self.E.exitProgram.assert_called()	Exit the program	SysExit Error

TestModuleAB:

Test Id	Test Description	Expected Result	Actual Result
1.	def testParseLoad(self): f = "data.txt" d = self.B.loadFile(f) self.A._data = d val = self.A.parseLoad(f) self.assertTrue(val)	Load data into the file	Loaded data into the file

TestModuleAC:

Test Id	Test Description	Expected Result	Actual Result
1.	def testRunSort(self): data1 = [Entry("Jacob", "5656")]	Sort by name	Sorted by name

	val1 = self.C.sortData(data1) self.A._data = val1 ret_val = self.A.runSort() self.assertTrue(ret_val)		

TestModuleAD:

Test Id	Test Description	Expected Result	Actual Result
1.	def testParseDelete(self): i = 0 j = 1 dataT = [Entry("Jacob","5656")] f = "data.txt" d = self.D.deleteData(dataT,i,f) self.A._data = d val = self.A.parseDelete(j) self.assertTrue(val)	Delete data from file	IndexError: list assignment index out of range
2.	def testParseAdd(self): n = "Paul" num = "1234" f = "data.txt" dataT = [Entry("Jacob","5656")] d = self.D.insertData(dataT, n, num,f) self.A._data = d	Add data into the file.	Added data into the file.

	val = self.A.parseAdd(n,num) self.assertTrue(val)		
3.	def testParseUpdate(self): i = 0 j = 2 dataT = [Entry("Jacob", "5656")] f = "test.txt" n = "Gurbani" num = "8989" d = self.D.updateData(dataT,j,n,num,f) self.A._data = d val = self.A.parseUpdate(i,n,num) self.assertTrue(val)	Update data in the file.	IndexError: list assignment index out of range

TestModuleAE:

Test Id	Test Description	Expected Result	Actual Result
1.	def testRunExit(self): self.E.exitProgram() with self.assertRaises(SystemExit): self.A.run("exit")	Exit the program.	sys.exit() error

TestModulesABCDEFG:

Test Id	Test Description	Expected Result	Actual Result
1	def testParseLoad(self): f = "data.txt"	Load data into the file	Loaded data into the file

	d = self.B.loadFile(f) self.A._data = d val = self.A.parseLoad(f) self.assertTrue(val)		
2	def testRunSort(self): data1 = [Entry("Jacob", "5656")] val1 = self.C.sortData(data1) self.A._data = val1 ret_val = self.A.runSort() self.assertTrue(ret_val)	Sort by name	Sorted by name
3	def testParseDelete(self): i = 0 j = 1 dataT = [Entry("Jacob","5656")] f = "data.txt" d = self.D.deleteData(dataT,i,f) self.A._data = d val = self.A.parseDelete(j) self.assertTrue(val)	Delete data from file	IndexError: list assignment index out of range
4	def testParseAdd(self): n = "Paul" num = "1234" f = "data.txt" dataT = [Entry("Jacob","5656")] d = self.D.insertData(dataT, n, num,f) self.A._data = d	Add data into the file.	Added data into the file.

	val = self.A.parseAdd(n,num) self.assertTrue(val)		
5	def testParseUpdate(self): i = 0 j = 2 dataT = [Entry("Jacob", "5656")] f = "test.txt" n = "Gurbani" num = "8989" d = self.D.updateData(dataT,j,n,num,f) self.A._data = d val = self.A.parseUpdate(i,n,num) self.assertTrue(val)	Update data in the file.	IndexError: list assignment index out of range
6	def testRunExit(self): self.E.exitProgram() with self.assertRaises(SystemExit): self.A.run("exit")	Exit the program.	sys.exit() error

Bottom - Up Testing

TestModuleF:

Test ID	Test Description	Expected Result	Actual Result
1	def testDisplayData(self): self.F = ModuleF() dataT = [Entry("Jacob", "5656")] self.F.displayData(dataT)	Display data	Displayed data

TestModuleG:

Test ID	Test Description	Expected Result	Actual Result
1	def testUpdateData(self): self.G = ModuleG() openF = "data.txt" dataT= [Entry("Jacob","5656")] self.G.updateData(openF,dataT)	Update data to the file	Updated data to the file

TestModuleE:

Test ID	Test Description	Expected Result	Actual Result
1	def testExitProgram(self): self.E = ModuleE() self.E.exitProgram()	Exit the program.	sys.exit() error

TestModuleBF:

Test ID	Test Description	Expected Result	Actual Result
1	def testLoadFile(self): f = "data.txt" val = self.B.loadFile(f) self.F.displayData(val)	Load data to the file	Loaded data to the file

TestModuleCF:

Test ID	Test Description	Expected Result	Actual Result
1	def testSortData(self): d = [Entry("Gurbani", "1800")] val = self.C.sortData(d) self.F.displayData(val)	Sort data by name	Sorted data by name

	self.assertEqual(val,d,True)		

TestModuleDFG:

Test ID	Test Description	Expected Result	Actual Result
1	def testInsertData(self): n = "Gurbani" num = "1800" d = [Entry("Gurbani", "1800")] f = "test.txt" val = self.D.insertData(d, n, num, f) self.F.displayData(val) self.G.updateData(f,val) self.assertEqual(val,d,True)	Insert data to the file	Inserted data to the file
2	def testUpdateData(self): i = 0 n = "Harry" num = "1900" d = [Entry("Gurbani", "1800")] f = "test.txt" val = self.D.updateData(d,i, n, num, f) self.F.displayData(val) self.G.updateData(f, val) self.assertEqual(val, d, True)	Update data in the file.	IndexError: list assignment index out of range
3	def testDeleteData(self):	Delete data from file	Deleted data from file

	d = [Entry("Gurbani", "1800")] f = "test.txt" i = 0 val = self.D.deleteData(d, i,f) self.F.displayData(val) self.G.updateData(f, val) self.assertEqual(val, d, True)		

TestModulesABCDE:

Test ID	Test Description	Expected Result	Actual Result
1	def testParseLoad(self): f = "test.txt" d = self.B.loadFile(f) self.A._data = d val = self.A.parseLoad(f) self.assertTrue(val)	Load data into the file	Loaded data into the file
2	def testRunSort(self): data1 = [Entry("Jacob", "5656")] val1 = self.C.sortData(data1) self.A._data = val1 ret_val = self.A.runSort() self.assertTrue(ret_val)	Sort by name	Sorted by name
3	def testParseDelete(self): i = 0 j = 1 dataT = [Entry("Jacob","5656")]	Delete an entry from the file	IndexError: list assignment index out of range

	f = "data.txt" d = self.D.deleteData(dataT,i,f) self.A._data = d val = self.A.parseDelete(j) self.assertTrue(val)		
4	def testParseAdd(self): n = "Jacob" num = "5656" f = "data.txt" dataT = [Entry("GB","56")] d = self.D.insertData(dataT, n, num,f) self.A._data = d val = self.A.parseAdd(n,num) self.assertTrue(val)	Add data to file	Added data to file
5	def testParseUpdate(self): i = 0 j = 2 dataT = [Entry("Jacob", "5656")] f = "test.txt" n = "Gurbani" num = "8989" d = self.D.updateData(dataT,j,n,num,f) self.A._data = d val = self.A.parseUpdate(i,n,num) self.assertTrue(val)	Update data in the file.	IndexError: list assignment index out of range
6	def testRunExit(self):	Exit the program.	sys.exit() error

	self.E.exitProgram() with self.assertRaises(SystemExit): self.A.run("exit")		

Discussion of errors:

There were mainly two types of errors identified while testing using both bottom-up and top-down approach (highlighted in red). Whenever a test using module E was conducted, 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.

Some list index out of range errors were also observed while running tests on updating data and deleting data using ModuleD. This is because the method – updateData() and deleteData() in Module D, is initializing a list element as : data[index+1]=Entry(name, number) and del data[index]. Such errors are 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 and put conditions to check if an item in the list at a given index exists in the first place or not.

Q4: Discussion

Comments on the effectiveness of integration testing

Q4.1: Is module isolation effective?

Answer: Module isolation is effective because it enables us to verify the input/output behavior of the test objects. It ensures that the test object's functionality is working correctly and completely fine as per the desired specifications. Module isolation solves local dependency issues, locates tricky bugs, improves design and performance, and promotes clean production. It may not be completely ideal in cases where the number of modules is extremely large as we'll have to isolate a huge number of modules in that case. However, module isolation has benefits of its own as mentioned above.

Q4.2 : What does each type of integration testing target?

Answer:

- Bottom-Up Incremental Testing: Bottom-Up testing targets integration from the lowest level modules. In this integration testing approach, integration testing takes place from bottom to top so, the system integration begins with lowest level modules. This testing is usually targeted towards object-oriented programming languages. In this testing the lower level modules are tested first then the higher level modules are tested and then the modules are integrated accordingly. Thus, this approach targets the small components and moves to big components.

- Top-Down Incremental Testing: Top-Down testing targets integration from the highest-level modules. In this integration testing approach, integration testing takes place from top to bottom so, the system integration begins with top level modules. This testing is usually targeted towards structure/procedure-oriented programming languages. In this testing the higher level modules are tested first then the lower level modules are tested and then the modules are integrated accordingly.Thus, this approach targets the big components and moves to small components.

Q4.3: Would this type of testing scale well to a large scale system?

Answer:

Incremental 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. In incremental integration testing, defects are detected in smaller modules rather than denoting errors then editing and re correcting large files later on and it is more flexible and cost efficient as per requirements and scopes. Also, a customer gets the chance to respond to each building in this case. However, in a large-scale system, this type of testing may not do well as stubs and drivers are developed in this type of testing which need constant updates before being used in the tests. Thus, this process can be time consuming in a large scale system environment. Due to the heterogeneity of large scale systems, coordinating testing activities can be very challenging. Thus, integration testing can be challenging in such a case, since some components may not behave as expected or may not be ready for integration testing. So, we can conclude that such type of testing may not scale well to a large scale system.

Q4.4: Are stubs and drivers useful and effective?

Answer:

Drivers and stubs are useful and 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 used 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. 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. However, as stubs and drivers need constant updates before being used in the tests which can be time-consuming in many cases.

Q4.5 : Which type of integration testing do you think is best?

Answer:

Integration testing is of different types like big-bang, top-down, bottom-up, sandwich testing etc. However, there is no solid guide which classifies one of them as the best over the others. Each type has its own pros and cons, however, based on the type of system we are testing, we can pick a type of integration testing that could be best for that particular case.

Different factors can be considered when ranking one of them as best. The factors to consider are: the amount of test harness (number of stubs &drivers), location of critical parts in the system, availability of the hardware and the components, and scheduling concerns. Bottom-up approach can be good for object oriented design methodologies but the test driver interfaces must match component interfaces in this case. This approach can also postpone the detection of errors until the end of testing. In, top-down approach, the test cases can be defined in terms of functions examined and we need to maintain correctness of test stubs where writing stubs can be difficult.

Keeping all the abovementioned factors in mind, a best approach can be chosen based on the type of system under test. In the case of our lab where we tested a simple database, I think the best type of integration testing was bottom-up testing as it did not require any drivers and so the amount of test harness was significantly lower than top-down testing where 6 stubs were needed.

Q4.6 : Which would be the most appropriate for a test driven development environment?

Answer:

Top-down incremental integration testing would be most appropriate for a test-driven development environment. This is because, it makes sure that the integrated modules/components work properly. It can be started once the modules to be tested are available. It does not require the other modules to be completed for testing to be done, as stubs can be used for the same. In this case, test cases can be defined in terms of functions examined. As, top-level components are usually important and cannot be neglected up to the end of testing, a bottom-up approach may not be appropriate for test driven development environments. Also, with bottom-up, the detection of design errors will have to be postponed until end of testing. Thus, the top-down approach will be suitable in this case.

Q4.7: Which would be the most appropriate for library development?

Answer:

Since library development, focuses on a narrow scope, and their API's also tend to be smaller and require fewer dependencies, thus either the non-incremental big-bang testing approach or bottom-up

incremental approach would work just fine in case of library development. Since there are fewer dependencies, not many stubs or drivers would be needed and even if the code is integrated at the end (like in big-bang) or the detection of design errors will have to be postponed until end of testing (like in bottom-up) it would not be as challenging or troublesome like in other types of developing environments.

Q4.8: Which is the easiest to maintain?

Answer:

The easiest to maintain type of integration testing depends on the system which is under test. Different factors can be considered when classifying one of them as the easiest to maintain. However, the most significant factor to consider while making such a decision is the amount of test harness i.e., the number of stubs & drivers. Writing and maintain stubs and drivers can be challenging. Stubs and rivers need constant updates before being used in the tests and thus, having a large number of stubs or drivers would make a testing approach very challenging to maintain.

In the case of our lab where we tested a simple database, I think that bottom-up testing was easiest to maintain as it did not require any drivers and so the amount of test harness was significantly lower than top-down testing where 6 stubs were needed.

- Q4.9: Additional comments: Consider these (Q4.1 to Q4.8) and similar questions in your discussion of integration testing to show you have a strong understanding of the underlying concepts and motivations of this testing technique.

Answer:

In this lab we focused on incremental integration testing – both bottom-up and top-down testing. We could observe the following about the two types of incremental testing: top-down integration testing is one of the approach of integration testing in which integration testing takes place from top to bottom and so, the system integration begins with top level modules whereas in bottom-up approach, testing takes place from bottom to top so, the system integration begins with the bottom level modules. In top-down testing, stubs are used for simulating the submodules if the invoked submodules are not developed while in bottom-up testing, drivers are used for simulating the main module if the main module is not developed. Top-down testing is beneficial if the significant defect occurs toward the top of the program while bottom-down is beneficial if the crucial flaws encounters towards the bottom of the program. While top-down approach works on big to small components, the bottom-up approach works on small to big components.

Code component:

Read the manual and submit your code in zip format

NB: It's important that you use a test suite for each testing strategy and that you follow all the instructions provided in the manual.

Stubs/Mock objects

- Students should make use of Mock objects in their tests
- Mock objects should be used to simulate module functionality where appropriate
- Mock objects should verify that methods are called when they are supposed to be (testing the integration of the modules).

Conclusion

Answer:

In this lab, we learned about incremental integration white box testing using both top-down and bottom-up testing approach. We used Python and Python unittest and unittest.mock functionalities for incremental integration testing. Incremental Testing combines the next module to be tested with the set of previously tested modules before running tests. In a bottom-up method, the lowest level modules are tested in isolation, then higher- and higher-level modules are incrementally added. In top-down testing, the highest-level modules are tested in isolation, stubbing out lower-level functionality, and then lower modules are added incrementally. In this lab, we tested a simple database system consisting of several modules using both bottom-up and top-down incremental testing.

Two types of errors were identified while testing using both bottom-up and top-down approach. The use of ModuleE yielded the error - sys.exit() SystemExit which is a raised execption - SystemExit which basically tells that our progam had a behavior such that we want had to stop it and raise an error.Some list index out of range errors were also observed while running tests on updating data and deleting data using ModuleD as we were trying 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 and put conditions to check if an item in the list at a given index exists in the first place or not.

We observed some notable differences between bottom-up and top-down testing. Top-down integration testing is one of the approach of integration testing in which integration testing takes place from top to bottom and so, the system integration begins with top level modules whereas in bottom-up approach, testing takes place from bottom to top so, the system integration begins with the bottom level modules. In top-down testing, stubs are used for simulating the submodules if the invoked submodules are not developed while in bottom-up testing, drivers are used for simulating the main module if the main module is not developed.

We also observed that in a large-scale system, this type of testing may not do well as stubs and drivers are developed in this type of testing which need constant updates before being used in the tests. Thus, this process can be time consuming in a large scale system environment. Also, integration testing can be challenging in such a case, since some components may not behave as expected or may not be ready for integration testing. We also noticed that there is no solid guide to determine which approach in integration testing is better than the other. However, different factors can be considered when ranking one of them as best or the easiest to maintain such as the amount of test harness (number of stubs & drivers) – the lower

the better, the location of critical parts in the system, availability of the hardware and the components, and scheduling concerns.